DISS. ETH No. 16774, 2006

**Algorithm Design and Analysis of Problems in Manufacturing, Logistics, and Telecommunications:**

# An Algorithmic Jam Session

A dissertation submitted to the
Swiss Federal Institute of Technology, ETH Zürich
for the degree of Doctor of Sciences

presented by
Dipl. Inform. Marc Nunkesser
born May 29, 1976 in Dortmund, Germany

accepted on the recommendation of
Prof. Dr. Peter Widmayer, ETH Zürich, examiner
Dr. Thomas Erlebach, University of Leicester, co-examiner
Dr. Riko Jacob, ETH Zürich, co-examiner

# Abstract

The focus of this thesis is on algorithmic questions that are directly linked to practical problems from diverse applications like manufacturing, train scheduling or telecommunications. We present four problem settings together with combinatorial problems (or mathematical models) that are at the core of the practical applications. From this we derive both interesting theoretical and relevant practical results, which we back up by experiments. In detail, the topics of this thesis are as follows.

**Sequential Vector Packing** We study a novel variant of the well known $d$-dimensional bin (or vector) packing problem that is motivated by an application from the manufacturing industry. Given a sequence of non-negative $d$-dimensional vectors, the goal is to pack these into as few bins as possible. In the classical problem the bin size vector is given and the sequence can be partitioned arbitrarily. We study a variation where the vectors have to be packed in the order in which they arrive. The bin size vector can be chosen once in the beginning, under the constraint that the coordinate-wise bounds sum up to at most a given total bin size. We give both theoretical results and practical algorithms that we test on the original data.

**Optimization of a Freight Train System** We consider the optimization of a Swiss freight train service that is operated as a (multi-) hub spoke system. This can be seen as a more complicated version of classical vehicle routing problems. We derive several mathematical models, consider some theoretical questions linked to the operations at the hub, and test our models on the real-world instance. For the mathematical models we use linear programming based optimization techniques like branch and cut and column generation.

**OVSF Code Assignment** Orthogonal Variable Spreading Factor (OVSF-) codes are used in UMTS to share the radio spectrum among several connections of possibly different bandwidth requirements. The combinatorial core of the OVSF code assignment problem is to assign some nodes of a complete binary tree of height $h$ (the code tree) to $n$ simultaneous connections, such that no two assigned nodes (codes) are on the same root-to-leaf path. A connection that uses a $2^{-d}$ fraction of the total bandwidth requires some code at depth $d$ in the tree, but this code assignment is allowed to change over time. We consider the one-step code assignment problem: Given an assignment, move the minimum number of codes to serve a new request. Minn and Siu propose the so-called DCA-algorithm to solve the problem

optimally. In contrast, we show that DCA does not always return an optimal solution, and that the problem is NP-hard. We present results on exact, approximation, online, and fixed parameter tractable algorithms.

**Joint Base Station Scheduling** Consider a scenario where base stations need to send data to users with wireless devices. Time is discrete and slotted into synchronous rounds. Transmitting a data item from a base station to a user takes one round. A user can receive the data item from any of the base stations. The positions of the base stations and users are modeled as points in the Euclidean plane. If base station $b$ transmits to user $u$ in a certain round, no other user within distance at most $\|b - u\|_2$ from $b$ can receive data in the same round due to interference phenomena. The goal is to minimize, given the positions of the base stations and users, the number of rounds until all users have their data. We call this problem the Joint Base Station Scheduling Problem (JBS) and consider it on the line (1D-JBS) and in the plane (2D-JBS). We study the complexity of 2D-JBS and approximation algorithms for both variants. Moreover, we analyze a special graph class of *arrow graphs* that arises in the one-dimensional setting.

# Zusammenfassung

Im Mittelpunkt dieser Arbeit stehen algorithmische Fragen, die durch praktische Probleme aus verschiedenen Gebieten wie Fertigung, Zug-Optimierung oder Telekommunikation motiviert sind. Wir stellen vier Anwendungen vor und dazu jeweils ein kombinatorisches Problem (oder ein mathematisches Modell), das den Kern dieser Anwendung ausmacht. Daraus entwickeln wir sowohl theoretisch interessante als auch praktisch relevante Resultate, die wir noch durch Experimente untermauern. Im Detail betrachten wir die folgenden Themen.

**Sequentielles Vector Packing** Wir untersuchen eine neuartige Variante des bekannten $d$-dimensionalen Bin (oder auch Vector) Packing Problems, welche durch eine Anwendung aus der Fertigung motiviert ist. Gegeben sei eine Sequenz von nicht-negativen $d$-dimensionalen Vektoren. Die Aufgabe besteht darin, diese in so wenig Bins („Kisten") wie möglich zu packen. Beim klassischen Bin Packing ist der Bin-Grössenvektor gegeben und die Sequenz kann beliebig angeordnet werden. Wir untersuchen eine Variation, bei der die Vektoren in der durch die Sequenz gegebenen Reihenfolge in die Bins gepackt werden müssen. Die Bin-Grösse kann anfangs gewählt werden, mit der Einschränkung, dass ihre koordinatenweise Summe eine vorgegebene Grösse nicht überschreiten darf. Wir präsentieren sowohl theoretische Resultate als auch praktische Algorithmen, die wir auf den realen Daten testen.

**Optimierung eines Frachtbahnsystems** Wir betrachten die Optimierung eines Schweizer Frachtbahnsystems, das als (Multi-) Nabe-Speiche System operiert. Das Problem kann als kompliziertere Variante klassischer Vehicle Routing Probleme gesehen werden. Wir entwickeln eine Sequenz von mathematischen Modellen für das Problem, betrachten einige theoretische Fragestellungen bezüglich des Ablaufs an der Nabe und testen unsere Modelle auf der realen Instanz. Für die mathematischen Modelle benutzen wir Optimierungstechniken, die auf linearer Programmierung beruhen wie branch and cut oder column generation.

**OVSF Code Zuweisung** Orthogonal Variable Spreading Factor (OVSF-) Codes werden in UMTS Netzwerken benutzt, um es den verschiedenen Benutzern innerhalb einer Funkzelle (mit potentiell verschiedenen Bandbreitenanforderungen) zu ermöglichen, gleichzeitig auf die vorhandene Bandbreite zuzugreifen. Der kombinatorische Kern des OVSF Code-Zuweisungsproblems besteht darin, Knoten eines vollständigen binären Baumes (der Code-Baum) der Höhe $h$ einer Menge von $n$ aktiven Verbindungen zu-

zuweisen, so dass keine zwei zugewiesenen Knoten (Codes) auf dem selben Wurzel-Blatt Pfad sind. Eine Verbindung, die einen Anteil von $2^{-d}$ an der gesamten Bandbreite benötigt, muss einen Code der Tiefe $d$ im Code-Baum zugewiesen bekommen. Diese Zuweisung ist nicht fix, sondern kann sich mit der Zeit ändern. Wir betrachten das Ein-Schritt Code-Zuweisungsproblem: Gegeben eine Code-Zuweisung, bewege die minimale Anzahl von Codes, um eine neue Anfrage zu bedienen. Minn und Siu haben den so genannten DCA-Algorithmus vorgestellt, um das Problem optimal zu lösen. Demgegenüber zeigen wir, dass DCA nicht immer die optimale Lösung liefert und dass das Problem NP-hart ist. Wir geben Resultate zu exakten, approximations-, online- und fixed-parameter Algorithmen.

**koordiniertes Funkmast Scheduling** Wir betrachten ein Szenario, in dem Funkmasten Daten an Benutzer mit Mobilgeräten senden. In unserem Modell betrachten wir die Zeit als diskretisiert und in Runden segmentiert. Die Übertragung eines Datenpakets von einem Funkmasten zu einem Benutzer benötigt eine Runde. Ein Benutzer kann Daten von jedem Funkmasten empfangen. Die Positionen der Funkmasten und Benutzer werden als Punkte in der Euklidischen Ebene betrachtet. Wenn in einer Runde Funkmast $b$ an Benutzer $u$ sendet, kann kein anderer Benutzer innerhalb einer Distanz von $\|b - u\|_2$ von $b$ ein Datenpaket empfangen, weil das Senden an $u$ zu Interferenz in diesem Bereich führt. Das Ziel besteht darin, für eine gegebene Konfiguration von Benutzern und Funkmasten das Senden von Daten so zu koordinieren, dass eine minimale Anzahl von Runden benötigt wird, um alle Benutzer zu bedienen. Wir nennen dieses Problem koordiniertes Funkmast Scheduling (JBS) und betrachten es auf einer Geraden (1D-JBS) und in der Ebene (2D-JBS). Wir untersuchen die Komplexität von 2D-JBS sowie Approximationsalgorithmen für beide Varianten. Darüber hinaus analysieren wir die Graphklasse der *arrow graphs*, die sich als Konfliktgraph in der eindimensionalen Variante ergibt.

# Acknowledgement

One of the good things about doing this PhD thesis has been that I rarely thought about it as the “work” that I have to do. The main reason why instead it mostly felt like real fun to do it were the many interesting, creative and enjoyable people with whom I had the pleasure to cooperate. I would like to express my gratitude to

Peter Widmayer, my supervisor, for his support and advice, and for the freedom he granted me;

Thomas Erlebach, my co-examiner, for fruitful research discussions, a nice month in Leicester, and the most thorough proof-checking;

Riko Jakob, my co-examiner, for many interesting ideas;

my other co-authors Christoph Ambühl, Michael Gatto, Alex Hall, Matúš Mihaľák, Daniel Sawitzki, Gábor Szabó, for doing research with me;

all (former) members of our research group Luzi Anderegg, Mark Cieliebak, Jörg Derungs, Stephan Eidenbenz, Michael Gatto, Beat Gfeller, Barbara Heller, Fabian Hennecke, Tomas Hruz, Zsuzsanna Lipták, Leon Peeters, Paolo Penna, Conrad Pomm, Franz Roos, Konrad Schlude, Christoph Stamm, Elias Vicari, Mirjam Wattenhofer, Birgitta Weber for a good time at ETH;

Tobias Achterberg, for support and the addition of features to his SCIP library;

Alberto Ceselli, for cooperating on the SBB project and a nice trip to Milano;

the research group of Rolf Möhring, in particular Marco Lübbecke and Heiko Schilling for hosting me in Berlin and cooperating with me;

Ted Ralphs, for introducing me to his SYMPHONY library;

Kathrin Schulte and the SBB Cargo Team for a pleasant cooperation;

and finally, my family and my friends in Zürich and the rest of the world for a nice time without computer science.

# Contents

# Chapter 1

# Introduction

**The cocktail party problem** Have you ever tried to explain to the "average guy on the street" what a theoretical computer scientist actually does? Then maybe this cocktail party conversation sounds familiar to you.

Alice: "So what exactly is it that you do as a theoretical computer scientist?"

Bob: "We explore the limits of computability, what you can compute and what you can't. At the same time we try to invent efficient solution methods for fundamental problems."

Alice: "What is a fundamental problem?"

Bob: "For example sorting a sequence of numbers, finding a good packing of objects into a bin, or finding a cheap tour through a network."

Alice: "Aha."

I have made the experience that the significance of abstract fundamental algorithmic problems is hard to grasp for most non-computer scientists, even if these problems occur in many places in various disguises. Also personally, I find it often more motivating to directly start with a practical problem, to try to solve it with the algorithmic tool kit and on the way formulate and hopefully solve the underlying "fundamental" problem. It is this approach that I try to adopt in this thesis.

We will encounter four such practical problems and algorithmic questions that are motivated by these problems. An ideal goal would be to develop for each given problem both interesting and new algorithmic theory and a practical solution that uses this new theory and completely solves the practical problem. However, it is not a secret that different problems lend themselves more or less well to practical or theoretical advances. There is often some trade-off between the desired high level of detail of a practical model versus the maximum level of detail for which we can prove theorems in a theoretical setting. For this reason, varying emphasis will be given to the practical and theoretical aspects of the different problems.

It is also true that different algorithmic problems necessitate different algorithmic techniques, be it because the theoretical core of the problem lends itself well to the application of a particular technique or because the application demands a certain type of result that can only be delivered by some techniques. For that reason, we will see in this thesis a tour d'horizon of state-of-the-art algorithm design and analysis techniques, which are applied to real-world problems: Approximation algorithms, online algorithms, linear programming based optimization and a few results on fixed-parameter tractability.

In the following we give a short overview of the four problems that are studied in this thesis.

**Sequential vector packing** This problem seems to be one of the few exceptions to the above rule about the trade-off between theory and practice. An industry partner described a setting to us that led to a clean combinatorial problem from the start. As a small letdown, we are not allowed to describe the underlying application but we give a similar application in Chapter 3 and give an alternative motivation here: Assume you want to spray a long text on a wall using stencils for the letters and spray color. You start from the left and assemble as much of the beginning of the text as you have matching stencils at your disposal. Then you mask the area around the stencils and start spraying. Afterwards, you remove the stencils again. In the next steps you iterate this procedure starting from the last letter that was sprayed until the whole text is finished. The sequential vector packing problem can be formulated as the following question: Assume you have bought enough material to cut out a fixed number $B$ of stencils before you start. How many stencils of each letter do you cut out in order to minimize the steps that you need to spray the whole text? We deal with this question in Chapter 3, where we formulate it as a novel vector packing problem. We give an NP-hardness proof and develop a (bicriteria) approximation algorithm that is based on LP-rounding and some structural insights, and propose and analyze some practical heuristics for the problem. Finally, we present experiments on real-world data that substantiate that our approaches significantly improve over the previous attempts to produce solutions to the problem.

**Optimizing a freight railway system** Railway systems pose a multitude of interesting optimization problems. We consider a freight train service that is operated as a (multi-) hub and spoke system. For that system we want to find good routes for the trains to go to and from the hubs and also a good schedule for these trains. The routes and the schedule are subject to various practical constraints that make the minimization of the overall cost of a solution a challenging task. We present a sequence of models with which we tackled the problem and some preliminary experimental results with real world data. We mainly use LP-based optimization techniques like branch-and-cut and column generation. The foundations of these techniques are presented in a separate chapter. The emphasis in this chapter is on the modeling aspect, which sets it off from the other chapters, where the model is usually both simple and well-defined from the start. For the implementation of our algorithms we use state-of-the-art libraries. In particular, we are the first to use the SCIP library [3] for a column generation approach. Apart from the more practical results, we also analyze some aspects of the underlying scheduling and shunting problems from a more theoretical point of view. It turns out that one of these “fundamental problems” in algorithmics, the min-cut linear arrangement problem, occurs at the core of the scheduling problem at the shunting yards.

**OVSF-code assignment** In telecommunications, there are several technologies that allow different users in one cell of a radio network to share the common bandwidth. In UMTS the so called Wideband Code Division Multiple Access (WCDMA) method is used, which assigns different orthogonal spreading (channelization) codes to the users in one cell. One type of such codes are Orthogonal Variable Spreading Factor codes (OVSF) that can be thought of as being derived from a binary code tree, where each node of the tree corresponds to a code. Users request different bandwidths which correspond to different levels in the code tree. In order for the orthogonality property to hold, there can be at most one assigned code on each path from the root of the tree to each of its leaves. In a dynamic setting where users arrive and depart, thus request and return such codes, the code tree can get fragmented. It can then happen that an additional request cannot be served because the currently active codes block all root-to-leaf paths on the requested level, even though there is enough bandwidth available. In such a situation, the active users get assigned new codes, i.e., the assigned codes in the tree are changed, such that the additional request can be served. This induces a communication overhead for each changed code. Therefore, it is an interesting question, how one can minimize the number of necessary reassignments. One algorithmic question here is how one can minimize the number of necessary code changes for a given assignment. Minn and Siu propose the

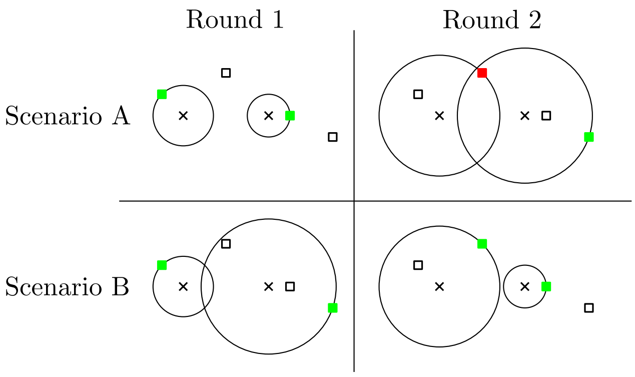


**Figure 1.0.1:** *Two scenarios for coordination of transmission between base stations. In Scenario A an interference problem arises in the second round (red box). In Scenario B the four users can be served in two rounds (currently served users depicted as green boxes).*

so-called DCA-algorithm to solve the problem optimally [93]. In contrast, we show that DCA does not always return an optimal solution, and that the problem is NP-hard. We also tackle the problem from other angles. We investigate approximation algorithms, and also consider the very natural online setting.

**Joint base station scheduling** A second technology to share the common bandwidth between users in one cell of a telecommunications network is Time Division Multiple Access (TDMA), in which time is slotted into synchronous rounds and each user is assigned a slot. We consider some theoretical problems that are motivated by an idea of Das, Viswanathan and Rittenhouse [43], who propose to coordinate the assignment of users to base stations to increase throughput and minimize interference. The coordination makes sense because of the power control mechanism of the base stations. Roughly speaking, the base stations adapt their signal-strength to the distance to the user that is served in the current time slot. The signal strength of a base station represents background noise for the other base stations and can lead to interference problems. For this reason, coordination can make sense: Consider the very simple setting in Figure 1.0.1, which also illustrates our basic model. Two base stations (crosses) are to serve four users (boxes) with unit demand. Base stations and users are modeled as points in the Euclidean plane. Our model of signal strength and interference are signal and interference disks. This means that in Scenario A the two base stations first serve the close by users without causing any interference. In the second time slot they try to serve the far away users, but this causes problems, because the left user is covered by two disks which leads to interference. In our model this means that the user cannot receive the signal. In Scenario B this does not happen. The base stations schedule the transmissions in such a way that no interference is caused. Observe that even if the interference disks intersect in the first step in B this causes no problem because no user that is currently served is contained in the interference region. Starting from this basic model, we investigate some algorithmic problems defined on such interference disks in one and two dimensions. We relate the one-dimensional problem to a coloring problem in a special graph class which we call *arrow graphs*, analyze some special cases, and also consider the complexity of the two-dimensional problem. We give simple approximation algorithms for both the one- and the two-dimensional problem.

**Outline of the thesis and research contributions** In this thesis the above four problems will be considered. In a final chapter the main results of the thesis are summarized. Some of the material covered has also been published in conference proceedings or in journals, see [51, 53, 63]. Chapter 3 is based on unpublished material and Chapter 5 extends the preliminary results presented in [63]. All of this work was done in collaboration with different co-authors. For this reason, the topics presented here will also partly be subject of other theses. For better readability, I will cover the complete results for each topic but refer to a thesis of a co-author for some of the proofs. In general, I will explicitly point out if the topic of a chapter is also covered in another thesis and explain my main contribution.

# Chapter 2

# Preliminaries

In this chapter we briefly review some basic definitions and notation that are used throughout this thesis. With the exception of LP-based optimization techniques, which are introduced separately in Chapter 4, we assume that the reader is familiar with the basic concepts from complexity theory, algorithm design and combinatorial optimization. There are numerous textbooks on these subjects, see for example [11, 80, 38, 35].

**Graphs** A graph $G = (V, E)$ is a pair of a *node set* $V$ and an *edge set* $E$. For *undirected* graphs each edge $e \in E$ is a two element subset of $V$, containing the *endpoints* of $e$. For *directed* graphs, each edge $e \in E$ is a pair $(u, v), u \in V, v \in V, u \neq v$, of *tail* and *head* of $e$, so that in particular $(u, v) \neq (v, u)$. A *walk* in a graph is an alternating list $v_0, e_1, v_1, \ldots, e_k, v_k$ of nodes and edges such that for $1 \leq i \leq k$ edge $e_i$ has endpoints $v_{i-1}$ and $v_i$. A *trail* is a walk with no repeated edge. A *path* is a walk with no repeated node. In the literature the definition of paths is a bit ambiguous. Other authors take the definition of walks or trails for paths and call the paths as they are defined here *simple* or also *elementary*. In places where we want to emphasize that a path is indeed elementary, we will also use this term. A $u, v$*-path* is a path starting at node $u$ and ending at $v$. For further standard graph-theoretic terminology, see the book by West [132].

**Basic complexity theory** There are many equivalent ways to define the basic concepts of complexity theory. Probably the most standard way is via nondeterministic Turing machines, see [62]. We give a very short introduction based on the certificate view on NP, following loosely [131].

An *alphabet* $\Sigma$ is a finite set of symbols, $\Sigma^*$ denotes the set of all finite strings (or *words*) of symbols in $\Sigma$. A *language* is a (possibly infinite) subset of $\Sigma^*$. An *algorithm* as we view it here maps every $w \in \Sigma^*$ to YES or NO, i.e., it *accepts* or *rejects* $w$. Deciding whether a given $w$ is contained in a language $L$ is called a *decision problem*.

The class P (of polynomially recognizable languages) is the set of all languages $L$ over $\{0, 1\}$, such that there is a polynomial time algorithm $A$ with $L = \{w \mid A \text{ accepts } w\}$. *Polynomial time* means that there is a polynomial $p_A$ such that $A$ terminates on any input in at most $p_A(|w|)$ steps.

A *verifier* for a language $L$ is an algorithm $V$ such that $L = \{w \mid \exists c \in \{0, 1\}^* : V \text{ accepts } (w, c)\}$. If $V$ accepts $(w, c)$ the string $c$ is called a *certificate* for membership of $w$ in $L$. The class NP is the class of all languages for which there is a verifier with a polynomial running time in the lengths of the words in $L$.

Defining languages on the alphabet $\{0, 1\}$ is not a real restriction. A typical instance of a combinatorial problem consists of various parameters and data items (like a graph), which can all be encoded as bitstrings on the alphabet $\{0, 1\}$.

A language $L_1$ is *polynomial time reducible* to language $L_2$, in symbols $L_1 \leq_P L_2$, if there is a polynomial time computable function $f : \{0, 1\}^* \to \{0, 1\}^*$ such that $w \in L_1$ if and only if $f(w) \in L_2$. In short, we also say $L_1$ *can be reduced to* $L_2$.

A decision problem $L$ is *NP-hard* if for all $L' \in$ NP the decision problem $L'$ can be reduced to $L$; if additionally $L \in$ NP, then $L$ is *NP-complete*.

A famous theorem by Cook states that the satisfiability problem for Boolean formulas is NP-complete [34]. The NP-hardness of a problem makes the existence of efficient, i.e., polynomial time algorithms for it highly unlikely: Under the largely believed hypothesis of $P \neq NP$ it holds that no polynomial time algorithm for an NP-hard problem can exist. Strictly speaking, we make the implicit assumption here that Turing machines and modern computers are equivalent with respect to polynomial time computations. This assumption is the famous *extended Church-Turing Thesis*. For a more thorough introduction to the theory of NP-completeness, see the book by Garey and Johnson [62].

**Approximation and online algorithms** In a typical application one is not so much interested in a decision problem but rather in an *optimization problem*, the goal of which is to minimize or maximize the value of an objective function that is defined on the feasible solutions of the problem. Also for optimization problems the notion of NP-hardness can be defined. It suffices to define the decision problem $\Pi_d$ associated with an optimization problem $\Pi$. Typically, one takes the question whether there are instances with an objective value better than a value $k$ (given in the input) as this decision problem. The NP-hardness of an optimization problem is then equivalent to the NP-hardness of the associated decision problem, see [11] for a more formal definition.

For such NP-hard optimization problems the existence of efficient, i.e., polynomial exact algorithms is again ruled out under the assumption of P $\neq$ NP. For this reason, researchers have developed *approximation algorithms* that do not necessarily find the optimum solution to an optimization problem but one for which the solution is in all cases provably only off by some factor $\rho$. Let $A$ be a polynomial time algorithm and denote by $A(I)$ the objective value returned by $A$ on instance $I$, further denote by $\mathrm{OPT}(I)$ the value of the optimum solution on $I$. We say that $A$ is a *$\rho$-approximation algorithm* if there exists a constant $k$ such that for all instances $I$ it holds that

$$A(I) \leq \rho \cdot \mathrm{OPT}(I) + k \text{ for a minimization problem}$$

and

$$\mathrm{OPT}(I) \leq \rho \cdot A(I) + k \text{ for a maximization problem.}$$

The value $\rho$ is called the *approximation ratio*[1] of the algorithm, it is always greater or equal to 1. Just as NP-hardness results rule out the existence of polynomial time exact algorithms, it is possible to show *inapproximability results* that rule out approximation algorithms with approximation ratios better than a given value or function.

In real-world applications it is often the case that the complete input is not available at the time when an algorithm is called. Data arrive over time in an input sequence and the algorithm might have to react online. An algorithm $A_{\text{on}}$ that operates in such a setting is called an *online algorithm*. An established method to evaluate the performance of $A_{\text{on}}$ is similar to the analysis of approximation algorithms. The objective function value of the solution that $A_{\text{on}}$ generates at the end of the input sequence $I$ is compared to the optimal objective function value $OPT(I)$. We say that $A_{\text{on}}$ is *$\rho$-competitive* if there exists a constant $k$ such that for each instance $I$

$$A_{\text{on}}(I) \leq \rho \cdot \mathrm{OPT}(I) + k \text{ for a minimization problem}$$

and

$$\mathrm{OPT}(I) \leq \rho \cdot A_{\text{on}}(I) + k \text{ for a maximization problem.}$$

[1]The above definition is again a bit ambiguous in the literature. Other authors take the definition for minimization problems also for maximization problems. It is also common to leave out the constant $k$ and to call the approximation ratio as defined above *asymptotic approximation ratio*.

The value $\rho$ is called the *competitive ratio*. It is common to think of the input sequence as a sequence being generated by a (malicious) *adversary*.

Sometimes it is hard to achieve satisfactory approximation ratios or competitive ratios. In this case it can make sense to compare the algorithm to an optimal algorithm that works on a more constrained input, or, equivalently to allow the algorithm to use more resources than the optimal algorithm. Historically, such algorithms are called *bicriteria approximation algorithms* for the approximation ratio and *resource augmented algorithms* in the online setting, which is also the terminology used in this thesis. In the literature the usage is not clear-cut, however.

**Definition 2.1 (Bicriteria Approximation, Resource Augmentation)** *Given a minimization problem with instances $I(r)$ that comprise as one input parameter $r$ the available amount of some resource. An algorithm $A$ is a* bicriteria $(\alpha, \beta)$-approximation algorithm *if a constant $k$ exists such that it finds for each instance $I(\beta \cdot r)$ a solution of value not more than $\alpha$ times the optimal objective value plus $k$ for $I(r)$, i.e.,*

$$A(I(\beta r)) \leq \alpha \cdot OPT(I(r)) + k \ . \tag{2.0.1}$$

The definition for maximization problems and online problems is analogous. In the online setting a bicriteria $(\alpha, \beta)$-competitive algorithm is typically called *$\alpha$-competitive with resource augmentation by a factor of $\beta$*.

Recently, a different relaxation to the definition of what one might understand by an “efficient” algorithm has attracted considerable attention. There are cases in which a (typically NP-complete) problem can be parameterized by a parameter $k$ such that for small $k$ there are efficient algorithms in the following sense.

**Definition 2.2 (Fixed Parameter Tractable)** *An algorithm $A$ is* fixed parameter tractable *(FPT) with respect to parameter $k$ if its running time is bounded by $f(k) \cdot n^{O(1)}$ for an arbitrary function $f$.*

In a typical fixed-parameter tractable algorithm, $f$ is something like $2^k$ or $2^{2^k}$. Intuitively, for such an algorithm the exponential behavior of the running time has been reduced to the parameter.

For a more thorough coverage of approximation algorithms, see for example the books by Ausiello et al. [11]. For more information on the techniques used to construct approximation algorithms, see the book by Vazirani [128] or the one edited by Hochbaum [73]. The books by Papadimitriou [100] or Wegener [130] give a broader overview over complexity theory. For textbooks on online algorithms, see the books by Borodin and El-Yaniv [18], Fiat and Woeginger [59] or the lecture notes by Albers [7]. Bicriteria approximation algorithms were originally introduced for scheduling problems [76, 101]. Later, the analogous resource augmentation was presented as an online analysis technique by Kalyanasundaram and Pruhs in [77]. See the books by Niedermeier [98] or Downey and Fellows [47] for a thorough introduction to the concept of fixed parameter tractability.

# Chapter 3

# Sequential Vector Packing

Tyrell: [explains to Roy why he can’t extend his lifespan]
“The facts of life...
to make an alteration in the evolvement of an organic life system is fatal.
A coding sequence cannot be revised once it’s been established.”
(from Blade Runner)

## 3.1 Introduction

Needless to say, many variations of bin packing have attracted a huge amount of scientific interest over the past decades, partly due to their relevance in diverse scheduling applications. The variation which we investigate in this chapter arises from a specific *resource constrained scheduling* problem: A sequence of jobs is given and must be processed in this order. Each job needs certain amounts of various types of resources (for instance 1 unit of resource A, 2 units of resource B, 0 units of all others). Several jobs can be processed in one batch if the resources they consume altogether are bounded by a predefined bound. Specifically, for each individual type of resource we have a reservoir containing some amount of the resource and none of these amounts may be exceeded during one batch. Within a given bound on the total amount of available resources one has the freedom to choose how these individual amounts are distributed once and for all (e.g., 10 units in the reservoir of resource A, 23 units in the reservoir of resource B, . . . ). The aim is to tune these amounts in such a way that the jobs are processed in as few batches as possible. To motivate this scenario we now describe a specific scheduling problem which is a variation of a setting presented by Müller-Hannemann and Weihe [95]. The setting in [95] arose as a subproblem in a cooperation with Philips/Assembléon B.V., Eindhoven, the Netherlands.

The task is to optimize an assembly line that consists of a conveyor belt on which different work pieces arrive in the work area, and of a set of robot arms that can process these work pieces. The robot arms can perform different tasks depending on the resources (for example tools) they load in a setup phase before each step from a resource reservoir (a toolbox). There are $d$ different such resources. The work pieces require different processing, i.e., robot arms equipped with a specific amount of each resource. The sequence $\mathbf{S}$ in which the work pieces arrive on the assembly line is fixed and cannot be altered, it can be thought of as a stack. This is the crucial difference compared to standard bin packing or scheduling scenarios, where it is assumed that reordering is possible.

The resource reservoir has a total size of $B$ and contains an amount $b_j$ of each resource $j$, $j \in \{1, \dots, d\}$, such that $\sum_{j=1}^{d} b_j = B$. Each production cycle consists of a setup phase in which the robots load the necessary resources, so that in a second phase as many work pieces as possible can be moved into the work area (i.e., "popped from the stack") and processed. For a set of work pieces to be processed all the necessary resources must have been loaded in the setup phase. The optimization task is to choose the values $b_j, j \in \{1, \dots, d\}$ once and for all such that the total number of cycles needed to process the whole sequence is minimized. This is not an on-line problem, since the sequence of work pieces arriving on the conveyor belt is known in the beginning.

### 3.1.1 Model and Notation

We now give a formal definition of the problem. The jobs or work pieces correspond to a sequence of vectors, i.e., the demand for resource $i$ is specified in the $i$th component of such a vector. Similarly, the available resources correspond to a bin vector, respectively. To the best of our knowledge this setting is novel. Due to its basic character we believe that it may be of interest also in contexts other than resource constrained scheduling. In some sense the problem can be seen as inverse vector packing: Instead of reordering a sequence for a given bin vector, the sequence is fixed and the bin vector needs to be chosen.

**Definition 3.1 (Sequential vector packing)**
**Given:** *a sequence* $\mathbf{S} = \mathbf{s}_1 \cdots \mathbf{s}_n$ *of* demand vectors $\mathbf{s}_i = (s_{i1}, \dots, s_{id}) \in \mathbb{Q}_+^d$, $d \in \mathbb{N}$, *and a* total bin size $B \in \mathbb{Q}_+$.

**Goal:** *a* bin vector *(or short:* bin*)* $\mathbf{b} = (b_1, \dots, b_d) \in \mathbb{Q}_+^d$ *with* $\sum_{j=1}^{d} b_j = B$ *such that* $\mathbf{s}_1 \cdots \mathbf{s}_n$ *can be packed in this order into a minimum number of such bins* $\mathbf{b}$. *More precisely, the sequence can be packed into* $k$ *bins, if* breakpoints

*$0 = \pi_0 < \pi_1 < \cdots < \pi_k = n$ exist, such that*

$$\sum_{i=\pi_l+1}^{\pi_{l+1}} \mathbf{s}_i \le \mathbf{b} \qquad \text{for } l \in \{0, \ldots, k-1\} \ ,$$

*where inequalities over vectors are component-wise. We denote the $j$-th component, $j \in \{1, \ldots, d\}$, of the demand vectors and the bin vector as* resource *$j$, i.e., $s_{ij}$ is the demand for resource $j$ of the $i$-th demand vector. We also refer to $\mathbf{s}_i$ as* position *$i$.*

The *sequential unit vector packing* problem is the restricted variant where $\mathbf{s}_i$, $i \in \{1, \ldots, n\}$, contains exactly one entry equal to 1, all others are zero, i.e., each work piece needs only one tool. Note that any solution for this version can be transformed in such a way that the bin vector is integral, i.e., $\mathbf{b} \in \mathbb{N}^d$, by potentially rounding down resource amounts to the closest integer (therefore one may also restrict the total bin size to $B \in \mathbb{N}$). The same holds if all vectors in the sequence are integral, i.e., $\mathbf{s}_i \in \mathbb{N}^d$, $i \in \{1, \ldots, n\}$. Following Definition 2.1 we call an algorithm $A$ a *bicriteria $(\alpha, \beta)$-approximation algorithm* for the sequential vector packing problem if it finds for each instance $(\mathbf{S}, \beta \cdot B)$ a solution which needs no more than $\alpha$ times the number of bins of an optimal solution for $(\mathbf{S}, B)$. That is, the approximation algorithm may not only approximate the value of the objective function within a factor of $\alpha$, but it may also relax the total bin size by a factor of $\beta$.

### 3.1.2 Related Work

There is an enormous wealth of publications both on the classical bin packing problem and on variants of it. The two surveys by Coffman, Garey and Johnson [32, 33] give many pointers to the relevant literature until 1997. In [36] Coppersmith and Raghavan introduce the multidimensional (on-line) bin packing problem. There are also some variants that take into consideration precedence relations on the items [134, 129] that remotely resemble our setting. Still, we are unaware of any publication that deals with the sequential vector packing problem.

The setting presented in [95] is similar to ours. Different work pieces arrive in an unalterable sequence on an assembly line and are processed by several robot arms. The principal goal is the same: to process all work pieces as quickly as possible. Moving the work pieces forward on the assembly line costs a fixed amount of time; this corresponds to the time needed for the setup phase. In contrast to our setting the processing times for each step can vary though, depending on which work piece can be accessed by which robot arm. In effect, each work piece has a certain number of tasks that need to be done by specific robot arms. Another difference to our setting is that the robot arms do not have limited resources. Müller-Hannemann and Weihe [95] give an NP-hardness proof for their setting, present several results pertaining to fixed parameter tractability, and derive a polynomial time approximation scheme. For a survey of general problems and approaches to the balancing and sequencing of assembly lines see Scholl [113]. Ayob et al. [12] compare different models and assembly machine technologies for surface mount placement machines.

### 3.1.3 Summary of Results

In Section 3.2 we present approximation algorithms for the sequential vector packing problem. These are motivated by the strong NP-hardness results that we give in Section 3.3. The approximation algorithms are based on an LP relaxation and two different rounding schemes, yielding a bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation and—as the main result of this chapter—a $(1, 2)$-approximation. Recall that the former algorithm, e.g., for $\varepsilon = \frac{1}{3}$, yields solutions with at most $3$ times the optimal number of bins while using at most $1.5$ times the given total bin size $B$, the latter may use at most the optimal number of bins and at most twice the given total bin size $B$. In Section 3.4.1 we present two simple greedy strategies and argue why they perform badly in the worst case. Furthermore, we give an easy to implement heuristic and present two optimizations concerning subroutines. In particular, we show how one can “evaluate” a given bin vector—i.e., compute the number $k$ of bins needed with this bin vector—in time $O(k \cdot d)$ after a preprocessing phase which takes $O(n)$ time. Finally, in Section 3.5 we briefly discuss the results of experiments with the heuristics and an ILP formulation on real world data.

## 3.2 Approximation Algorithms

In this section we present approximation algorithms for the sequential vector packing problem. These are motivated by the strong NP-hardness results that we give in Section 3.3.1. We start by presenting an ILP-formulation, which we subsequently relax to an LP. For ease of exposition we continue by first describing a simple rounding scheme which yields a bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation and then show how to modify it in order to obtain a $(1, 2)$-approximation. For $\varepsilon < 1/2$ the first rounding scheme yields solutions that violate the given bin size by less than a factor of two.

### 3.2.1 ILP Formulation

For a sequential vector packing instance $(\mathbf{S}, B)$, let $\mathbf{w}_{u,v} := \sum_{i=u+1}^{v} \mathbf{s}_i$, for $u, v \in \{0, \ldots, n\}$ and $u < v$, denote the *total demand* (or *total demand vector*) of the subsequence $\mathbf{S}_{u,v} := \mathbf{s}_{u+1} \cdots \mathbf{s}_v$. If $\mathbf{w}_{u,v} \leq \mathbf{b}$ holds, we can pack the subsequence $\mathbf{S}_{u,v}$ into bin $\mathbf{b}$. The following integer linear programming (ILP) formulation solves the sequential vector packing problem. Let $X := \{x_i \mid i \in \{0, \ldots, n\}\}$ and $Y := \{y_{u,v} \mid u, v \in \{0, \ldots, n\}, u < v\}$ be two sets of 0-1 variables, let $\mathbf{b} \in \mathbb{Q}_+^d$.

$$
\begin{array}{lllll}
\textbf{ILP:} & \text{minimize} & \displaystyle\sum_{i=1}^{n} x_i & & \\
& \text{s.t.} & x_0 = 1 & & (3.2.1) \\
& & \displaystyle\sum_{u=0}^{i-1} y_{u,i} = \sum_{v=i+1}^{n} y_{i,v} = x_i & \text{for } i \in \{1, \ldots, n-1\} & (3.2.2) \\
& & \displaystyle\sum_{\substack{u,v:\\ u<i\leq v}} \mathbf{w}_{u,v} \cdot y_{u,v} \leq \mathbf{b} & \text{for } i \in \{1, \ldots, n\} & (3.2.3) \\
& & \displaystyle\sum_{j=1}^{d} b_j = B & & (3.2.4) \\
& & \mathbf{b} \in \mathbb{Q}_+^d, x_i, y_{u,v} \in \{0, 1\} & \text{for } x_i \in X, y_{u,v} \in Y &
\end{array}
$$

The 0-1 variable $x_i$ indicates whether there is a breakpoint at position $i \geq 1$. Hence the objective: to minimize the sum over all $x_i$. The 0-1 variable $y_{u,v}$ can be seen as a flow which is routed on an (imagined) edge from position $u \in \{0, \ldots, n-1\}$ to position $v \in \{1, \ldots, n\}$, with $u < v$, see Figure 3.2.1. The Constraints (3.2.2) ensure that flow conservation holds for the flow represented by the $y_{u,v}$ variables and that $x_i$ is equal to the inflow (outflow) which enters (leaves) position $i$. Constraint (3.2.1) enforces that only one unit of flow is sent via the $Y$ variables. The path which is taken by this unit of flow directly corresponds to a series of breakpoints. For instance, if we have consecutive breakpoints at positions $u$ and $v$, there will be a flow of 1 from $u$ to $v$, i.e., $x_u = y_{u,v} = x_v = 1$.

In Constraints (3.2.3) the bin vector $\mathbf{b}$ comes into play: for any two consecutive breakpoints (e.g., $x_u = x_v = 1$) the constraint ensures that the bin vector is large enough for the total demand between the breakpoints (e.g., the total demand $\mathbf{w}_{u,v}$ of the subsequence $\mathbf{S}_{u,v}$). Note that Constraints (3.2.3) sum over all edges that span over a position $i$ (in a sense the cut defined by position $i$), enforcing that

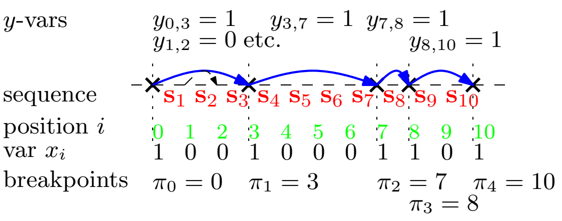


**Figure 3.2.1:** *An exemplary sequential vector packing instance together with a potential ILP solution, and a flow representation of the solution. The objective value is 4.*

the total resource usage is bounded by $\mathbf{b}$. For the two consecutive breakpoints $x_u$ and $x_v$ this amounts to $\mathbf{w}_{u,v} \cdot y_{u,v} \leq \mathbf{b}$. Finally, Constraint (3.2.4) ensures the correct total size of the bin vector.

### 3.2.2 An easy $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-Approximation

As a first step we relax the ILP formulation to an LP: here this means to have $x_i, y_{u,v} \in [0,1]$ for $x_i \in X, y_{u,v} \in Y$. We claim that the following Algorithm Eps Rounding computes a $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$-approximation:

1. Solve the LP optimally. Let $(X^\star, Y^\star, \mathbf{b}^\star)$ be the obtained fractional solution.
2. Set $(\hat{X}, \hat{Y}, \hat{\mathbf{b}}) = (X^\star, Y^\star, \frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star)$ and $i_s = 0$. Stepwise round $(\hat{X}, \hat{Y})$:
3. Let $i_e > i_s$ be the first position for which $\sum_{i=i_s+1}^{i_e} \hat{x}_i \geq \varepsilon$.
4. Set $\hat{x}_i = 0$ for $i \in \{i_s + 1, \ldots, i_e - 1\}$, set $\hat{x}_{i_e} = 1$. Reroute the flow accordingly (see also Figure 3.2.2):
   (a) Set $\hat{y}_{i_s, i_e} = 1$.
   (b) Increase $\hat{y}_{i_e, i}$ by $\sum_{i'=i_s}^{i_e - 1} \hat{y}_{i', i}$, for $i > i_e$.
   (c) Set $\hat{y}_{i_s, i'} = 0$ and $\hat{y}_{i', i} = 0$, for $i' \in \{i_s + 1, \ldots, i_e - 1\}, i > i'$.
5. Set the new $i_s$ to $i_e$ and continue in Line 3, until $i_s = n$.

**Theorem 3.2** *The algorithm* Eps Rounding *is a* $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$*-approximation algorithm for the sequential vector packing problem.*

**Proof.** We show the desired result in three separate steps.

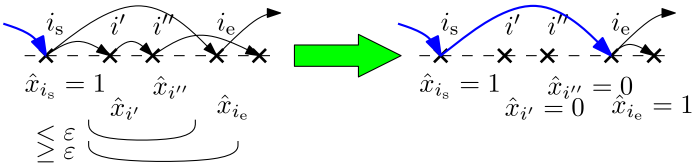


**Figure 3.2.2:** *An example of the rerouting of flow in Lines 4 (a)-(c) of the algorithm.*

**Integer Rounding.** The following invariant is easy to see by considering Figure 3.2.2: at the beginning of each iteration step (i.e., at Line 3) the current, partially rounded solution $(\hat{X}, \hat{Y}, \hat{\mathbf{b}})$ corresponds to a valid flow, which is integral until position $i_s$. From this invariant it follows that $(\hat{X}, \hat{Y}, \hat{\mathbf{b}})$ in the end corresponds to a valid integer flow.

**At Most $\frac{1}{\varepsilon}$-times the Number of Breakpoints.** In Line 4, $\hat{x}_i$ values which sum up to at least $\varepsilon$ (see Line 3) are replaced by $\hat{x}_{i_e} = 1$. Therefore, the rounding increases the total value of the objective function by at most a factor of $\frac{1}{\varepsilon}$.

**At Most $\frac{1}{1-\varepsilon}$-times the Total Bin Size.** Again consider one step of the iteration. We need to check that by rerouting the flow to go directly from $i_s$ to $i_e$ we do not exceed the LP bin capacity by more than $\frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star$. We show the stronger invariant that in each step after rerouting, the current, partially rounded solution fulfills Constraint (3.2.3) until $i_e$ w.r.t. $\hat{\mathbf{b}}$ and from $i_e+1$ to the end of the sequence w.r.t. $\mathbf{b}^\star$. First let us consider the increase of $\hat{y}_{i_e,i}$, for $i > i_e$, in Line 4 (b). Since the total increase is given directly by some $\hat{y}_{i',i}$ which are set to $0$ (Line 4 (c)), the Constraint (3.2.3) still holds after the change w.r.t. $\mathbf{b}^\star$. In other words, flow on edges is rerouted here onto shorter (completely contained) edges; this does not change the feasibility of Constraint (3.2.3).

Now we consider the increase of $\hat{y}_{i_s,i_e}$ to $1$. We will show that the total demand $w_{i_s,i_e}$ between the new breakpoints $i_s$ and $i_e$ is bounded by $\frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star$. With $\hat{x}_{i_s} = 1$ and since $\sum_{i=i_s+1}^{i_e-1} \hat{x}_i < \varepsilon$ (see Line 3), we know that $\sum_{i=i_e}^{n} \hat{y}_{i_s,i} = \hat{x}_{i_s} - \sum_{i=i_s+1}^{i_e-1} \hat{y}_{i_s,i} \geq 1 - \sum_{i=i_s+1}^{i_e-1} \hat{x}_i > 1 - \varepsilon$; note that the first equality holds by the flow conservation constraint (3.2.2). By Constraint (3.2.3) we obtain $w_{i_s,i_e} \cdot \sum_{i=i_e}^{n} \hat{y}_{i_s,i} \leq \sum_{i=i_e}^{n} w_{i_s,i} \cdot \hat{y}_{i_s,i} \leq \sum_{u,v:u<i_e\leq v} w_{u,v} \cdot \hat{y}_{u,v} \leq \mathbf{b}^\star$, where the last inequality follows by the invariant for the last step. Thus, plugging these two inequalities together, we know for the total demand $w_{i_s,i_e} < \frac{1}{1-\varepsilon} \cdot \mathbf{b}^\star = \hat{\mathbf{b}}$. Since this holds for all iteration steps and thus for all consecutive breakpoints of the final solution, it is clear that multiplying the bin vector of the LP solution by a factor $\frac{1}{1-\varepsilon}$ yields a valid solution for the ILP. □

Note that one would not actually implement the algorithm EPS ROUNDING. Instead, it suffices to compute the bin vector $\mathbf{b}^\star$ with the LP and then multiply it by $\frac{1}{1-\varepsilon}$ and evaluate the obtained bin vector, e.g., with the algorithm given in Section 3.4.4.

### 3.2.3 A $(1, 2)$-Approximation

We start by proving some properties of the LP relaxation and then describe how they can be applied to obtain the rounding scheme yielding the desired bicriteria ratio.

#### Properties of the Relaxation

Let $(X, Y, \mathbf{b})$ be a fractional LP solution; recall that the $Y$ variables represent a flow. Let $e_1 = (u, v)$ and $e_2 = (u', v')$ denote two flow carrying edges, i.e., $y_{u,v} > 0$ and $y_{u',v'} > 0$. We say that $e_1$ *is contained in* $e_2$ if $u' < u$ and $v' > v$, we also call $(e_1, e_2)$ an *embracing pair*. We say an embracing pair $(e_1, e_2)$ is *smaller* than an embracing pair $(\hat{e}_1, \hat{e}_2)$, if the length of $e_1$ (for $e_1 = (u, v)$, its length is $v - u$) is less than the length of $\hat{e}_1$ and in case of equal lengths, if $u < \hat{u}$ (order by left endpoint). That is, for two embracing pairs with distinct $e_1$ and $\hat{e}_1$ we always have that one is smaller than the other. We show that the following structural property holds:

**Lemma 3.3 (no embracing pairs)** *Any optimal fractional LP solution $(X^\star, Y^\star, \mathbf{b}^\star)$ can be modified in such a way that it contains no embracing pairs without increasing the objective function and without modifying the bin vector.*

**Proof.** We set $Y = Y^\star$ and show how to stepwise treat embracing pairs contained in $Y$, proving after each step that $(X^\star, Y, \mathbf{b}^\star)$ is still a feasible LP solution. We furthermore show that this procedure terminates and in the end no embracing pairs are left in $Y$.

Let us begin by describing one iteration step, assuming $(X^\star, Y, \mathbf{b}^\star)$ to be a feasible LP solution which still contains embracing pairs. Let $(e_1, e_2)$, with $e_1 = (u, v)$ and $e_2 = (u', v')$, be an embracing pair. We now modify the flow $Y$ to obtain a new flow $Y'$ by rerouting $\lambda = \min\{y_{u,v}, y_{u',v'}\}$ units of flow from $e_1, e_2$ onto the edges $e'_1 = (u, v')$ and $e'_2 = (u', v)$: $y'_{u,v} = y_{u,v} - \lambda$, $y'_{u',v'} = y_{u',v'} - \lambda$

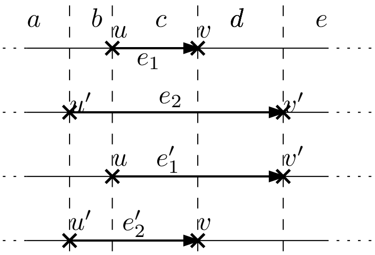


**Figure 3.2.3:** *Replacement of $\lambda$ units of flow on $e_1$ and $e_2$ by $\lambda$ units of flow on $e'_1$ and $e'_2$ in Lemma 3.3.*

and $y'_{u',v} = y_{u',v} + \lambda$, $y'_{u,v'} = y_{u,v'} + \lambda$; see also Figure 3.2.3. The remaining flow values in $Y'$ are taken directly from $Y$. It is easy to see that the flow conservation constraints (3.2.2) still hold for the values $X^\star, Y'$ (consider a circular flow of $\lambda$ units sent in the residual network of $Y$ on the cycle $u', v, u, v', u'$). Since $X^\star$ is unchanged this also implies that the objective function value did not change, as desired. It remains to prove that the Constraints (3.2.3) still hold for the values $Y'$, $\mathbf{b}^\star$ and to detail how to consecutively choose embracing pairs $(e_1, e_2)$ in such a way that the iteration terminates.

**Feasibility of the Modified Solution.** Constraints (3.2.3) are parameterized over $i \in \{1, \ldots, n\}$. We argue that they are not violated separately for $i \in \{u' + 1, \ldots, u\}$, $i \in \{u + 1, \ldots, v\}$, and $i \in \{v + 1, \ldots, v'\}$, i.e., the regions $b$, $c$, and $d$ in Figure 3.2.3. For the remaining regions $a$ and $e$ it is easy to check that the values of the affected variables do not change when replacing $Y$ by $Y'$. So let us consider the three regions:

**Region $b$ ($d$)** The only variables in (3.2.3) which change when replacing $Y$ by $Y'$ for this region are: $y'_{u',v'} = y_{u',v'} - \lambda$ and $y'_{u',v} = y_{u',v} + \lambda$. This means that flow is moved to a shorter edge, which can only increase the slack of the constraints: With $\mathbf{w}_{u',v} < \mathbf{w}_{u',v'}$ it is easy to see that (3.2.3) still holds in region $b$. Region $d$ is analogous to $b$.

**Region $c$** Here the only variables which change in (3.2.3) are: $y'_{u,v} = y_{u,v} - \lambda$, $y'_{u',v'} = y_{u',v'} - \lambda$, $y'_{u',v} = y_{u',v} + \lambda$, and $y'_{u,v'} = y_{u,v'} + \lambda$. In other words $\lambda$ units of flow were moved from $e_1$ to $e'_1$ and from $e_2$ to $e'_2$. Let us consider the fraction of demand which is contributed to (3.2.3) by these units of flow before and after the modification. Before (on $e_1$ and $e_2$) this was $\lambda \cdot (\mathbf{w}_{u,v} + \mathbf{w}_{u',v'})$ and afterwards (on $e'_1$ and $e'_2$) it is $\lambda \cdot (\mathbf{w}_{u',v} + \mathbf{w}_{u,v'})$. Since both quantities are equal, the left hand side of (3.2.3) remains unchanged in region $c$.

**Choice of $(e_1, e_2)$ and Termination of the Iteration.** In each step of the iteration we always choose the smallest embracing pair $(e_1, e_2)$, as defined above. If there are several smallest embracing pairs (which by definition all contain the same edge $e_1$), we choose one of these arbitrarily.

First we show that the modification does not introduce an embracing pair that is smaller than $(e_1, e_2)$. We assume the contrary and say w.l.o.g. that the flow added to edge $e'_1$ creates a new embracing pair $(e, e'_1)$ that is smaller than the (removed) embracing pair $(e_1, e_2)$. Clearly, $e$ is also contained in $e_2$. Therefore, before the modification $(e, e_2)$ would have been an embracing pair as well. Since $(e, e_2)$ is smaller than $(e_1, e_2)$ it would have been chosen instead, which gives the contradiction.

It follows that we can divide the iterations into a bounded number of phases: in each phase all considered embracing pairs are with respect to the same $e_1$-type edge. As soon as a phase is finished (i.e., no embracing pairs with the phase's $e_1$-type edge remain) this $e_1$-type edge will never be considered again, since this could only happen by introducing a smaller embracing pair later in the iteration. Thus, there are at most $O(n^2)$ phases.

Now we consider a single phase during which an edge $e_1$ is contained in possibly several other edges $e_2$. By the construction of the modification for an embracing pair $(e_1, e_2)$ it is clear that $e_2$ could not be chosen twice in the same phase. Therefore, the number of modification steps per phase can also be bounded by $O(n^2)$. $\square$

## Choose a Flow Carrying Path

We will use the structural insights of the last section to prove that bin vector $2 \cdot \mathbf{b}^\star$ yields a $(1, 2)$-approximation to the optimal solution.

Due to Lemma 3.3 an optimal fractional LP solution $(X^\star, Y^\star, \mathbf{b}^\star)$ without embracing pairs exists. Let $p_{\min}$ denote the shortest flow carrying path in $(X^\star, Y^\star, \mathbf{b}^\star)$, where shortest is meant with respect to the number of breakpoints. Clearly, the length of $p_{\min}$ is at most the objective function value $\sum_{i=1}^{n} x_i^\star$, since the latter can be seen as a linear combination of the path lengths of an arbitrary path decomposition. Below we show that the integral solution corresponding

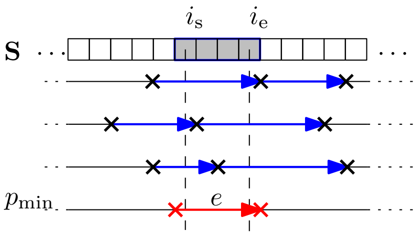


**Figure 3.2.4:** *Extracting the integral solution. Edge $e$ together with other potential edges in $Y^\star$ in Theorem 3.4.*

to $p_\mathrm{min}$ is feasible for the bin vector $2 \cdot \mathbf{b}^\star$, and thus $p_\mathrm{min}$ and $2 \cdot \mathbf{b}^\star$ are our $(1, 2)$-approximation. Observe that the approximation algorithm does not actually need to transform an optimal LP solution, given, e.g., by an LP solver, into a solution without embracing pairs. The existence of path $p_\mathrm{min}$ in such a transformed solution is merely taken as a proof that the bin vector $2 \cdot \mathbf{b}^\star$ yields less than $\sum_{i=1}^{n} x_i^\star$ breakpoints. To obtain such a path, we simply evaluate $2 \cdot \mathbf{b}^\star$ with the algorithm presented in Section 3.4.4 ($\mathbf{b}^\star$ given by the LP solver).

**Theorem 3.4** *Given an optimal fractional LP solution $(X^\star, Y^\star, \mathbf{b}^\star)$ without embracing pairs, let $p_\mathrm{min}$ denote the shortest flow carrying path. The integral solution corresponding to $p_\mathrm{min}$ is feasible for $2 \cdot \mathbf{b}^\star$.*

**Proof.** We only have to argue for the feasibility of the solution w.r.t. the doubled bin vector. Again we will consider Constraints (3.2.3). Figure 3.2.4 depicts an edge $e$ on path $p_\mathrm{min}$ and other flow carrying edges. We consider the start and end position $i_\mathrm{s}$ and $i_\mathrm{e}$ in the subsequence defined by $e$. Denote by $E_{i_\mathrm{s}} = \{(u, v) \mid 0 \leq u < i_\mathrm{s} \leq v \leq n\}$ (and $E_{i_\mathrm{e}}$, respectively) the set of all flow carrying edges that cross $i_\mathrm{s}$ $(i_\mathrm{e})$ and by $i_\mathrm{min}, (i_\mathrm{max})$ the earliest tail (latest head) of an arc in $E_{i_\mathrm{s}}, (E_{i_\mathrm{e}})$. Furthermore, let $E' = E_{i_\mathrm{s}} \cup E_{i_\mathrm{e}}$. Summing up the two Constraints (3.2.3) for $i_\mathrm{s}$ and $i_\mathrm{e}$ gives $2\mathbf{b}^\star \geq \sum_{(u,v)\in E_{i_\mathrm{s}}} y^\star_{u,v} \cdot \mathbf{w}_{u,v} + \sum_{(u,v)\in E_{i_\mathrm{e}}} y^\star_{u,v} \cdot \mathbf{w}_{u,v} =: A$ and thus

$$2\mathbf{b}^\star \geq A \quad \geq \quad \sum_{i_\mathrm{min} < i \leq i_\mathrm{max}} \; \sum_{\substack{(u,v)\in E': \\ u<i\leq v}} y^\star_{u,v} \cdot \mathbf{s}_i \tag{3.2.5}$$

$$\geq \quad \sum_{i_\mathrm{s} < i \leq i_\mathrm{e}} \; \sum_{\substack{(u,v)\in E': \\ u<i\leq v}} y^\star_{u,v} \cdot \mathbf{s}_i = \sum_{i_\mathrm{s} < i \leq i_\mathrm{e}} \mathbf{s}_i = \mathbf{w}_{i_\mathrm{s}, i_\mathrm{e}} \quad . \tag{3.2.6}$$

The second inequality in (3.2.5) is in general an inequality because the sets $E_{i_\mathrm{s}}$ and $E_{i_\mathrm{e}}$ need not be disjoint. For the first equality in (3.2.6) we rely on the fact that there are no embracing pairs. For this reason, each position between $i_\mathrm{s}$ and $i_\mathrm{e}$ is covered by an edge that covers either $i_\mathrm{s}$ or $i_\mathrm{e}$. We have shown that the demand between any two breakpoints on $p_{\min}$ can be satisfied by the bin vector $2 \cdot \mathbf{b}^\star$. $\square$

Observe that for integral resources the above proof implies that even $\lfloor 2\mathbf{b}^\star \rfloor$ has no more breakpoints than the optimal solution. Note also that it is easy to adapt both approximation algorithms so that they can handle pre-specified breakpoints. The corresponding $x_i$ values can simply be set to one in the ILP and LP formulations.

## 3.3 Complexity Considerations

In this section, we study the computational complexity of the sequential vector packing problem. First, we show that finding an optimal solution is NP-hard, and then we consider special cases of the problem that allow a polynomial time algorithm or that are fixed parameter tractable (FPT). Our NP-hardness proofs also identify parameters that cannot lead to an FPT-algorithm.

### 3.3.1 Minimizing the Number of Breakpoints (Bins)

For all considered problem variants it is easy to determine the objective value once a bin vector is chosen. Hence, for all variants of the sequential vector packing problem considered in this chapter, the corresponding decision problem is in NP.

To simplify the exposition we first consider a variant of the sequential unit vector packing problem where the sequence of vectors has prespecified breakpoints, always after $w$ positions. Then the sequence effectively decomposes into a set of windows of length $w$, and for each position in such a window $i$ it is sufficient to specify the resource that is used at position $j \in \{1, \ldots, w\}$, denoted as $s^i_j \in \{1, \ldots, d\}$. This situation can be understood as a set of sequential unit vector packing problems that have to be solved with the same bin vector. The objective is to minimize the total number of (additional) breakpoints, i.e., the sum of the objective functions of the individual problems. Later, we also show strong NP-hardness for the original problem.

**Lemma 3.5** *Finding the optimal solution for sequential unit vector packing with windows of length 4 (dimension $d$ and bin size $B$ as part of the input) is NP-hard.*

**Proof.** By reduction from the NP-complete problem Clique [62] or more generally from $k$-densest subgraph [56]. Let $G = (V, E)$ be an instance of $k$-densest subgraph, i.e., an undirected graph without isolated nodes in which we search for a subset of nodes of cardinality $k$ that induces a subgraph with the maximal number of edges.

We construct a sequential unit vector packing instance $(\mathbf{S}, B)$ with windows of length 4 and with $d = |V|$ resources. Assume as a naming convention $V = \{1, .., d\}$. There is precisely one window per edge $e = (u, v) \in E$, the sequence of this window is $s^e = uvuv$. The total bin size is set to $B = d + k$. This transformation can be carried out in polynomial time and achieves, as shown in the following, that $(\mathbf{S}, B)$ can be solved with at most $|E| - \ell$ (additional) breakpoints if and only if $G$ has a subgraph with $k$ nodes containing at least $\ell$ edges.

Because every window contains at most two vectors of the same resource, having more than two units of one resource does not influence the number of breakpoints. Every resource has to be assigned at least one unit because there are no isolated nodes in $G$. Hence, a solution to $(\mathbf{S}, B)$ is characterized by the subset $R$ of resources to which two units are assigned (instead of one). By the choice of the total bin size we have $|R| = k$. A window does not induce a breakpoint if and only if both its resources are in $R$, otherwise it induces one breakpoint.

If $G$ has a node induced subgraph $G'$ of size $k$ containing $\ell$ edges, we chose $R$ to contain the nodes of $G'$. Then, every window corresponding to an edge of $G'$ has no breakpoint, whereas all other windows have one. Hence, the number of (additional) breakpoints is $|E| - \ell$.

If $(\mathbf{S}, B)$ can be scheduled with at most $|E| - \ell$ breakpoints, define $R$ as the resources for which there is more than one unit in the bin vector. Now $|R| \leq k$, and we can assume $|R| = k$ since the number of breakpoints only decreases if we change some resource from one to two, or decrease the number of resources to two. The set $R$ defines a subgraph $G'$ with $k$ nodes of $G$. The number of edges is at least $\ell$ because only windows with both resources in $R$ do not have a breakpoint. $\square$

It remains to consider the original problem without pre-specified breakpoints.

**Lemma 3.6** *Let $(\mathbf{S}, B)$ be an instance of sequential (unit) vector packing of length $n$ with $k$ pre-specified breakpoints and $d$ resources ($d \leq B$) where every resource is used at least once.*

*Then one can construct in polynomial time an instance $(\mathbf{S}', B')$ of the (unit) vector packing problem with bin size $B' = 3B+2$ and $d' = d+2B+2$ resources that can be solved with at most $\ell + k$ breakpoints if and only if $(\mathbf{S}, B)$ can be solved with at most $\ell$ breakpoints.*

**Proof.** The general idea is to use for every prespecified breakpoint some "stopping" sequence $F_i$ with the additional resources in a way that the bound $B'$ guarantees that there is precisely one breakpoint in $F_i$. This sequence $F_i$ needs to enforce exactly one breakpoint, no matter whether or not there was a breakpoint within the previous window (i.e., between $F_{i-1}$ and $F_i$). If we used same sequence for $F_{i-1}$ and $F_i$, a breakpoint within the window would yield a "fresh" bin vector for $F_i$. Therefore, the number of breakpoints in $F_i$ could vary depending on the demands in the window (and whether or not they incur a breakpoint).

To avoid this, we introduce two different stopping sequences $F$ and $G$ which we use alternatingly. This way we are sure that between two occurrences of $F$ there is at least one breakpoint. The resources $1, \ldots, d$ of $(\mathbf{S}', B')$ are one-to-one to the resources of $(\mathbf{S}, B)$. The $2B + 2$ additional resources are divided into two groups $f_1, \ldots, f_{B+1}$ for $F$ and $g_1, \ldots, g_{B+1}$ for $G$. The first pre-specified breakpoint in $\mathbf{S}$, the third and every other odd breakpoint is replaced by the sequence $F := f_1 f_2 \cdots f_{B+1} f_1 f_2 \cdots f_{B+1}$, the second and all even breakpoints by the sequence $G := g_1 g_2 \cdots g_{B+1} g_1 g_2 \cdots g_{B+1}$.

To see the backward direction of the statement in the lemma, a bin vector $\mathbf{b}$ for $(\mathbf{S}, B)$ resulting in $\ell$ breakpoints can be augmented to a bin vector $\mathbf{b}'$ for $(\mathbf{S}', B')$ by adding one unit for each of the new resources. This does not exceed the bound $B'$. Now, in $(\mathbf{S}', B')$ there will be the original breakpoints and a breakpoint in the middle of each inserted sequence. This shows that $\mathbf{b}'$ results in $\ell + k$ breakpoints for $(\mathbf{S}', B')$, as claimed.

To consider the forward direction, let $\mathbf{b}'$ be a solution to $(\mathbf{S}', B')$. Because every resource must be available at least once, and $B' - d' = 3B + 2 - (d + 2B + 2) = B - d$, at most $B - d < B$ entries of $\mathbf{b}'$ can be more than one. Therefore, at least one of the resources $f_i$ is available only once, and at least one of the resources $g_j$ is available only once. Hence, there must be at least one breakpoint within each of the $k$ inserted stopping sequences. Let $k + \ell$ be the number of breakpoints induced by $\mathbf{b}'$ and $\mathbf{b}$ the projection of $\mathbf{b}'$ to the original resources. Since all resources must have at least one unit and by choice of $B'$ and $d'$ we know that $\mathbf{b}$ sums to less than $B$.

Now, if a subsequence of $\mathbf{S}$ not containing any $f$ or $g$ resources can be packed with the resources $\mathbf{b}'$, this subsequence can also be packed with $\mathbf{b}$. Hence, $\mathbf{b}$ does not induce more than $\ell$ breakpoints in the instance $(\mathbf{S}, B)$ with pre-specified breakpoints. □

**Theorem 3.7** *The sequential unit vector packing problem is strongly NP-hard.*

**Proof.** By Lemma 3.5 and Lemma 3.6, with the additional observation that all used numbers are polynomial in the size of the original graph. □

### 3.3.2 Polynomially Solvable Cases and FPT

Here, we consider the influence of parameters on the complexity of the problem, and ask whether fixed parameter tractable algorithms can exist, see Definition 2.2.

If the windows in the vector packing problem are limited to length three, the problem can be solved in polynomial time: There is no interaction between the resources, thus, it is impossible that avoiding a breakpoint induced by one resource depends upon the multiple availability of another resource. Hence, a natural greedy algorithm that always takes the resource that currently causes most breakpoints is optimal. Additionally, Lemma 3.5 shows that the problem is NP-hard even if all windows have length 4. Hence, the parameter window size does not allow an FPT-algorithm if P$\neq$NP.

On the other hand, for integral $\mathbf{S}$ and $B$, the parameter $B$ allows (partly because $d \leq B$) to enumerate (see Section 3.4.3) and evaluate (Section 3.4.4) the number of breakpoints in time $f(B) \cdot n^{O(1)}$, i.e., this is an FPT-algorithm.

A constant upper limit on the number of breakpoints allows to enumerate all positions of breakpoints and to determine the necessary bin vector in polynomial time. Note that this is not an FPT algorithm.

## 3.4 Practical Algorithms

In this section we consider different practical algorithms for sequential vector packing together with some algorithmic questions about the enumeration and evaluation of solutions that arise in the context of the second heuristic presented here.

### 3.4.1 Greedy Algorithms

In this section we analyze two natural greedy heuristics. Given an input $(\mathbf{S}, B)$ we denote by $k(\mathbf{b})$ the minimal number of breakpoints needed for a fixed bin vector $\mathbf{b}$. Observe that it is relatively easy to calculate $k(\mathbf{b})$ in linear time. We will discuss this in more detail in Section 3.4.4. The two greedy algorithms we discuss here are GREEDY-GROW which grows the bin vector greedily starting with the all one vector and GREEDY-SHRINK which shrinks the bin vector starting with a bin vector $\mathbf{b}$ with $k(\mathbf{b}) = 0$ that initially ignores the bin size $B$.

Also in the light of the following observations it is important to specify the tie-breaking rule for the case that there is no improvement at all after the addition

**Algorithm 1**: Algorithm GREEDY-GROW

```
input  : an instance $(\mathbf{S}, B)$ of the sequential vector packing problem
output: bin vector $\mathbf{b}$

$\mathbf{b} \longleftarrow \mathbf{1}$ ;  $B_{\text{curr}} \longleftarrow d$
while $B_{curr} < B$ do
    // add the resource by which the most breakpoints are saved
    $r_{\text{greedy}} \longleftarrow \text{argmin}_{1 \leq r \leq d} k(b_1, \ldots, b_r + 1, \ldots, b_d)$
    $b_{r_{\text{greedy}}} \longleftarrow b_{r_{\text{greedy}}} + 1$ ;  $B_{\text{curr}} \longleftarrow B_{\text{curr}} + 1$
end
return $\mathbf{b}$
```

**Algorithm 2**: Algorithm GREEDY-SHRINK

```
input  : an instance $(\mathbf{S}, B)$ of the sequential vector packing problem
output: bin vector $\mathbf{b}$

// start with minimal bin vector that incurs no breakpoints
$\mathbf{b} \longleftarrow \sum_{i=1}^{n} \mathbf{s}_i$ ;  $B_{\text{curr}} \longleftarrow \sum_{i=1}^{d} b_i$
while $B_{curr} > B$ do
    // remove the resource by which the smallest increase in breakpoints is
      incurred
    $r_{\text{greedy}} \longleftarrow \text{argmin}_{1 \leq r \leq d} k(b_1, \ldots, b_r - 1, \ldots, b_d)$
    $b_{r_{\text{greedy}}} \longleftarrow b_{r_{\text{greedy}}} - 1$ ;  $B_{\text{curr}} \longleftarrow B_{\text{curr}} - 1$
end
return $\mathbf{b}$
```

of a resource. We show next that GREEDY-GROW can be forced to produce a solution only by this tie breaking rule, which is an indicator for its bad performance:

**Observation 3.8** *Given any instance $(\mathbf{S}, B)$, this instance can be modified to an instance $(\mathbf{S}', B')$, with $n' = n, d' = 2d, B' = 2B$ such that all of* GREEDY-GROW*'s choices of which resource to add depend entirely on the tie-breaking rule.*

The idea is to split each resource $r$ into two resources $r_1, r_2$ and to replace each occurrence of $r$ in a demand vector $\mathbf{s}$ by a demand for $r_1$ and $r_2$. We call this transformation *doubling* and will come back to it in the experimental section. Then, considering GREEDY-GROW's approach to reduce the number of breakpoints, increasing $r_1$ or $r_2$ alone is not enough. Only if $r_1$ and $r_2$ are both increased, the number of breakpoints may decrease. That is, for all resources the number of saved breakpoints in the beginning is zero, and greedy is forced to take an arbitrary resource in Step 1 and then the partner of this resource in Step 2. Then GREEDY-GROW again chooses an arbitrary resource in Step 3 and its partner in Step 4, and so on. With this scheme it is obvious that GREEDY-GROW can be fooled to produce arbitrary solutions.

It follows that GREEDY-GROW with an unspecified tie-breaking rule can be led to produce arbitrarily bad solutions. Also GREEDY-SHRINK can produce bad solutions depending on the tie breaking scheme as the following observation shows.

**Observation 3.9** *There are instances with $d$ resources on which the solution produced by* GREEDY-SHRINK *is a factor of $\lfloor d/2 \rfloor$ worse than the optimal solution, if the tie breaking-rule can be chosen by the adversary.*

Let $k = \lfloor d/2 \rfloor$, consider the following unit vector instance with $2k$ resources and $B = 3k$:

$$1 \cdots k \; 1 \cdots k (k+1)(k+1)(k+2)(k+2) \cdots (2k)(2k) \ .$$

At the beginning of the algorithm $\mathbf{b}$ is set to $(2, \ldots, 2)$. In the first step the removal of each of the resources incurs one breakpoint. Therefore, GREEDY-SHRINK deletes an arbitrary resource depending on the tie-breaking scheme. We let this resource be one of the last $k$ ones. After this deletion the situation remains unchanged except for the fact that the chosen resource must not be decreased any more. It follows that in $k$ steps GREEDY-SHRINK sets the last $k$ resources to one, which incurs a total cost of $k$, whereas the optimal solution sets the first

$k$ resources to one, which incurs a cost of $1$. Thus, the ratio of greedy versus optimal solution is $k = \lfloor d/2 \rfloor$.

For the experiments (see Section 3.5) we use for both heuristics a *round-robin* tie breaking rule that cycles through the resources. Every time a tie occurs it chooses the cyclic successor of the resource that was increased (decreased) in the last tie.

## 3.4.2 Enumeration Heuristic

In this section we present an enumeration heuristic for integral demand vectors $\mathbf{s}_i \in \mathbb{N}^d$, $i \in \{1, \ldots, n\}$, that is inspired by a variant of Schöning's 3-SAT algorithm [114] that searches the complete hamming balls of radius $\lfloor n/4 \rfloor$ around randomly chosen assignments, see [42].

The following algorithm uses a similar combination of randomized guessing and complete enumerations of parts of the solution space that are exponentially smaller than the whole solution space. The idea is to guess uniformly at random (u.a.r.) subsequences $\mathbf{S}_{i_1,i_2}$ of the sequence that do not incur a breakpoint in a fixed optimal solution $\mathbf{b}_{\text{opt}}$. For such a subsequence we know that $\mathbf{b}_{\text{opt}} \geq \mathbf{w}_{i_1,i_2}$. In particular, if we know a whole set $W$ of such total demand vectors that all come from subsequences without breakpoints for $\mathbf{b}_{\text{opt}}$, we know that $\mathbf{b}_{\text{opt}} \geq \max_{\mathbf{w} \in W} \mathbf{w}$ must hold for a component-wise maximum. This idea leads to the RANDOMIZED HEURISTIC ENUMERATION (RHE) algorithm, see Algorithm 3. The parameter subsequence length can be set to meaningful values if (estimates of) the minimal number of breakpoints are available. The following lemma states the resulting success probability of the algorithm:

**Lemma 3.10** *Let $\mathbf{b}_{opt}$ be an optimal bin vector for an integral instance $(\mathbf{S}, B)$ and choose ssl as $\lfloor \frac{n}{2k+1} \rfloor + 1$, where $k$ is the minimal number of breakpoints. Then for each of the demand vectors $\mathbf{w}_{\underline{\sigma}_i,\overline{\sigma}_i}$, $i \in \{1, \ldots, p\}$ in Algorithm* RHE *it holds that $\Pr[\mathbf{w}_{\underline{\sigma}_i,\overline{\sigma}_i} \leq \mathbf{b}_{opt}] \geq \frac{1}{2}$ .*

**Proof.** A sufficient (but not necessary) condition for $\mathbf{w}_{\underline{\sigma}_i,\overline{\sigma}_i} \leq \mathbf{b}_{\text{opt}}$ is that the optimal solution $\mathbf{b}_{\text{opt}}$ has no breakpoint in the subsequence $\mathbf{S}_{\underline{\sigma}_i,\overline{\sigma}_i}$. There are $n - \text{ssl} + 1$ many intervals that are chosen uniformly at random. Each breakpoint can hit only $\text{ssl} - 1$ of them. Therefore, $n - \text{ssl} + 1 - k(\text{ssl} - 1)$ is a lower bound on the number of intervals without breakpoints. This bounds the probability of choosing an interval that contains no breakpoint in its interior from below by $\frac{n-\text{ssl}+1-k(\text{ssl}-1)}{n-\text{ssl}+1}$. By equating this with $\frac{1}{2}$ we get the above choice for ssl without the flooring. By rounding down to the next integer the success probability cannot decrease. □

**Algorithm 3**: RANDOMIZED HEURISTIC ENUMERATION (RHE)

```
input : an instance $(\mathbf{S}, B)$ of the sequential vector packing problem, the
        subsequence length ssl and a number $p$ of repetitions
output: a feasible solution $\mathbf{b}$

1  $\mathbf{t} \longleftarrow 0$
2  for $i \in \{1 \dots p\}$ do
3      $\underline{\sigma}_i \longleftarrow_{\text{u.a.r}} \{0, \dots, n - \text{ssl}\}$ ; $\overline{\sigma}_i \longleftarrow \underline{\sigma}_i + \text{ssl}$
4      $\mathbf{t} \longleftarrow \max\left\{\mathbf{t}, \mathbf{w}_{\underline{\sigma}_i, \overline{\sigma}_i}\right\}$
5  end
6  minstops $\longleftarrow \infty$
7  forall $\mathbf{b} \in \{\mathbf{b}' \mid \mathbf{b}' \geq \mathbf{t}, \sum_{j=1}^{d} b'_j = B\}$ do
8      stops $\longleftarrow$ evaluate(b)
9      if stops < minstops then minstops $\longleftarrow$ stops; $\mathbf{b}_{\text{rhe}} \longleftarrow \mathbf{b}$
10 end
11 return $\mathbf{b}_{\text{rhe}}$
```

As the value of $k$ is not known a priori we use (over-)estimates in the experiments, which we adapt in the course of the algorithm as described in Section 3.5. Note that an overestimate of $k$ leads in general to a success probability greater than $\frac{1}{2}$ but to a smaller subsequence length than in the lemma. The first subsequence that is guessed increases the lower bound vector by its full length. Subsequent guesses can, but need not, improve the lower bounds. The growth of the lower bound depends on the distribution of demand vectors in the fixed input sequence and is therefore difficult to analyze for arbitrary such sequences. On the other hand, analyzing the growth of the lower bound seems possible for random input sequences, but we doubt that this would give any meaningful insights. For this reason, we only give experimental evidence that the algorithm performs well, see Section 3.5.

### 3.4.3 Enumeration

As easy as the enumeration in the second phase of our RHE-algorithm looks, this should be done efficiently. So let us have a look at the problem at hand: We want to enumerate all possible $b_1, \dots, b_d$ with sum $B$ and individual lower and upper bounds $\ell(i), u(i) \in \{0, \dots, B\}$ on the summands $\ell(i) \leq b_i \leq u(i)$, $i \in \{1, \dots, d\}$. For short, we also denote these bounds as vectors $\mathbf{l}$ and $\mathbf{u}$. In the literature on combinatorial generation algorithms such summations with upper bounds only are known as *(d)-compositions with restricted parts*, see [111] or

[99]. There is a bijection to combinations of a multiset. All compositions with restricted parts can be enumerated by a constant amortized time (CAT) algorithm, which can be easily extended to the case with lower bounds without changing the CAT behavior. We give the modified algorithm SUMMATIONS$(B, d, U)$ that enumerates the compositions with restricted parts in colexicographical order for convenience and refer to [111] for its unchanged analysis. The total number of compositions with restricted parts for given $\mathbf{l}$ and $\mathbf{u}$ is the Whitney number of the second kind of the finite chain product $(\overline{u(1) - \ell(1)} + \overline{1}) \times \cdots \times (\overline{u(r) - \ell(r)} + \overline{1})$, where $\overline{x}$ denotes a chain of $x$ elements, see again [111] for details.

```
Procedure SUMMATIONS(position $p$, resource $r$, bound $n$)
input  : dimension $d$, lower bound vector $\mathbf{l}$, upper bound vector $\mathbf{u}$,
         sum $B$
output : SUMMATIONS($B - \sum_{r=1}^{d} \ell(r), d+1, \sum_{r=1}^{d} u(r) - \ell(r)$)
         evaluates all $d$-compositions with restricted parts for the
         above parameters.

if $p = 0$ then
    evaluate b
else
    for $c \in \{\max(0, p - n + u(r) - \ell(r)) \ldots \min(u(r) - \ell(r), p)\}$ do
        $b_r \longleftarrow c + \ell(r)$
        SUMMATIONS($p - c, r - 1, n - u(r) + \ell(r)$)
    end
    $b_r \longleftarrow \ell(r)$
end
```

The initial call is SUMMATIONS$(B', d+1, U')$ for $B' = B - \sum_{r=1}^{d} \ell(r)$ and $U' = \sum_{r=1}^{d} u(r) - \ell(r)$. This algorithm has CAT behavior for $B' \leq U'/2$. For $B' > U'/2$ there is a similar algorithm that can be adapted from algorithm GEN2 in [111]. We sum up the results of this section in the following theorem.

**Theorem 3.11** *The $d$-compositions with restricted parts and lower bounds necessary for algorithm* RHE *can be enumerated in constant amortized time.*

## 3.4.4 Evaluation

For the general problem with demand vectors $\mathbf{s}_i \in \mathbb{Q}_+^d$, $i \in \{1, \ldots, n\}$, the evaluation of a given bin vector $\mathbf{b}$ can be done in the obvious way in $O(n \cdot d)$ time: Scan through the sequence starting at the last breakpoint $\pi_\ell$ (initially at

$\pi_0 = 0$) updating the total demand vector $\mathbf{w}_{\pi_\ell,i}$ of the current bin until the addition of the next vector $\mathbf{s}_{i+1}$ in the sequence would make the demand vector exceed $\mathbf{b}$. Then add breakpoint $\pi_{\ell+1} = i$ and continue the scan with the next bin starting from there.

In the special case of sequential unit vector packing the runtime can be improved to $O(n)$, since for each demand vector only one of the $d$ resources needs to be updated and checked.

For a single evaluation of a bin vector this algorithm is basically the best one can hope for. On the other hand, in the setting of our heuristic enumeration algorithm where many bin vectors are evaluated, the question arises, whether we can speed up the evaluations if we allow for preprocessing. We describe an approach that we developed for our application, that is, an approach for the sequential unit vector packing problem with large values of $n$ compared to $k$, the number of breakpoints. It is possible to extend parts of the approach to the general problem with a loss in space efficiency.

A first simple approach builds an $(n \times d \times B)$ table $T_1$ as sketched in Figure 3.4.1. In this table we store in entry $T_1(p, r, \delta)$ the position of the next breakpoint in the sequence starting from position $p$ for a bin vector $\mathbf{b}$ with capacity $\delta$ for resource $r$, i.e., $b_r = \delta$, and $b_k = \infty$ for $k \neq r$. To evaluate a given bin vector $\mathbf{b}$ we start at position 1 and inspect positions $(1, r, b_r)$ for $1 \leq r \leq d$. The next breakpoint must be at the minimum of these values. Thus, we have

$$\pi_{i+1} = \min_{1 \leq r \leq d} T_1(\pi_i, r, b_r) \quad . \tag{3.4.1}$$

Equation 3.4.1 directly gives an $O(kd)$ algorithm for the evaluation of a bin vector. Here $k$ denotes as usual the number of breakpoints. On instances with $kd \ll n$ this is a speedup. The space complexity of this approach seems to be $\Theta(n \cdot d \cdot B)$ at first glance. But notice that between two occurrences of a resource $r$ in the sequence the value of $T_1(\cdot, r, \cdot)$ remains the same. More precisely, if for all $p'$ with $p_1 \leq p' < p_2$ it holds that $s_{p'} \neq r$, then we have $T_1(p_1, r, \delta) = T_1(p_2, r, \delta)$ for all $\delta$. Let us call such an interval with equal entries for a given resource $r$ a *block*. An example can be found in Figure 3.4.1, where the blocks are depicted as grey rectangles. The total number of blocks is bounded by $n + d = O(n)$ because at each position exactly one block ends in the setting of unit vector packing. Also the answer vectors in the blocks need not be stored explicitly: In the $i$-th block of resource $r$ the table entry for $b_r$ is simply the position before the end position of block $i + b_r$ as indicated by the exemplary arrows in the figure. Therefore, in our approach we store the block structure in an array of size $O(n)$ to get a constant lookup time for a given table entry $T_1(p, r, \delta)$. More precisely, we store $d$ arrays $\{A_1, \ldots, A_d\}$ of total size

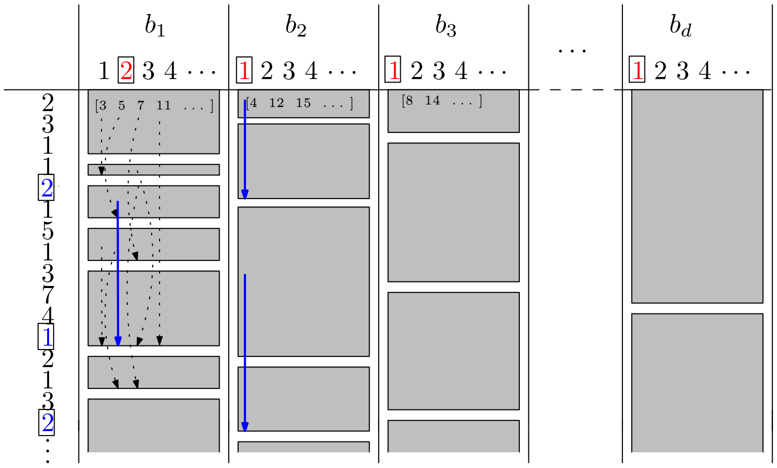


**Figure 3.4.1:** *Simple data structure that accelerates the evaluation of a bin vector. The first column shows an exemplary unit vector demand sequence, i.e., the rows correspond to positions in* $\mathbf{S}$*. The exemplary entries in the table stand for the position of the next breakpoint in the sequence starting from the current row for a bin vector with capacity* $i$ *for the resource of the current column and no resource bound for the other resources. The solid arrows show the minima of Equation 3.4.1 for the example bin vector* $(2, 1, 1, \ldots, 1)$*. Breakpoints are highlighted in the sequence (leftmost column). The exemplary dotted arrows indicate the end positions of the blocks, before which the relevant breakpoints are located.*

$O(n)$, such that $\{A_r(i)\}$ gives the end position of block $i$ of resource $r$ or equivalently the position of the $i$-th occurrence of $r$ in $\mathbf{S}$. It is easy to see that the block structure can be (pre-)computed in linear time.

However, with this approach a different problem arises: After the computation of breakpoint $\pi_{i+1}$, we need to know at which positions we should access each of the arrays next. To answer this question we introduce a second table. Let $T_2$ be an $(n \times 2)$-table that stores in $T_2(p, 1)$ the index of the (unique) new block[1] that starts at position $p$ and in $T_2(p, 2)$ the index of the current block of resource $(p \bmod d) + 1$ in array $A_{(p \bmod d)+1}$. In order to recompute the indices for breakpoint $\pi_{i+1}$ we read the $d$ rows $\{T_2(\pi_{i+1} - d + 1, \cdot), \ldots, T_2(\pi_{i+1}, \cdot)\}$. Each resource $r$ occurs once in the second column of the read rows and might occur several times in the first column. Take as index for resource $r$ the value of the last occurrence of $r$ in the read rows, regardless of the column, i.e., the occurrence with the highest row index. This approach correctly computes all new indices in the arrays $\{A_1, \ldots, A_d\}$ in $O(d)$ time, which is also the time that a single step takes without this index computation. Obviously, table $T_2$ needs $O(n)$ space. Alternatively, this table $T_2$ can be understood as a persistent version of the list containing for every resource its next occurrence, that is updated during a scan along the sequence. In this situation a general method for partially persistent data structures like [48] can be applied and yields the same time and space bounds. Altogether, we have shown the following theorem:

**Theorem 3.12** *Given a sequence* $\mathbf{S}$ *we can construct a data structure with* $O(n \cdot d \cdot B)$ *space and preprocessing time such that an evaluation query for sequential vector packing for a bin vector* $\mathbf{b}$ *takes* $O(kd)$ *time, where* $k$ *denotes the number of breakpoints for* $\mathbf{b}$. *For sequential* unit *vector packing only* $O(n)$ *space and preprocessing time is needed.*

Note that for RHE if we have already found a bin vector with $k'$ many breakpoints we can stop all subsequent evaluations already after $k' \cdot d$ many steps.

## 3.5 Experiments

In this section we report on some experiments on real world data. The data are electronically available at `www.inf.ethz.ch/personal/mnunkess/SVP/`. All instances are sequential unit vector packing instances.

[1]Strictly speaking, the first column in table $T_2$ is not necessary as it simply reproduces the sequence. Here it clarifies the presentation and the connection to the technique in [48].

We implemented the greedy algorithms, the enumeration heuristic and the integer linear program. We performed our experiment on Machine B, see Appendix A.1.

### 3.5.1 Mixed Integer Program

Even if the ILP-formulation of Section 3.2.1 is a good starting point for our theoretical results, it turns out that in practice only medium sized instances can be solved with it. One problem is the potentially quadratic number of edge flow variables that makes the formulation prohibitive already for small instances. To reduce the number of variables, it is helpful to have an upper bound on the length of the longest edge. One such bound is of course $B$, but ideally there are smaller ones. As our real-world instances are windowed instances the window size is a trivial upper bound that helps to keep the number of variables low. A further problem is that, even if the bin vector is already determined, the MIP-solver needs to branch on the $x$ and $y$ variables to arrive at the final solution. This can be avoided by representing the bin vector components $b_r$ as a sum of 0-1-variables $z_i^r$, such that $b_r = \sum_i z_i^r$ and $z_i^r \geq z_{i+1}^r$. If an edge $e$ uses $i$ units of resource $r$, i.e., the $r$-th entry of $w_e$ is $i$, we include a constraint of the form $y_e \leq z_i^r$. This allows to use the edge only if there are at least $i$ resources available. With these additional constraints, the edge variables $y$ and $x$ need no longer be binary. For integral values of $z_i^r$, only edge-variables that do not exceed the bin vector can have a value different from zero, so that in this case every fractional path of the resulting flow is feasible with respect to the bin vector, and thus all paths have the same (optimal) number of breakpoints (otherwise a shorter path could be selected resulting in a smaller objective function value). With this mixed integer linear program the number of branching nodes is drastically reduced, but the time spent at every such node is also significantly increased. Still, the overall performance of this program is a lot better than the original integer program. Small instances (inst2 and inst3) can now be solved to optimality within a few minutes, cf. Table 3.1 that summarizes information on our instances. The bigger instance inst4 of dimension 22 can be solved to optimality for total bin sizes in the range from 22 to 130 within less than 3 hours.

We observed that on this mixed integer program feasible solutions are found after a small fraction of the overall running time. Hence, we consider this approach also as an alternative heuristic to come up with good solutions.

### 3.5.2 Setup and Computational Results

We will mainly compare solution qualities, because the running times of the different approaches are orders of magnitude apart. On many of the instances a calculation for a fixed $B$ takes at most some seconds for the greedy algorithms and several hours for the mixed integer linear program. For our experiments, we let the enumeration heuristic run for 10 minutes which seems like a realistic maximum time that an "online" user would be willing to wait for a result in our application. This value is relatively arbitrary, we also observed good results for shorter running times. We then fix the number of repetitions of the guessing phase of RHE to be as many as it takes to let $\|\mathbf{t}\|_1$, the sum of the guessed lower bounds, exceed some fraction of $B$. This fraction is initially chosen as $99\%$ and adaptively decreased after each run as long as the targeted time of 10 minutes is not exceeded. The subsequence length is initially fixed with respect to the estimated number of breakpoints that we get by running both greedy approaches. We set it to one half times the average distance between two breakpoints and increase it adaptively if the lower bound does not grow any more after a fixed number of repetitions in the initialization phase. By the adaptive choice of both the subsequence length and the fraction the algorithm is robust with respect to changing values of $B$, $d$ and the time that it is run.

In Figure 3.5.1 we show the relative performances on our biggest real world instance (Inst1). The different data points correspond to the algorithms GREEDY-GROW, GREEDY-SHRINK, RHE and the linear relaxations of the two different ILP formulations, i.e., the one with ("frac") and the one without ("frac_d") the $z$-variables introduced in Section 3.5.1. The values represent the ratio of the solution found by the algorithm to the optimal integral solution that we calculated using the mixed integer programming formulation. The figure shows that for small values of $B$ GREEDY-GROW produces results that are close to optimal, whereas for bigger values the quality gets worse. An explanation for this behavior is that GREEDY-GROW builds a solution iteratively. As the results of Section 3.4.1 show, the greedy algorithms can be forced to take decisions based only on the tie-breaking rule. On this instance tie-breaking is used at several values of $B$, which leads to an accumulation of errors in addition to the inherent heuristic nature of the method. Note that by definition GREEDY-SHRINK is optimal for $B = \|\sum_{i=1}^{n}\mathbf{s}_i\|_1$, which is 196 on this instance. In order to have a meaningful scale we let the $x$-axis stop before that value.

In Figure 3.5.2(a) we present the quality of the solutions delivered by RHE relative to the optimal solution on four further instances. Note that for different instances different values of $B$ make sense. Instance Inst1-doubled is obtained from Inst1, by the doubling transformation used in Observation 3.8.

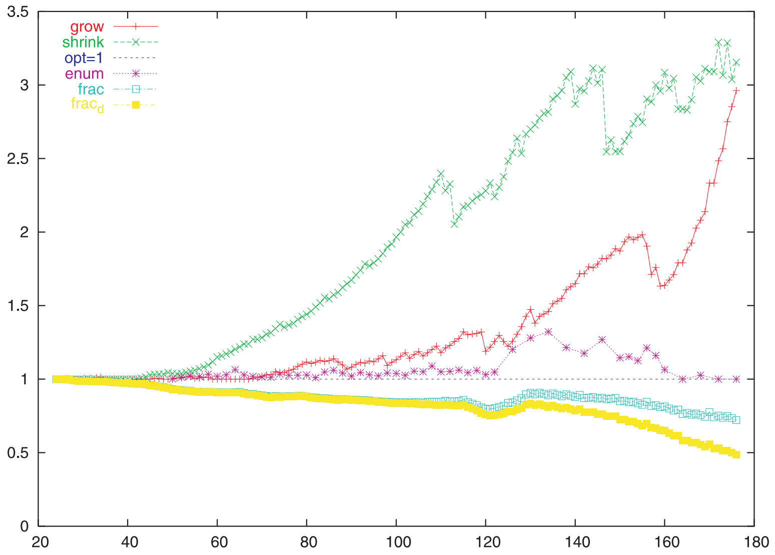


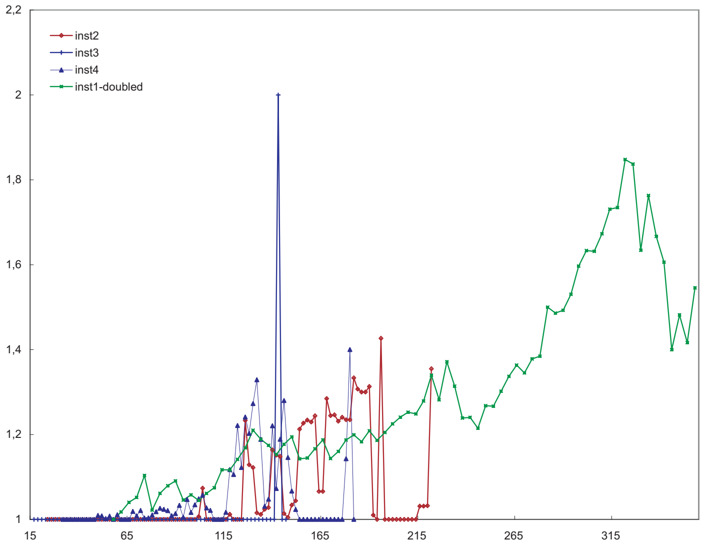
**Figure 3.5.1:** *Computational results for the different approaches. Plot of ratio of solution value to optimal solution value versus total bin size* $B$.

In Figure 3.5.2(b) we compare the best of the two greedy results to the result of RHE[2] . Instance rand-doubled is an instance where first the demand unit vectors are drawn uniformly at random and then a doubling transformation is done to make the instance potentially more complicated. It could not be solved to optimality by our MIP approach and does therefore not occur in Figure 3.5.2(a). Compared to the other instances the greedy algorithms do not perform too badly. One reason for this is that we chose a uniform distribution for the resources. Therefore, the tie-breaking rules make the right choices "on average". On the other hand, on the doubled real-world instance Inst1-doubled RHE gives superior results and in particular for higher values of $B$ the greedy algorithms perform comparatively poorly.

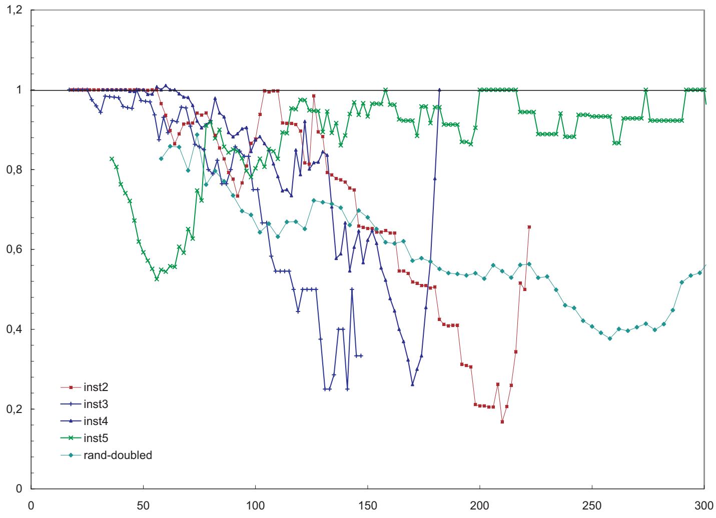
## 3.6 Open Problems

The main open problems for sequential vector packing are:

[2]Note that even if we use the greedy algorithms to determine the parameter settings of RHE, these results are not visible to RHE.



(a) Ratio of enumeration heuristic to optimal solution on five instances.



(b) Ratio of enumeration heuristic to best of greedy algorithms on five instances.

**Figure 3.5.2:** *Results on the instances of Table 3.1.*

| name | $n$ | $d$ | window size | note |
|---:|---|---|:---:|---|
| inst1 | 4464 | 24 | 28 | |
| inst1-doubled | 8928 | 48 | 56 | inst1, “doubled” |
| inst2 | 9568 | 8 | 28 | |
| inst3 | 7564 | 7 | 28 | |
| inst4 | 4464 | 22 | 28 | |
| rand-doubled | 2500 | 26 | 2500 | “doubled” random instance |

**Table 3.1:** *Summary information on the instances*

- What is the best achievable approximation ratio with respect to the number of breakpoints or the size of the bin-vector?
- How is the “best-possible” trade-off curve between these two goals?
- Is there a method to analyze the growth of the lower bounds of the randomized enumeration heuristic in a meaningful way?
- More generally: Is there a practical algorithm or heuristic that clearly outperforms our algorithms on instances similar to our test instances?

# Chapter 4

# Foundations of LP-based Optimization Techniques

Die beste Basis für die Grundlage ist das Fundament.
(toast)

In this section we present the theoretical foundations of the railroad optimization related material covered in Chapter 5. In contrast to the very brief algorithmic preliminary Chapter 2 we will describe in more detail the underlying theory because this topic is further off the classical computer science canon.

## 4.1 Background

Generally, the entity we are dealing with are linear programs of the form

$$z_{\text{LP}} = \max\left\{cx : Ax \leq b, x \in \mathbb{R}^n_+\right\} \quad , \tag{LP}$$

and integer linear programs of the form

$$z_{\text{IP}} = \max\left\{c'x : A'x \leq b', x \in \mathbb{Z}^n_+\right\} \quad , \tag{IP}$$

where $A, A' \in \mathbb{R}^{m\times n}$, $b, b' \in \mathbb{R}^m$, $c, c' \in \mathbb{R}^n$. In this presentation we will not distinguish between column and row vectors notation-wise, i.e., it should always be possible to infer a meaningful interpretation of the vectors as column and row vectors from the formulae. This slightly sloppy approach is also adopted in the classical textbook [97]. In contrast to the general integer programming problem (for example in the form IP) the linear programming problem (for example in the form LP) can be solved in polynomial time [79].

One goal of this section is to show how to use linear programming methods to obtain solutions to integer programs of the form IP. One obvious way to do so is to set $A = A', b = b', c = c'$. In this case LP is called the *linear relaxation* of IP and it holds $z_{\text{IP}} \leq z_{\text{LP}}$. Furthermore, if the optimal solution $x_{\text{opt}}$ to $z_{\text{LP}}$ happens to be integral, it is an optimal solution to IP. Obviously, for NP-complete problems this easy case does not occur in general. Two approaches that use linear programming to find optimal or at least provably good solutions to IP are *branch-and-cut* and *branch-and-price* algorithms.

In this chapter, we review some of the basic theory necessary for the understanding of these approaches: Linear Programming Duality, the Simplex Algorithm, some fundamental theorems from Polyhedral Theory, and finally the basic branch-and-cut and branch-and-price approaches themselves. As most of this is well-established theory that has been presented from different angles in several textbooks we will not prove all results. The outline and most of the theorems of the first part of this section follow parts of [97] and are complemented with material from [81, 29, 128, 6, 133, 115, 91].

## 4.2 Duality

With every (primal) linear program we can associate a *dual* linear program, the dual of which is again the same primal linear program. It has proven fruitful to study the properties of such primal/dual pairs of linear programs. Here, we will consider the following linear program P and its dual D:

$$z_{\text{LP}} = \max\{cx : Ax \leq b, x \in \mathbb{R}^n_+\} \tag{P}$$

$$w_{\text{LP}} = \min\{ub : uA \geq c, u \in \mathbb{R}^m_+\} \tag{D}$$

It is straight-forward to show that the dual of D is again P. If both P and D are feasible, we can get upper bounds to $z_{\text{LP}}$ from the dual:

**Lemma 4.1 (Weak Duality)** *If $x^*$ is primal feasible and $u^*$ is dual feasible, then $cx^* \leq z_{LP} \leq w_{LP} \leq u^*b$.*

**Proof.** Dual feasibility and nonnegativity of $x^*$ implies

$$cx^* \le u^* A x^* \qquad \Big( = \sum_{j=1}^{n} \Big( \sum_{i=1}^{m} a_{ij} u_i \Big) x_j \Big) \ . \tag{4.2.1}$$

Primal feasibility and nonnegativity of $u^*$ implies

$$\Big( \sum_{i=1}^{m} \Big( \sum_{j=1}^{n} a_{ij} x_j \Big) u_i = \Big) \qquad u^* A x^* \le u^* b \ . \tag{4.2.2}$$

As this holds for any primal-dual feasible pair, it follows $z_{\text{LP}} \le w_{\text{LP}}$ . □

The strong version of the above theorem states that for primal/dual feasible linear programs even equality holds for the primal objective value $z_{\text{LP}}$ and the dual $w_{\text{LP}}$.

**Theorem 4.2 (Strong Duality)** *If either of $z_{LP}$ and $w_{LP}$ is finite, then both $P$ and $D$ have finite optimal value and $z_{LP} = w_{LP}$.*

The correctness of this theorem follows from the correctness of the simplex algorithm that we review in the next section.

**Corollary 4.3** *There are four possibilities for a dual pair of problems P and D.*

1. *$z_{LP}$ and $w_{LP}$ are finite and equal.*
2. *$z_{LP} = \infty$ and[1] D is infeasible.*
3. *$w_{LP} = -\infty$ and P is infeasible.*
4. *Both P and D are infeasible.*

By strong duality $(x^*, u^*)$ is a pair of primal/dual optimal solutions if and only if $cx^* = bu^*$. For this to happen, equations (4.2.1) and (4.2.2) have to hold with equality. From this the following result about the structure of optimal solutions follows (after inspection of equations (4.2.1) and (4.2.2)).

**Theorem 4.4 (Complementary Slackness)** *Let $x$ and $u$ be primal and dual feasible solutions, respectively. Then $x$ and $u$ are both optimal if and only if the following two conditions hold:*

[1]In this section we use the slightly sloppy notation "$z_{\text{LP}} = \infty$" to say that the primal is unbounded, respectively for the dual.

**Primal complementary slackness** *For each column index* $1 \le j \le n$ *either* $x_j = 0$ *or* $\sum_{i=1}^m a_{ij}u_i = c_j$.

**Dual complementary slackness** *For each row index* $1 \le i \le m$ *either* $u_i = 0$ *or* $\sum_{j=1}^n a_{ij}x_j = b_i$.

This simple theorem is crucial both for the general theory of linear programming and for the analysis of many approximation algorithms. Another simple but useful result is *Farkas' Lemma*, which gives a certificate for the infeasibility of a system of linear inequalities.

**Theorem 4.5 (Farkas' Lemma)** *Either* $Q := \{x \in \mathbb{R}^n_+ : Ax \le b\}$ *is nonempty or (exclusively) there exists* $v \in \mathbb{R}^m_+$ *such that* $vA \ge 0$ *and* $vb < 0$.

**Proof.** Consider $z_{\text{LP}} = \max\{0x : Ax \le b, x \in \mathbb{R}^n_+\}$ and its dual $w_{\text{LP}} = \min\{vb : vA \ge 0, v \in \mathbb{R}^m_+\}$. Setting $v = 0$ yields a dual feasible solution. Therefore, by Corollary 4.3 only the following two cases can occur:

1. $z_{\text{LP}} = w_{\text{LP}} = 0$, then $Q \ne \emptyset$ and for all $v \in \mathbb{R}^m_+$ it holds that if $vA \ge 0$ then $vb \ge 0$.

2. $w_{\text{LP}} = -\infty$, then $Q = \emptyset$ and there must be a $v \in \mathbb{R}^m_+$ such that $vb < 0$ and $vA \ge 0$.

$\square$

There are more variants of the Farkas' Lemma that can easily be shown to be equivalent. An exemplary one is

**Corollary 4.6** *Either* $\{x \in \mathbb{R}^n_+ : Ax = b\} \ne \emptyset$ *or* $\{v \in \mathbb{R}^m : vA \ge 0, vb < 0\} \ne \emptyset$.

## 4.3 Simplex Algorithm

Before we present the simplex algorithm, we first introduce the necessary terminology to describe this algorithm concisely. For ease of presentation we consider linear programs with equality constraints, in particular we redefine what we mean by a linear program of form LP in this section:

$$z_{\text{LP}} = \max\{cx : Ax = b, x \in \mathbb{R}^n_+\} \tag{LP}$$
$$w_{\text{LP}} = \min\{ub : uA \ge c, u \in \mathbb{R}^m\} \tag{DLP}$$

The theorems on weak and strong duality and complementary slackness also hold for this formulation. We assume that redundant equations (in the linear algebra sense) in $A$ have been removed and thus $\text{rank}(A) = m \leq n$. From this it directly follows that there exists an $m \times m$ nonsingular submatrix $A_B = (a_{B_1}, \ldots, a_{B_m})$. Without loss of generality, we assume that $A_B$ consists of the first $m$ columns of $A$. Let the rest of $A$'s columns be the matrix $A_N$, so that we can write $A = (A_B, A_N)$ and $A_B x_B + A_N x_N = b$ for $Ax = b$. As $A_B$ is of full rank, it is meaningful to define $x_B = A_B^{-1} b$ and $x_N = 0$. The vector $x = (x_B, x_N)$ is a solution to $Ax = b$.

**Definition 4.7** • *The nonsingular $m \times m$ matrix $A_B$ is called a* basis.

- *The solution $x_B = A_B^{-1} b, x_N = 0$ is called a* basic solution *of $Ax = b$.*
- *$x_B$ is the vector of* basic variables *and $x_N$ is the vector of* nonbasic variables.
- *If $x_B = A_B^{-1} b \geq 0$ then $(x_B, x_N)$ is called a* basic primal feasible solution *to LP and $A_B$ is called a* primal feasible basis.

Similarly, for the dual let $c = (c_B, c_N)$ be the partition of $c$ induced by the index set $B$ of the basis. We set $u = c_B A_B^{-1} \in \mathbb{R}^m_+$, the motivation for which are the primal complementary slackness conditions. Recall that we are considering a formulation with equality constraints in the primal LP. Therefore, the dual complementary slackness conditions are always fulfilled. With the above setting the slack $uA - c$ of the dual constraints is given by:

$$uA - c = c_B A_B^{-1}(A_B, A_N) - (c_B, c_N) = (\underbrace{0}_{``x_B"}, \underbrace{c_B A_B^{-1} A_N - c_N}_{``x_N"}) \ . \quad (4.3.1)$$

As also $x_N = 0$, the complementary slackness conditions are always fulfilled, so that the definition $u = c_B A_B^{-1}$ leads to a pair of primal and dual solutions $(x_B, x_N) = (A_B^{-1} b, 0)$ and $u = c_B A_B^{-1}$ that is complementary slack. On the other hand, it is not guaranteed that $u$ is a dual feasible solution. For this, we need $uA \geq c$, which simplifies to $c_B A_B^{-1} A_N \geq c_N$ from (4.3.1). If this inequality holds, $A_B$ is called a *dual feasible basis*. From the complementary slackness theorem it follows that if $A_B$ is primal and dual feasible, then $x = (x_B, x_N) = (A_B^{-1} b, 0)$ is an optimal solution to LP and $u = c_B A_B^{-1}$ is an optimal solution to DLP.

Given a basis $B$, it is useful to rewrite the LP in a form that reflects the choice of $B$. The following equivalent reformulation of LP is called the *canonical form* of a linear program:

$$\begin{aligned} z_{\text{LP}} = c_B A_B^{-1} b + \max(c_N - c_B A_B^{-1} A_N) x_N \\ x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ x_B \geq 0,\; x_N \geq 0 \end{aligned} \tag{LP$_B$}$$

Let $\overline{A}_N = A_B^{-1} A_N, \overline{b} = A_b^{-1} b$ and $\overline{c}_N = c_N - c_B A_B^{-1} A_N$. Then, LP$_B$ becomes

$$\begin{aligned} z_{\text{LP}} = c_B \overline{b} + \max \overline{c}_N x_N \\ x_B + \overline{A}_N x_N = \overline{b} \\ x_B \geq 0, x_N \geq 0 \ . \end{aligned} \tag{LP$_B$}$$

Analogously, we define $\overline{a}_j = A_B^{-1} a_j$ and $\overline{c}_j = c_j - c_B \overline{a}_j$. Then we can rewrite $LP_B$ in the following form:

$$\begin{aligned} z_{\text{LP}} = c_B \overline{b} + \max \sum_{j \in N} \overline{c}_j x_j \\ x_B + \sum_{j \in N} \overline{a}_j x_j = \overline{b} \\ x_B \geq 0, x_N \geq 0 \end{aligned} \tag{LP$_B$}$$

Apart from the importance for the general simplex algorithm the above form has intuitive appeal: For a feasible basis $B$ we can directly read off the values of the basic primal feasible solution $x^*$ associated with $B$ as $x^* = (\overline{b}, 0)$. Furthermore, dual feasibility (and therefore optimality for a primal feasible basis) is equivalent to $\overline{c} \leq 0$. The value

$$\overline{c}_N = c_N - c_B \overline{A}_N = c_N - u A_N \tag{4.3.2}$$

is called the *reduced price vector*. For minimization problems it is called the *reduced cost vector*. Observe that the prices can be calculated independently of each other, i.e.,

$$\overline{c}_j = c_j - u a_j \ . \tag{4.3.3}$$

The first summand $c_B \overline{b}$ in the objective function of LP$_B$ is the objective value for the current basic feasible solution, the sum $\sum_{j \in N} \overline{c}_j x_j$ expresses in terms of reduced prizes the potential improvement from increasing non-basic variables. From this formulation we can also read off simple upper bounds on the objective function, which we also use in our column-generation algorithm.

**Lemma 4.8** *Assume that* $\sum_{j=1}^n x_j \le \kappa$ *holds then* $c_B\overline{b} \le z_{LP} \le c_B\overline{b} + \kappa\overline{c}_{\max}$, *where* $\overline{c}_{\max} = \max_{j\in N} \overline{c}_j$.

Two given bases $A_B$ and $A_{B'}$ are *adjacent* if $|B \setminus B'| = |B' \setminus B| = 1$, i.e., $B'$ can be obtained from $B$ by exchanging one column. The simplex algorithm to be presented below works by moving from one basis to an adjacent one and iterating this step. For such a move we have to decide which column enters and which one leaves the basis. If all primal basic feasible solutions are *non-degenerate* in the sense defined below, we will see that the entering column determines the leaving column.

**Definition 4.9 (Degeneracy)** *A primal basic feasible solution* $x_B = \overline{b}, x_N = 0$ *is* degenerate *if* $\overline{b}_i = 0$ *for some* $i$.

**Lemma 4.10** *Suppose all primal basic feasible solutions are nondegenerate. If* $A_B$ *is a primal feasible basis and* $a_r$ *is any column of* $A_N$*, then the matrix* $(A_B, a_r)$ *contains, at most, one primal feasible basis other than* $A_B$.

**Proof.** All variables in $N \setminus \{r\}$ have to stay nonbasic, so that we can write the constraints as

$$\begin{aligned} x_B + \overline{a}_r x_r &= \overline{b} \\ x_B \ge 0, x_r &\ge 0 \ . \end{aligned}$$

**case 1** $\overline{a}_r \le 0$. Then as $x_B = \overline{b} - \overline{a}_r x_r$ it follows that to whatever positive value $\alpha > 0$ we set $x_r$, no basic variable will ever become zero. Therefore, there is no other primal feasible basis in $(A_B, a_r)$ than $A_B$.

**case 2** At least one component of $\overline{a}_r$ is positive. Then define $\lambda$ by the *minimum ratio rule*

$$\lambda_r = \min\{\overline{b}_i/\overline{a}_{ir} : \overline{a}_{ir} > 0\} = \overline{b}_s/\overline{a}_{sr} \ , \tag{4.3.4}$$

so that $\overline{b} - \overline{a}_r\lambda_r \ge 0$ and $\overline{b}_s - \overline{a}_{sr}\lambda_r = 0$. An adjacent primal feasible basis $A_{B^{(r)}}$ can be obtained from $A_B$ by deleting $B_s$ from $B$ and replacing it with $r$. By the nondegeneracy assumption no other basic variable than $x_s$ becomes zero and no other adjacent basis exists.

□

In order to move from the canonical form LP$_B$ to LP$_{B^r}$ one can execute one Gauss elimination step with pivot element $\overline{a}_{sr}$.

**Corollary 4.11** *Suppose $A_B$ is a primal feasible nondegenerate basis that is not dual feasible and $\overline{c}_r > 0$. Then if $\overline{a}_r \leq 0$, the primal is unbounded. Otherwise, at least one component of $\overline{a}_r$ is positive and $A_{B(r)}$ the unique primal basis adjacent to $A_B$ that contains $a_r$ is such that $c_{B(r)} x_{B(r)} > c_B x_B$.*

Having introduced the basic terminology we can now concisely describe the main part of the primal simplex algorithm.

**Primal Simplex Algorithm, Phase II**

**Initialization** Start with a primal feasible basis $A_B$.

**Optimality Test** If $\overline{c}_N \leq 0$, $A_B$ is dual feasible, stop. $(x_B, x_N) = (\overline{b}, 0)$ is an optimal solution. Otherwise continue with the Pricing step.

**Pricing** Choose an $r \in N$ with positive reduced prize ($\overline{c}_r > 0$).

**Unboundedness test** If $\overline{a}_r \leq 0, z_{\mathrm{LP}} = \infty$.

**Basis change** Otherwise, find an adjacent primal feasible basis $A_{B(r)}$ that contains $a_r$. Set $B$ to $B^{(r)}$ and return to Optimality Test.

The choice of the entering variable $x_r$ is not specified here. There are various possible pricing rules, one of which is to take $r = \text{argmax}_{j \in N} \overline{c}_j$. However, in practice other rules are used, see [29, 126]. As we have seen in Lemma 4.10 under the nondegenerate assumption the leaving variable $s$ is well-defined and the values of the basic feasible solutions increase in each iteration. As there are only a finite number of bases the following theorem holds:

**Theorem 4.12** *Under the assumption that all basic feasible solutions are nondegenerate, Phase II terminates in a finite number of steps either with an optimal or an unbounded solution.*

If not all basic feasible solutions are nondegenerate, the above theorem still holds if one additionally uses specific rules to choose the leaving variable, see [14, 115].

To make the description of the primal simplex algorithm complete, it remains to specify how we can get (in Phase I) a primal feasible basis $A_B$ if the problem is feasible. To achieve this, one sets up a transformed linear program with additional penalty variables $x_i^a$ for each row, and changes signs of the rows such that $b \geq 0$.

$$z_a = \max\{-\sum_{i=1}^{m} x_i^a : Ax + Ix^a = b, (x, x^a) \in \mathbb{R}_+^{n+m}\} \qquad (\text{LP}^a)$$

This linear program is feasible: a basic feasible solution is $(x^a, x) = (b, 0)$. It is also not unbounded as $z_a \leq 0$ and thus has an optimal solution. It can be solved by the Phase II simplex algorithm above starting with the basis $(b, 0)$. If $z^a < 0$ the original LP must be infeasible, as it is impossible to set all penalty variables to zero and preserve feasibility of the original LP. If $z_a = 0$ any optimal solution has $x^a = 0$ and hence yields a feasible solution to the LP. If all penalty variables are nonbasic this solution is basic and can be directly used for Phase II. Otherwise a sequence of degenerate basis changes might yield a basic feasible solution to LP. Otherwise a case might occur where this is not possible. But then it can be shown that certain constraints in LP must be redundant and the corresponding penalty variables can be dropped. This leads to a basic feasible solution again, see [29].

One issue not discussed here is how to adapt the above standard simplex algorithm to problems in *general form*, where we have lower and upper bounds on the variables and on the constraints.

$$\begin{aligned} z_{\text{LP}} &= \max cx \qquad (\text{LP}_{\text{gen}}) \\ b &\leq Ax \leq d \\ e &\leq x \leq f \end{aligned}$$

It is of course possible to apply the above algorithm to such a formulation, but in practice one uses an algorithm that is tuned for this setting, in which then the definition of basic, nonbasic and dual variables and complementary slackness needs to be adapted, see [127, 29]. We will briefly encounter this type of problem in Section 5.6.4.

In general, an implementation of the simplex method will not store or calculate the matrices $\overline{A}_N$. Instead, it will store the current basis $A_B$ and calculate the necessary parts of $\overline{A}_N$, namely $c_N A_B^{-1}$ for the reduced cost vector $\overline{c}_N = c_N - (c_N A_B^{-1}) A_N$ and the column $\overline{a}_r = A_B^{-1} a_r$ for the minimum ratio test. Any simplex algorithm that follows this scheme is called *revised simplex method*. In a real implementation of the simplex algorithm one has to deal with diverse efficiency issues, numerical and stability problems that are out of the scope of this thesis, again see [29] for details.

## 4.4 Dual Simplex Algorithm

The dual simplex algorithm is the dual equivalent to the primal simplex algorithm: Instead of moving from one primal feasible basis to another until it is also dual feasible, it moves from one dual feasible basis to another until this basis is also primal feasible. The following theorem corresponds to Theorem 4.10 and Corollary 4.11 for the dual simplex algorithm.

**Theorem 4.13** *Let $A_B$ be a dual feasible basis with $\overline{b}_s < 0$. If $\overline{a}_{sj} \geq 0$ for all $j \in N$, then LP is infeasible; otherwise there is an adjacent dual feasible basis $A_{B^{(r)}}$, where $B^{(r)} = B \cup \{r\} \setminus \{B_s\}$ and $r \in N$ satisfies $\overline{a}_{sr} < 0$ and $r = \text{argmin}_{j \in N}\{\overline{c}_j / \overline{a}_{sj} : \overline{a}_{sj} < 0\}$.*

The simple proof can be found in [97]. The resulting Phase II of the dual simplex algorithm is symmetric to the Phase I primal simplex algorithm.

**Dual Simplex Algorithm, Phase II**

**Initialization** Start with a dual feasible basis $A_B$.

**Optimality Test** If $\overline{b}_N > 0$ $A_B$ is primal feasible, stop. $x_B = \overline{b}, x_N = 0$ is an optimal solution. Otherwise continue with the Pricing step.

**Pricing** Choose an $s \in N$ with $\overline{b}_s < 0$.

**Infeasibility test** If $\overline{a}_{sj} \geq 0$ $\forall j \in N$ LP is infeasible.

**Basis change** Otherwise, let $r = \text{argmin}_{j \in N}\{\overline{c}_j / \overline{a}_{sj} : \overline{a}_{sj} < 0\}$. $B \longleftarrow B^{(r)} = B \cup \{r\} \setminus \{B_s\}$, return to Optimality Test.

Often the dual simplex algorithm is preferred over the primal simplex algorithm because its implementations are usually faster. In particular, in a branch-and-cut setting it is preferable to use the dual simplex, see Section 4.6.

## 4.5 Polyhedral Theory

Here we present some connections of linear programming to polyhedral theory. Polyhedral aspects become particularly important when the goal is to solve an integer program by means of linear programming. The problem here is that the set of feasible solutions to IP is given implicitly via a linear program. We begin with the necessary definitions, then we present the most important theorems without proof and discuss their relevance in the context of practical approaches to solving large integer programs.

An *affine combination* of points $x^1, \dots, x^k$ in $\mathbb{R}^n$ is a linear combination $\lambda_1 x^1 +, \dots, +\lambda_k x^k$ such that $\sum_{i=1}^k \lambda_i = 1$.
A *conic combination* of points $x^1, \dots, x^k$ in $\mathbb{R}^n$ is a linear combination $\lambda_1 x^1 +, \dots, +\lambda_k x^k$ such that $\lambda_i \geq 0$ for all $i = 1, \dots, k$.
A *convex combination* of points $x^1, \dots, x^k$ in $\mathbb{R}^n$ is a linear combination $\lambda_1 x^1 +, \dots, +\lambda_k x^k$ such that $\sum_{i=1}^k \lambda_i = 1$ and $\lambda_i \geq 0$ for all $i = 1, \dots, k$.

The *affine hull* of a set $S \subseteq \mathbb{R}^n$, denoted by aff$(S)$, is the set of all points that are affine combinations of (a finite number of) points in $S$. Similarly, we define the *conic hull* cone$(S)$ and the *convex hull* conv$(S)$. A set of points is *affinely independent* if and only if none of the points is an affine combination of the other points.

**Definition 4.14** *A set $P \subseteq \mathbb{R}^n$ is a* polytope *if it is the convex hull of finitely many vectors. A set $C \subseteq \mathbb{R}^n$ is a* cone *if $x \in C$ implies $\lambda x \in C$ for all $\lambda \geq 0$. A cone $C$ is* polyhedral *if it can be represented as $\{x \in \mathbb{R}^n \mid Ax \leq 0\}$. A* polyhedron *$P \subseteq \mathbb{R}^n$ is the set of points that satisfy a finite number of linear inequalities; i.e., $P = \{x \in \mathbb{R}^n : Ax \leq b\}$.*

Any conic hull is a cone. A polyhedron is said to be *bounded* if it is contained in a box $[-\omega, \omega]^n \in \mathbb{R}^n$; it is *rational* if it can be represented as above by a matrix $A$ and vector $b$ with rational coefficients. We will always assume that all coefficients are rational. The *dimension* dim$(P)$ of $P$ is defined to be one less than the maximum number of affinely independent points in $P$.

**Definition 4.15** *An inequality $\pi x \leq \pi_0$ [or $(\pi, \pi_0)$], $\pi \in \mathbb{R}^n, \pi_0 \in \mathbb{R}$, is called a* valid inequality *for $P$ if it is satisfied by all points in $P$. It is called* supporting *for $P$ if it has a non-empty intersection with $P$.*

In the integer programming setting, valid inequalities are of particular interest.

**Definition 4.16** *The set $F$ defines a* face *of the polyhedron $P$ if $F = \{x \in P : \pi x = \pi_0\}$ for some valid inequality $(\pi, \pi_0)$ of $P$. If $F$ is a face of $P$ with $F = \{x \in P : \pi x = \pi_0\}$ the valid inequality $(\pi, \pi_0)$ is said to* represent *or* define *the face. The zero-dimensional faces of $P$ are called* extreme points. *$F$ is a* facet *of $P$ if $F$ is a face of $P$ and dim$(F) =$ dim$(P) - 1$.*

To exemplify these definitions we can give a simple geometric interpretation to the second form of Farkas' Lemma, Corollary 4.6: If (and only if) a vector $b$

is not contained in the cone $C = \text{cone}\{a_1, \ldots, a_n\}$ generated by the columns of matrix $A$, then there exists a valid inequality for $C$ separating it from $b$. More importantly, the above definitions allow us to state two important theorems in polyhedral theory concisely:

**Theorem 4.17 (Finite Basis Theorem (Minkowski, Weyl), [115])** *A convex cone is polyhedral if and only if it can be represented as a conic hull of finitely many vectors.*
*A set $P$ of vectors is a polyhedron if and only if $P$ can be represented as $P = Q + C$ for a polytope $Q$ and a polyhedral cone $C$.*

**Theorem 4.18 (Fundamental Theorem of Linear Programming, [91, 127])**
*For any linear program LP the following statements hold*

- *If there is not an optimal solution, then the linear program is either unbounded or infeasible.*
- *If the linear program has an optimal solution, then there is an optimal extreme point solution to this linear program.*
- *$x^* \in \mathbb{R}^n_+$ is an extreme point of $P = \{x \in \mathbb{R} \mid Ax = b, x \geq 0\}$ if and only if $x^*$ is a basic feasible solution of the system $Ax = b, x \geq 0$.*

The finite basis theorem states that there are two equivalent views of a polyhedron $P$: It can be seen as the intersection of half-spaces as expressed by the inequality system $Ax \leq b$ or it can be seen as the sum of a polytope and a polyhedral cone. Both the polytope and the cone are generated via convex/conic combinations by a finite set of points, therefore $P = \text{conv}(\{x^1, \ldots, x^k\}) + \text{cone}(\{x^{k+1}, \ldots, x^{k'}\})$ and one also says that $P$ is *finitely generated* by the sets $\{x^1, \ldots, x^k\}$ and $\{x^{k+1}, \ldots, x^{k'}\}$.

The fundamental theorem of linear programming, the proof of which does not necessarily rely on the simplex method, identifies the extreme points of the polyhedron defined by the LP and the basic feasible solutions produced by the simplex method.

The above theorems give a theoretical foundation to LP-based approaches for solving integer programs. Assume there are a finite number of solutions $S$ to a given integer linear program. Then $\text{conv}(S)$ can be described by a finite number of linear inequalities by the finite basis theorem. The maximum of a linear function over $\text{conv}(S)$ is attained in an extreme point $s$ of $\text{conv}(S)$, by the fundamental theorem, which implies $s \in S$. Therefore, if the description of $\text{conv}(S)$ by linear inequalities is given, the solution of the integer linear program can be obtained from the solution of a linear program. The assumption that $S$ is finite can be dropped, also without this assumption conv$(S)$ is a polyhedron. To sum up, the above theorems show that given a polyhedron $P = \{x \in \mathbb{R}^n_+ : Ax \leq b\}, S = \mathbb{Z}^n \cap P$ the integer linear programming problem $\max\{cx, x \in S\}$ can be solved by solving the problem $\max\{cx : x \in$ conv$(S)\}$ and this can be written and solved as a linear program if the description of conv$(S)$ is available.

From a theoretical point of view, the above insights raise the question which inequalities in a given formulation are necessary to describe a polyhedron. As it turns out, exactly the inequalities that represent facets are necessary, see [97].

**Lemma 4.19** *For each facet $F$ of $P$, one of the inequalities representing $F$ is necessary in the description of $P$.*

**Lemma 4.20** *Every inequality $a^r x \leq b$ in the formulation, that represents a face of dimension less than dim$(P) - 1$ is irrelevant to the description of $P$.*

For full-dimensional polyhedra these lemmata lead to a nice characterization of facets:

**Theorem 4.21 ([97])** *A full-dimensional polyhedron $P$ has a unique (to within scalar multiplication) minimal representation by a finite set of linear inequalities. In particular, for each facet $F_i$ of $P$ there is an inequality $a^i x \leq b_i$ (unique to within scalar multiplication) representing $F_i$ and $P = \{x \in \mathbb{R}^n : a^i x \leq b_i, 1 \leq i \leq t\}$.*

For polyhedra of lower dimension there are similar statements, see [97, 115].

## 4.6 Branch-and-Cut

In practice, the complete inequality description of conv$(S)$ typically has exponential size and is almost never available except for a few well-studied problems. On the other hand, this critique seems overly pessimistic as it is not necessary to have the complete description: For an optimal solution $x_{\text{opt}}$ with respect to a fixed objective function only a small part of the constraints $C$ will be binding, in the sense that $x_{\text{opt}}$ lies only on a few of the hyperplanes defined by $C$. For this reason, $x_{\text{opt}}$ could also have been obtained from a formulation in which one leaves out all of the non-binding constraints as long as they are not violated. This motivates an approach in which we use valid inequalities to iteratively strengthen a given linear programming relaxation with the aim of finding an (optimal) solution that violates none of the constraints including those that were left out.

More formally, such an approach starts with an integer linear program and solves its linear programming relaxation $\text{LP}_0$, obtaining a solution $x^0$ with objective value $z^0_{\text{LP}}$. As already discussed in the introduction, if $x^0$ is feasible for IP it is an optimal solution to IP. Otherwise, $z_{\text{IP}} < z^0_{\text{LP}},\ x^0 \notin \text{conv}(S)$ and there is a valid inequality $(\pi, \pi_0)$ that separates $x^0$ from $\text{conv}(S)$. Assume that we get such a valid inequality from a *separation oracle* and add it to the linear programming formulation to obtain a new relaxation $\text{LP}_1$. Solution $x^0$ is no longer feasible for $\text{LP}_1$, Therefore, by resolving the linear program we get a solution $x^1 \neq x^0$ and we have $z^1_{\text{LP}} \leq z^0_{\text{LP}}$. By iterating this approach we get a sequence of solutions and objective values that hopefully converges to an integer optimal solution. As a given optimal solution remains dual feasible after a valid inequality (also called a *cut* in this context) has been added it is usually advantageous to use the dual simplex algorithm to reoptimize. This approach is called *cutting-plane algorithm*. In theory, it can be shown that there are general purpose separation algorithms that always find violated valid inequalities as long as the iterative solution $x^i$ is fractional, such that the above cutting-plane algorithm is finite. The separation algorithms were developed by Gomory, Lovász, and Schrijver [64, 65, 115].

In practice, pure cutting plane algorithms are rarely used. The complexity of this approach is hidden in the complexity of the separation oracle. This is indeed problematic, as finding a separating hyperplane can be as hard as solving the whole problem. A famous theorem by Grötschel, Lovász and Schrijver [68] says that under mild technical assumptions on the description of the polyhedron there is a polynomial time reduction from the optimization problem of finding an optimal solution to an IP to the separation problem of finding a valid inequality $(\pi, \pi_0)$ that cuts off a given infeasible solution $x'$ from $\text{conv}(S)$. Symmetrically, there is a polynomial time reduction from the separation problem to the optimization problem.

Still, it is possible that for a given solution $\tilde{x}^i$ to $\text{LP}_i$ the separation problem is easy. In particular, there can be families of valid inequalities for which separation is a polynomial time algorithm or for which there are at least efficient heuristics that often find a separating hyperplane, when there is one. In this scenario, it often pays off to combine the available non-exact separation algorithms with a branch-and-bound approach as follows.

Run the cutting-plane algorithm as long as no integral feasible solution has been found and as long as the non-exact separation algorithm finds separating hyperplanes. Then *branch*, i.e., partition (a fixed subset of) the set of optimal integral feasible solutions into two parts, by imposing a condition that explicitly excludes the current infeasible solution $\tilde{x}^i$ and solve recursively the two subproblems, using the solutions obtained from the cutting-plane steps as bounds. A simple way to achieve a partition of the solution space is to take a fractional variable $\tilde{x}^i_j \in \tilde{x}^i$ and impose $x_j \leq \lfloor \tilde{x}^i_j \rfloor$ for the first subproblem and $x_j \geq \lceil \tilde{x}^i_j \rceil$ for the second subproblem.

As in classical branch and bound algorithms we can discard a subproblem if the objective value of the relaxation is smaller than a known integral feasible solution. Put simply, a branch-and-cut algorithm is nothing but a classical branch and bound algorithm, in which a cutting plane algorithm is used in every node of the branch and bound tree to strengthen the bound.

As for the cutting planes used, it is often easy to find some family of separating hyperplanes, but it is not always clear, how effective these are. Ideally, we would like to have only facets of conv$(S)$ as separating hyperplanes. Unfortunately, it is often very difficult to find these or to prove that some separating hyperplane is indeed facet-defining. For the case of the vehicle routing problem, a variant of which we will study in the next chapter, even the dimension of the vehicle-routing-polytope is unknown, which makes it difficult to prove that a given hyperplane is facet-defining. One reason is that the most prominent proof strategy is to exhibit dim(conv$(S)$) many affinely independent points in $S$ that lie on the hyperplane. Moreover, the associated separation problem could be too difficult to solve. In this context it makes more sense to have relative measures for the quality of valid inequalities (and trade them off against the difficulty of finding them).

**Definition 4.22** *If $\pi x \leq \pi_0$ and $\mu x \leq \mu_0$ are two valid inequalities for $P$, $(\pi, \pi_0)$* dominates *$(\mu, \mu_0)$ if there exists $u > 0$ such that $\pi \geq u\mu$ and $\pi_0 \leq u\mu_0$, and $(\pi, \pi_0) \neq (u\mu, u\mu_0)$.*

**Definition 4.23** *A valid inequality $(\pi, \pi_0)$ is* redundant *in the description of $P$, if there exist $k \geq 1$ valid inequalities $(\pi^i, \pi^i_0)$ and weights $u_i > 0$, for $i = 1, \ldots, k$ such that $(\sum_{i=1}^k u_i \pi^i, \sum_{i=1}^k u_i \pi_0)$ dominates $(\pi, \pi_0)$.*

Sometimes it is already remarkable that the inequalities of a family are supporting.

This was only a superficial description of branch-and-cut algorithms. There are several important aspects that need to be considered in an implementation: The generated cuts can be globally or only locally valid, i.e., only in the subtree of the current node in the branch and cut tree. The cuts are usually managed in *cut pools*. The branch and cut tree needs to be stored in an efficient data-structure that also depends on the policy of selecting the next node to expand. Connected to this is the choice of the *branching strategy* that selects how to branch in a given node. It is also not always clear when one should branch: It can happen that the cutting plane algorithm keeps finding violated cuts (and increasing the size of the linear program to solve) without significant improvement in the bound. For this reason, one needs to decide when to interrupt the cutting plane algorithm and branch. In connection with branch-and-cut also *pre- and postprocessing strategies* are discussed, because in practical applications they are equally important for the success of an implementation.

A complete description of all these components is out of the scope of this thesis. Some of these aspects are discussed in [1] and also in [82], where the authors explain how they implemented the SYMPHONY branch-and-cut framework, which we also used for our implementation. In [2] Aardal and van Hoesel discuss the underlying theory and exemplary families of valid inequalities for basic combinatorial optimization problems in more detail. Finally, [21] and [104] provide a wealth of references for branch-and-cut.

## 4.7 IP Column Generation

In the last section we reviewed the branch-and-cut approach. The basic idea of this technique is to leave rows out of the LP-relaxation, because there are too many of them to handle them efficiently and most of them will not be binding for the given objective function. The column generation approach applies the same idea to the columns (variables) of the formulation: Initially most of the columns are left out of the LP-relaxation because there are too many of them to handle them efficiently and most of them will be non-basic in an optimal solution anyway. From this short description it is clear that column generation can be the method of choice when there are many variables in the formulation. Following [13] there can be several reasons why such a formulation arises:

- A compact formulation of the problem may have a weak LP-relaxation. This relaxation can be tightened by a reformulation that involves a huge number of variables.
- A compact formulation of the problem may have a symmetric structure that causes branch-and-bound to perform poorly. A reformulation with a huge number of variables can eliminate the symmetry.
- Column generation provides a decomposition into master and subproblems (as we will see in the following). The subproblems can have a natural interpretation in the problem setting which allows a straight-forward incorporation of additional constraints into these subproblems.
- A formulation with a huge number of variables may be the only known choice.

One class of problems that is particularly well-suited for column generation consists of problems where the variable set can be partitioned into sets $x^1 \in \mathbb{Z}_+^{n_1}, \dots, x^K \in \mathbb{Z}_+^{n_K}$ such that any feasible solution has to fulfill two types of constraints. First, constraints that are defined on each of the variable sets alone, i.e., constraints of the form $x^k \in X^k = \{x \in \mathbb{Z}_+^{n_k} : D^k x \leq d_k\}$ for all $k = 1 \dots, K$ that are independent of each other. For example, in a vehicle routing setting this could be the constraints for the feasibility of a route of a single vehicle. Second, coupling constraints that involve all variable sets. In a vehicle routing setting this could be the constraints that define when a set of routes constitutes a feasible solution. Such a problem can be written as

$$z_{\text{IP}} = \max \left\{ \sum_{k=1}^{K} c^k x^k : \sum_{k=1}^{K} A^k x^k = b, x^k \in X^k, k = 1, \dots, K \right\} \qquad (\text{IP}_{\text{compact}})$$

Assuming that the sets $X^k$ are bounded, such a problem can be automatically transformed into an Integer Linear Program with many variables by the *Danzig-Wolfe reformulation*:

As the sets $X^k$ are bounded, they contain a finite number of points $\{x^{k,t}\}_{t=1}^{T_k}$ and can thus be represented as

$$\left\{ x^k \in \mathbb{R}^{n_k} : x^k = \sum_{t=1}^{T_k} \lambda_{k,t} x^{k,t}, \sum_{t=1}^{T_k} \lambda_{k,t} = 1, \lambda_{k,t} \in \{0,1\}, t = 1, \dots, T_k \right\} \tag{4.7.1}$$

Now we simply substitute this equation for $x^k$ and obtain the *IP Master problem*:

$$z_{\text{IP}} = \max \sum_{k=1}^{K} \sum_{t=1}^{T_k} (c^k x^{k,t}) \lambda_{k,t} \tag{IPM}$$

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} (A^k x^{k,t}) \lambda_{k,t} = b \tag{4.7.2}$$

$$\sum_{t=1}^{T_k} \lambda_{k,t} = 1 \text{ for } k = 1, \dots, K \tag{4.7.3}$$

$$\lambda_{k,t} \in \{0,1\}, \text{ for } t = 1, \dots, T_k \text{ and } k = 1, \dots, K \tag{4.7.4}$$

The column generation method applies to linear programs, so we first consider the linear relaxation of the above integer program, also called the *linear programming master problem* LPM with objective value $z_{\text{LPM}}$. Its formulation is identical to the integer program except that one demands $\lambda_{k,t} \geq 0$ instead of

$\lambda_{k,t} \in \{0,1\}$ ($\lambda_{k,t} \leq 1$ is redundant). The idea of column generation is now to mimic the behavior of the (revised) primal simplex algorithm—but instead of explicitly calculating the reduced prizes (pricing) of all nonbasic variables one solves an optimization problem (the *pricing problem*) and thereby implicitly prices the non-basic variables. To be more precise, the revised simplex algorithm must be adapted in the following way to implement a column generation approach:

Initially, one selects a set of columns (at least one for each $k$) such that the *Restricted Linear Programming Master* (RLPM) problem, i.e., the LP relaxation that comprises these columns only is feasible. In a typical application this is easy to obtain or can be enforced by adding columns with highly unfavorable cost that are infeasible to the subproblems but guarantee feasibility for the master.

In the next steps the simplex algorithm is mimicked by alternating LP solving and pricing steps. In step $i$ the current RLPM is solved using the primal simplex algorithm to obtain a primal optimal solution vector $\hat{x}^i$ and a dual optimal solution vector $u = (u^b, u^K)$, where $u^b$ represents the dual values of the constraints of type (4.7.2) and $u^K$ refers to (4.7.3). For each subproblem $k$ the dual vector $u$ defines reduced prices $c^k x - u^b(A^k x) - u^K_k$ for each $x \in X^k$ (c.f. (4.3.3)). By solving the following pricing problem

$$\overline{c}^k_{\max} = \max\left\{(c^k - u^b A^k)x - u^K_k : x \in X^k\right\} \tag{4.7.5}$$

one can mimic the simplex algorithm by adding the column defined by the maximum in (4.7.5) if $\overline{c}^k_{\max} > 0$. If $\overline{c}^k_{\max} \leq 0$ for all $1 \leq k \leq K$ the current solution $\hat{x}^i$ to RLMP is optimal for the whole problem and the algorithm terminates. A column generation algorithm iterates the pricing and resolving steps until the above termination criterion is met and an optimal solution to the linear relaxation is found. The correctness of this procedure follows from the correctness of the simplex algorithm.

Historically, column generation was developed for linear programs not all variables of which fit into main memory. Today, it is almost always used for integer linear programs in combination with a branch and bound approach, which together is then called *IP Column Generation* or *branch-and-price*.

One important reason for using column generation for Integer Programs is that the linear relaxation LPM is often stronger than a simple linear relaxation of IP$_{\text{compact}}$: It follows directly from the definition of Danzig Wolfe decomposition that the linear programming master attains the value

$$z_{\text{LPM}} = \max\left\{\sum_{k=1}^{K} c^k x^k : \sum A^k x^k = b, x^k \in \text{conv}(X^k) \text{ for } k = 1, \dots, K\right\} , \tag{4.7.6}$$

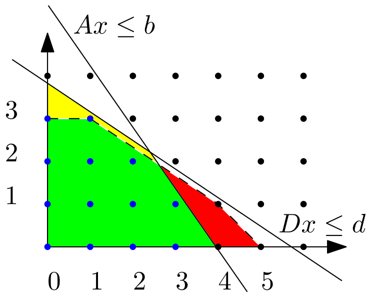


**Figure 4.7.1:** *Illustration of different relaxations of an Integer Linear Program with master $Ax \leq b$ and one single subproblem $Dx \leq d$. The blue points are feasible to the integer linear program. The yellow and the green area are the solution space of the LP relaxation. The green and the red area are the convex hull of the feasible points for $Dx \leq d$ and therefore the solution space of the subproblem. The green area is solution space of the linear relaxation of the master problem after a Danzig-Wolfe transformation or a Lagrangian Relaxation of $Dx \leq d$, adapted from [6].*

which must be greater or equal than the linear relaxation. In Figure 4.7.1 we see a simple example where the linear programming master can achieve tighter bounds than the linear relaxation. The value $z_{\text{LPM}}$ is exactly the value that one could have obtained with a branch-and-cut approach that has exact separation routines for the subproblems. This means that for a given solution it checks $x \in X^k$ for $k = 1, \ldots, K$ and adds a separating hyperplane to the formulation if one of the tests fails. Such an approach is called the *partial convexification relaxation*. Finally, also a Lagrangian relaxation (not discussed here), in which the master constraints are dualized and the subproblems are solved separately, attains exactly the value $z_{\text{LPM}}$. The equality of the objective values of these three approaches does not mean that they are equivalent. Most importantly, during the solution process the column generation approach has a primal feasible solution available, which holds neither for the branch-and-cut approach nor for the Lagrangian Relaxation. Also the separation routine of the branch-and-cut problem is in general a different algorithmic problem than the subproblems of the column generation approach. Finally, it follows directly from the finiteness of the simplex algorithm with the right selection rules that column generation terminates after a finite number of iterations, which is not always the case for the partial convexification approach.

Similar to the branch-and-cut section, in this section we have only seen a short review of column generation. There are numerous details and implementation issues that have been glossed over, we mention a few of them: Typically ordinary branching schemes that branch on single variables perform very poorly in a column generation setting so that *special branching schemes* are necessary, and the same holds for primal heuristics. Also the column selection is non-trivial. There is a trade-off between pricing in many columns at once in order to decrease the number of simplex iterations at the cost of additional computations in the pricers and a bigger formulation of the master. It seems to not even be settled, what the most effective pricing rule is, i.e., which column among those with negative cost should be included into the next RLPM formulation? A typical problem of column generation algorithms are oscillating dual values. Several schemes exist to remedy this problem. Finally, pre- and postprocessing plays an important role in column generation algorithms and also in the literature. For information on these aspects and more see [13, 126, 45, 46, 117].

# Chapter 5

# Optimizing a Hub and Spoke Railway System

Loaded like a freight train.
Flyin' like an aeroplane.
Feelin' like a space brain.
One more time tonight.
(Guns N' Roses - Nighttrain)

## 5.1 Introduction

In the present chapter everything revolves around a particular freight train system: The Cargo-Express service [112] of Swiss federal railways SBB Cargo Ltd., which provides fast overnight transportation of goods. The Cargo-Express network is operated as a (multi-) hub spoke system, the hubs being the shunting-yards.

The whole system is complex enough to pose a multitude of challenging optimization problems, a few of which we will study here. The main problem we tackle is to route the freight trains through the network and to find a cheap schedule for them that respects constraints given by the real-world problem. For the cost of the schedule, we consider the cost of operating the engines and the cost of the traveled distance.

The outline of this chapter is as follows: First we discuss how the Cargo-Express service works in detail, then we present three models: First a naive ILP approach; then an approach that decomposes the routing and scheduling problem and uses the branch-and-cut method; finally we describe an alternative method that is based on column generation, considers more constraints and jointly optimizes the routing and the scheduling problem. After having discussed the models and their application to the real world problem we discuss some algorithmic questions of more theoretical flavor which are related to the shunting and scheduling aspects. At the end of the chapter we present experimental results and present related work. (Although we usually give the related work and the summary of results in the introduction of a chapter, it is more natural here to postpone it, because a big part of the chapter is concerned with building the models which constitute a prerequisite for the understanding of related work). It is clear that the principal results of this section also apply to any other hub-spoke system that is operated similarly. However, in this chapter we will stick closely to the SBB Cargo Express system: First, it is a motivating running example, second and more importantly, we have had access to all the necessary data to run our optimization codes on it and could discuss the quality of our solutions with the SBB Cargo planners.

## 5.2 Problem Description

We explain how our reference hub-spoke system, the SBB Cargo Express Service works. We describe the way of operation from the point of view of a customer who wants to transport some containers or a set of freight cars from a source station in Switzerland to a destination station somewhere else in Switzerland. More precisely, the Cargo Express System is meant for customers who need such transports regularly: A typical customer wants to have a fixed amount of goods transported from station A to station B every week-day, or every Monday in the next half year. The customer has to announce in advance her fixed demand for a given period. In general, this period corresponds to the lifetime of the fixed schedule that is currently generated once every year taking into consideration the announced demands. The schedule is constructed by hand and slightly adapted in a “trial-period” after its implementation. If the demands change over a year or new customers want to be served, the SBB Cargo team generally succeeds in adapting the existing schedule to the new situation. This is also done by hand.

The transport itself works as follows. In the evening, the customer deposits her cars at the requested station. She can do this until a negotiated departure time that can also vary among different customers for the same station. After that time the cars are picked up by a freight train which transports them together with other cars to one of the shunting yards (to which we usually refer as hubs). Along its route a freight train picks up cars at different stations. The process of picking up the cars incurs a non negligible amount of time, the *couple time at the stations*, which is mainly needed for a brake test and is therefore relatively independent of the number of picked up cars. After it has arrived at a hub, a train is decoupled. The shunting is performed on humps. The cars of the customer are coupled to an outgoing train that departs roughly after all its cars have arrived from incoming trains and have been coupled to the outgoing train. This outgoing train then delivers the cars at the destination. It can also happen that the train goes to another hub where shunting takes place again. Trains that commute between hubs neither deliver nor pick up cars. A last possibility is that the cars of a customer are transported by a dedicated train directly to their destination without going through a hub. In one night each engine can perform exactly one of these tasks, i.e., going to and from a hub once including potentially a few rides between hubs, or transporting a shipment directly to its destination.

We next discuss some subtleties of the problem. The capacity of the shunting yards is limited: Only a limited number of cars can be stored there and only a limited number of shunting operations can be performed in a given time period. The trains themselves have a limited capacity. The network may contain *switch-backs*, i.e., a crossing or a furcation, which have the effect that the time to go through this switchback to a given destination depends on the direction a train comes from. Furthermore, there might be stations at which two or three trains are decoupled and coupled. Thus, such a station acts as a sort of shunting yard without hump, but in general only very few trains are shunted there. There is also the issue of track-availability, although this is not crucial during the night. Finally, also the engine drivers have to be assigned to the trains and transported to and from the trains in a way that is subject to various regulations.

In 2005 and 2006 the system was operated as a single-hub system: Two close-by shunting yards Däniken and Olten were used as a single hub. Starting from 2007 the system will be operated as a multi-hub system with more than one hub. One reason for the change was the insufficient hub capacity at Däniken and Olten. Also the bigger Cargo-Rail system that does not guarantee delivery in one night comprises several hubs and works similarly on a larger scale. The models developed in this chapter are therefore also applicable to this system (except for Model 1 below, which assumes a single hub).

The primary goal of our optimization efforts has been to make the system cheaper. The main costs are the costs of operating the engines and the costs of driving the necessary routes on the tracks. From the above description it is clear that it can be important to minimize the necessary hub-capacity.

## 5.3 Models

We present a sequence of models that mirror our attempts to solve the optimization problem sketched above.

### 5.3.1 Scope of the Models

It should be clear from the above description that a complete model of the SBB Cargo system is very difficult to formulate. For this reason, we decided to ignore some of the seemingly secondary aspects in our models: We do not consider the problem of engine driver assignment. We do not consider the problem of switchbacks and furcations. It is rather straight-forward however to construct gadgets that transform the network in such a way that switchback and furcations are taken care of at the cost of an additional blow-up of the network. For the models we will ignore the exact sequence of shunting operations at the hub and then have closer look at these in Section 5.7.

In the next section we discuss the aspects of the problem that we model. The three mathematical models that we present afterwards will partly only consider a subset of these aspects.

### 5.3.2 Common Notation

We assume that we are given a *(railway) network* $N = (V, E, \ell, c)$. The nodes represent stations, hubs and junctions, the edges represent the tracks connecting those; $\ell : E \to \mathbb{R}^+$ is the length function on the edges and $c : E \to \mathbb{R}^+$ is the cost function on the edges. By $H \subset V$ we denote the set of hubs in the network. To simplify the presentation we will write $c(e)$ mostly as $c_e$. In the following we give a list of parameters and features that we consider in our models.

- The set $S$ of *shipments* defines the cars to be transported. Associated with a shipment $s \in S$ are the following properties:
  - source$(s)$ the *source station*,
  - dest$(s)$ the *destination station*,
  - depart$_\mathrm{S}(s)$ the *earliest possible departure time* at station source$(s)$,
  - arrive$_\mathrm{S}(s)$ the *latest possible arrival time* at station dest$(s)$,
  - vol$(s)$ the *volume*, i.e., the number of cars of $s$.

- The *maximum train load* $L_{\max}$ bounds the total volume that any engine can take.

- The *shunting time at the hub* $T^h_{\text{shunt}}$ is the minimum additional time that a departing train has to wait due to shunting after its last shipment has arrived at hub $h \in H$. This time is assumed to be independent of the number and volume of the shipments.

- The *couple time at the stations* $T^s_{\text{couple}}$ is the additional time incurred by taking any set of shipments at a station. This time is independent of the number and volume of the shipments taken.

- The *hub capacity* $\text{cap}_h$ for each hub $h \in H$ specifies how many cars can at most stay in the hub at any given moment in time.

- The engine cost $C_{\text{engine}}$ represents the cost of operating one engine.

- The average speed $\bar{v}$ is used to calculate the time it takes the trains to travel on the tracks.

Sometimes it is more useful to represent the source, destination and volume information of the shipments in an $(n \times n)$ *supply-and-demand* matrix $M$, such that $M(\text{source}(s), \text{dest}(s)) = \text{vol}(s)$ for all $s \in S$.

One part of a solution is a *route* of an engine through the network. By route we mean a graph theoretic walk, which can in particular contain repeated edges and nodes. Some of our solution approaches will restrict the walks to (elementary) paths. This restriction is not admissible for our problem at hand, however it can be circumvented in our solution approaches by a transformation of the original sparse network to a network on the complete graph. Another part of a solution is a specification of arrival and departure times for each of the nodes on the routes. We now define more formally the most general version of the railway problem that we want to study.

**Definition 5.1 (General Train Optimization Problem (GTOP))** *Given a railway network $N = (V, E, \ell, c)$, a set of hubs $H \subset V$ a set of shipments $S$ and the parameters $L_{\max}$, $T^h_{\mathit{shunt}}$, $T^s_{\mathit{couple}}$, $\mathit{cap}_h$, $C_{\mathit{engine}}$, $\bar{v}$ as defined above, find a feasible solution of minimum cost. A feasible solution consists of the size $k$ of the necessary train fleet and time consistent routes for $k$ engines with time consistent arrival and departure times at the stations and hubs such that all shipments are transported from their source to their destination respecting the time windows and the other constraints given by the above parameters.*

GTOP is obviously strongly NP-hard as it contains problems like the traveling salesman, bin-packing, and diverse scheduling problems.

We discuss three models. The first one, Model 0, is a more or less straightforward translation of the problem into an integer linear program. In Model 1 the problem is decomposed and branch-and-cut is applied. We developed this model at a time, when the SBB Cargo Express System had a single hub, so that it only applies to single-hub systems. Model 2 is a column generation approach for the multi-hub case.

The railway network for the SBB Cargo Express Service has 651 nodes and 1488 edges. Around 200 shipments are transported every day. In a preprocessing phase we could condense the network to a network with 121 nodes and 332 edges.

## 5.4 Model 0

From a practical point of view, when dealing with NP-hard problems, it can be a good idea to try to formulate the whole problem as an ILP and to solve this on a realistic instance to get an idea of how "difficult", in a fuzzy sense, the problem is. Therefore, we experimented with different ILP formulations. As one could expect from the number of constraints mentioned, the formulations get rather lengthy. In Appendix A.2 we present one that models all of GTOPs constraints except for the hub capacity. We give the model in the ILOG OPL modeling language [71] with comments. To get a rough idea of this model we consider the sets of variables only:

```
// train uses arc on its way to some hub
var bool travelsForth[Trains,Arcs];
// train uses arc on its way from some hub
var bool travelsBack[Trains,Arcs];
// train goes between two hubs
var bool travelsBetween[Trains, Hubs, Hubs];
// train starts at node
var bool starts[Trains,Nodes];
// train ends at node
var bool ends[Trains,Nodes];

// time at which a train arrives at a station on its way to some hub
var departTimes arrivesForth[Trains, Nodes];
// time at which a train arrives at a station on its way from some hub
var arriveTimes arrivesBack[Trains, Nodes];
// time at which train z starts a hub hub ride
var betweenTimes startsBetween[Trains];

// direct paths from Shipments
var bool direct[Trains, Shipments];

// second train depends on first for its front/back journey through h
var bool depFB[Trains,Trains, Hubs];
```

```
// second train depends on first for Hub Hub journey through h
var bool depFH[Trains,Trains, Hubs];
// second train depends on first for hub back journey through h
var bool depHB[Trains,Trains, Hubs];

// train takes shipment and goes to hub
var bool takesForth[Trains, Shipments, Hubs];
// train takes shipment from hub
var bool takesBack[Trains, Shipments, Hubs];
// train takes shipment between hubs
var bool takesBetween[Trains, Shipments, Hubs, Hubs];
```

In [122] Toth and Vigo discuss different formulations for the related but simpler vehicle routing problem (VRP). In terms of their classification the first sets of variables `travelsForth[]` to `ends` can be seen as the variable set of a *three-index vehicle flow formulation* for the VRP. This model needs to be augmented by variables for the time windows `departTimes` to `betweenTimes` and by a variable `direct` that models direct trains for shipments (that do not go via a hub). Until this point the model is a relatively "standard" model. The main complicating effect, which is not covered by standard formulations, comes from the hubs: An outgoing train can only depart after its incoming shipments have arrived. This creates dependencies between incoming and outgoing trains that we model by the `dep` variables. For these variables to be computable we also need to know which train takes which shipments (note that this is not implied by the `travels` variables as many trains can pass a station and pick up shipments there. We first ran the model on the sparse network which entails the elementariness of all routes as an unwanted side effect. It would have been possible to find also non-elementary routes by a transformation similar to the one given in Section 5.5.3. By further complicating the model it would have also been possible to model the hub capacity constraints. However, we performed some computational experiments with this model, which made it seem unlikely that a model of this type could ever yield a result for our real instances. We created two toy instances on a graph with 14 nodes and 24 edges, one with four and one with 11 shipments. On Machine B, see A.1, the small instance needed a quarter of an hour to solve. We ran the 11 shipments instance for more than 40 hours and did not even get a feasible solution.

## 5.5 Model 1

The computational experience with Model 0 and similar models motivated us to decompose GTOP into a routing part and a scheduling part, each of which should be solvable in reasonable time. Moreover, Model 1 is formulated for a single hub because at the time when it was developed the SBB Cargo Express network had a single hub only. In the following we explain the decomposition.

### 5.5.1 Routing

In the routing part we search for short routes from the stations to the hub and vice versa but do not compute the arrival and departure times of the routes. We treat the transport from the stations to the hub and the transport from the hub to the stations separately. Obviously, these two problems are symmetric, therefore it is sufficient to analyze only one direction. Here we mainly consider the transport from the stations to the hub. The decomposition entails that the time windows are ignored in the selection of routes. For this reason, we have to guarantee in a different way that the routes do not get too long (even if also the load limit and the objective function tend to keep routes short). To this end, we introduce a global maximum trip distance $D_{\max}$ that limits the length of all routes. The model implies that we do not use the whole information of the supply and demand matrix $M$ but rather the row and column totals (depending on the direction) in this matrix. This is expressed by a supply (demand) value that is associated with each node. This value is the volume (in freight cars) that is to be transported from the station to the hub, or vice versa. From this simplification alone, the number of shipments in the instance drops considerably. The following definition gives a formalization of the problem.

**Definition 5.2 (Train Routing Problem with fixed train fleet (TRP))** *Assume we are given a network $N^{TRP} = (V, E, \ell, c)$, a specified single hub node $\hat{h} \in V$, a set of shipments $S$, a maximum train load $L_{\max}$, a maximum trip distance $D_{\max}$, the average speed $\bar{v}$, the couple time $T^s_{couple}$, and the fleet size $K$. A feasible solution $\sigma = (R^x, R^y, \rho^x, \rho^y)$ consists of two sets of $K$* routes *each, the $X$*-routes *$R^x = \{r^x_1, \ldots, r^x_K\}$ and the $Y$-routes $R^y = \{r^y_1, \ldots, r^y_K\}$, i.e. graph-theoretic walks in the network having one endpoint in $\hat{h}$, and an association of each shipment $s \in S$ in the network with one route $\rho^x(s)$ in $R^x$ and one route $\rho^y(s)$ in $R^y$ such that the following properties hold:*

1. *No route is longer than $D_{\max}$. The* length *of an $X$- route $r^x$ is defined as the length of the route plus $|\{v \in V \mid \rho^x(v) = r\}| \cdot T^s_{couple} \cdot \bar{v}$. The length of a $Y$ route is defined accordingly.*

2. *No train is loaded more than $L_{\max}$. The* load *of a train for an $X$-route $r^x$ is $\sum_{s:\rho^x(s)=r}$ vol$(s)$, and accordingly for a $Y$-route.*

3. *For each shipment $s$ route $\rho^x(s)$ visits source$(s)$.*

4. *For each shipment $s$ route $\rho^y(s)$ visits dest$(s)$.*

*The cost of a solution is the sum of the lengths of the routes. The train routing problem is to find a minimum cost solution.*

The restriction to a fixed fleet size $K$ is not very limiting. In practice, one wants to minimize a weighted sum of the number of used trains and the traveled distance. Reasonable values for the number of used trains are usually in a very small interval so that the optimization can be done for all of these values. Finally, it is also possible to choose different fleet sizes for the $X$- and the $Y$-routes.

### 5.5.2 Scheduling

For the scheduling problem we assume that we are already given a solution $(R^x, R^y, \rho^x, \rho^y)$ to TRP. For the routes in $R^x$ and $R^y$ it remains to specify the exact arrival and departure times at the station that respect the time windows, the hub shunting time, and also keep the hub-capacity low. In fact, we choose the hub capacity as the objective function. A complete schedule contains the departure and arrival times of each train at each station. However, it is not necessary to specify a schedule in such detail: The arrival time and departure time windows are one-sided in the sense that there is a priori no latest departure time or an earliest arrival time for the shipments; for that reason it does not make sense for a train to slow down on the tracks or to wait before a station until it is "possible" to enter it. Therefore, we can completely specify a schedule by giving the arrival and departure times of the trains at the hub and assume w.l.o.g. that the engines arrive there and start from there traveling the route in a fastest possible way, i.e., going at speed $\bar{v}$ and only waiting the required couple time $T^s_{\text{couple}}$ at the stations where they pick-up or deliver shipments.

**Definition 5.3 (Train Shunting and Scheduling Problem (TSSP))** *Let a solution $(R^x, R^y, \rho^x, \rho^y)$ to TRP for a set $S$ of shipments be given. A feasible solution to TSSP defines for each $r^x \in R^x$ an arrival time at the hub arrive$_{\hat{h}}(r^x)$ and for each $r^y \in R^y$ a departure time at the hub dep$_{\hat{h}}(r^y)$ such that*

- *the (inferred) arrival and departure times at the stations respect the time windows.*
- *an outgoing train $\tau$ only departs after all incoming trains carrying cars for $\tau$ have arrived and have been shunted:*

$$\forall r^x \in R^x\ \forall r^y \in R^y\ \forall s \in S:$$

$$\rho^x(s) = r^x\ \wedge\ \rho^y(s) = r^y\ \Rightarrow\ \mathit{dep}_{\hat{h}}(r^y) \geq \mathit{arrive}_{\hat{h}}(r^x) + T^{\hat{h}}_{\mathit{shunt}} \quad (5.5.1)$$

*The cost of a solution $\sigma$ equals the maximum number of cars that are in the hub at the same time:*

$$\mathit{cost}(\sigma) = \max_{t \in \mathit{EVENTS}} \sum_{s:\mathit{arrive}_{\hat{h}}(\rho^x(s)) \leq t \,\wedge\, \mathit{dep}_{\hat{h}}(\rho^y(s)) \geq t} \mathit{vol}(s) \ . \tag{5.5.2}$$

*where* $\mathit{EVENTS} = \{t' \in \mathbb{R} \mid \exists r^x \in R^x : \mathit{arrive}_{\hat{h}}(r^x) = t' \vee \exists r^y \in R^y : \mathit{dep}_{\hat{h}}(r^y) = t'\}$*. An optimal solution to TSSP is one with minimum cost.*

We will consider the theoretical and practical aspects of this scheduling problem later in Section 5.7. In the following we focus on a branch and cut solution approach to the TRP.

To sum up, the above decomposition of GTOP into the sequential solution of TRP and TSSP allowed us to produce solutions to a real world instance with one hub as documented in the experimental Section 5.8. Observe also that the sequential approach might find no feasible solution for GTOP even if there is one, because the optimal solution to TRP can be an infeasible instance to TSSP. We also address this problem in the experimental section.

The TRP can also be seen as a problem in its own right: It is a sensible variation of a vehicle routing type problem. In the next section we will illuminate the connections to classical vehicle routing problems.

### 5.5.3 Branch and Cut Approach

In general, there are many different possibilities to tackle a problem like the train routing problem, even after one has decided to use an exact approach like branch and cut. Out of these possibilities we could only evaluate a small subset. Natural candidates are those based on formulations that proved successful for related vehicle routing problems, i.e., *vehicle flow* models with the classical two- or three index formulations, *commodity flow* models or *set partition* models, see [122] for an excellent overview. After some preliminary experiments with such ILP models for TRP we decided to use a two-index formulation. In particular, we based our implementation on an existing VRP package for the vehicle routing problem that is based on the two index formulation. We extended it such that it can solve our problem. In the following we discuss the distance constrained capacitated vehicle routing problem and its connection to TRP.

**Definition 5.4 (Distance Constrained Vehicle Routing Problem (DCVRP))**
*The input consists of a network $N^{\mathit{DCVRP}} = (V, E, c, \ell)$, a specified hub node $\hat{h} \in V$, furthermore demands $d_i, i \in V$ on the nodes, a maximum load (or*

*capacity)* $L^{\text{DCVRP}}_{\max}$*, and a maximum distance* $D^{\text{DCVRP}}_{\max}$*. All edges are present in the network* $N$*.*

*Find* $K$ *elementary circuits with minimum total cost, such that the following properties hold.*

1. *Each circuit visits the hub node* $\hat{h}$*.*
2. *Each customer node is visited by exactly one circuit.*
3. *The sum of the demands on each circuit does not exceed the allowed load* $L^{\text{DCVRP}}_{\max}$*.*
4. *The length of each circuit does not exceed* $D^{\text{DCVRP}}_{\max}$*.*

*The cost of a path equals the sum of the edge costs.*

### DCVRP ILP-Formulation

The two index formulation of the DCVRP uses Boolean variables $x_e$ to indicate if a given edge $e \in E$ is chosen. We give it for complete undirected graphs, following [123] and explain it below.

$$\begin{aligned}
\textbf{DCVRP:}\quad \min \quad & \sum_{e \in E} c_e x_e \\
\text{s.t.} \quad & \sum_{e=\{i,j\} \in E} x_e = 2 && \forall i \in V \setminus \{\hat{h}\} && \text{(5.5.3a)} \\
& \sum_{e=\{\hat{h},j\} \in E} x_e = 2K && && \text{(5.5.3b)} \\
& \sum_{e=\{i,j\} \in E, i \in Q, j \notin Q} x_e \geq 2r(Q) && \forall Q \subset V \setminus \{\hat{h}\}, Q \neq \emptyset && \text{(5.5.3c)} \\
& x_e \in \{0,1\} && \forall e \in E && \text{(5.5.3d)}
\end{aligned}$$

Equations (5.5.3a) enforce that each node except for the hub has degree two, (5.5.3b) enforces that the hub has degree $2K$.

Equations (5.5.3c), the *capacity cut constraints*, are the most interesting constraints. They play a similar role for the VRP as the subtour elimination constraints do for the TSP [87]. The left hand side, evaluated at a solution vector, gives the number of edges in that solution that cross the cut $[Q, V \setminus Q]$. Note that every vehicle that serves customers in $Q$ contributes two to the number of edges of the cut. The right hand side should therefore represent the minimum number of necessary crossings of vehicles due to the connectivity requirement, capacity reasons, and the distance constraints. The value $r(Q)$ can be understood as the maximum of two values: $d(Q)$, which accounts for the maximum distance constraints; and $\lambda(Q)$, which accounts for the capacity constraints (and also for the connectivity constraints).

There are several valid but not equivalent choices for a definition of $d(Q)$ and $\lambda(Q)$. In fact, there is a whole hierarchy of possible values for $\lambda(Q)$ that lead to different families of valid inequalities with nondecreasing right-hand side, so that the higher families of inequalities dominate (see Definition 4.22) the lower families but also lead to increasingly difficult separation problems. The simplest choice is $\lambda(Q) = \frac{\sum_{v \in Q} d_v}{L_{\max}}$, which leads to a separation problem for which there is a polynomial time algorithm but still gives (together with a valid choice for $d(Q)$) a valid formulation for DCVRP, see [16, 10]. The generated *fractional capacity inequalities* are in general not supporting for the DCVRP-polytope. As all solutions have integral $x_e$ values it is obviously admissible to choose $\lambda(Q) = \left\lceil \frac{\sum_{v \in Q} d_v}{L_{\max}} \right\rceil$ which gives the *rounded capacity inequalities* that dominate the simple capacity inequalities. However, the associated separation problem is $NP$-complete [96], which also implies that even computing the LP-relaxation of a classical VRP with rounded capacity inequalities is an NP-complete problem. An even tighter formulation could be obtained by solving the associated bin-packing problem. But even this approach does not necessarily lead to supporting inequalities, see [96] for a more detailed discussion.

The value $d(Q)$ is the minimum value $k \in \mathbb{N}$ such that the objective value $v^k_{\text{TSP}}$ of a $k$-TSP problem on $Q$ divided by $D^{\text{DCVRP}}_{\max}$ and rounded up equals $k$, see [123]:

$$d(Q) = \min \left\{ k \in \mathbb{N} \middle| k \geq \left\lceil \frac{k\text{-TSP}(Q)}{D^{\text{DCVRP}}_{\max}} \right\rceil \right\} . \tag{5.5.4}$$

### Adapting the Model

Although not identical, the train routing problem bears many similarities to DCVRP. In the following, we give a transformation such that the optimal solution of any TRP instance $I_{\text{TRP}}$ can be derived from the optimal solution of the corresponding transformed DCVRP instance $\Psi(I_{\text{TRP}})$. This approach allows us to use existing software packages extended with code for the distance constraints.

An optimal solution to an instance $I_{\text{TRP}}$ of TRP consists of two independent parts $(R^x, \rho^x)$ and $(R^y, \rho^y)$. In the following we describe w.l.o.g. how to transform $I_{\text{TRP}}$ in order to obtain $(R^x, \rho^x)$. Roughly the task of the transformation is to do the following: Translate a problem defined on a sparse graph for which the solution consists of a set of circuits covering the network to a problem on the complete graph for which the solution consists of a set of paths covering the network. Moreover, we have to correctly translate the length and capacity constraints. Note that the common transformation $c_{\{i,j\}} \longleftarrow c_{\{i,j\}} - c_{\{\hat{h},i\}} - c_{\{\hat{h},j\}}$ by Clarke and Wright savings [31] only works in the other direction, in the sense that it transforms a problem with circuits into a problem with paths.

The transformation $\Psi$ applies the following types of modifications to instance $I_{\text{TRP}}$ to achieve the above goals:

1. Add all missing edges to $N^{\text{TRP}}$. The length of such a new edge $e = (u, v)$ is set to the length of the shortest $u, v$ path in $N^{\text{TRP}}$.

2. Add $T^s_{\text{couple}} \cdot \bar{v}$ to the weight of each edge of the network.

3. (optional) Partly merge shipments with identical source that will definitely be transported by the same train.

4. Replace stations with $j$ shipments, $j > 1$, by a $j$-clique with zero cost and length edges. Identify each shipment with this source with one of the nodes of the clique by setting the $d.$ values of the clique nodes to the vol values of these shipments.

5. Put a gadget on top of the network as explained below.

After Step 4 there is a specific *shipment node* for each shipment. Figure 5.5.1 shows how the TRP-instance after the first three modifications, represented by the circular nodes is transformed into a DCVRP instance by adding extra nodes and edges. The extra nodes are shown as rectangular nodes. The underlying idea of the gadget is to allow each vehicle to "jump" from the hub to a start node in the network. To that purpose, $K$ extra nodes $\{v^e_1, \ldots, v^e_K\}$ are added to the network. These nodes are all connected to the hub $\hat{h}$ with edges of length and cost $-M$, with $M > \sum_{e \in E} c_e$. The extra nodes are connected to the rest of the network via the complete bipartite graph. The length of each such edge is zero. The extra nodes are not interconnected. Each extra node has an associated demand of $M'$, with $M' > \sum_{s \in S} \text{vol}(s)$. Moreover, we set the load limit $L^{\text{DCVRP}}_{\max}$ of the vehicles in the DCVRP instance to $M' + L_{\max}$, and finally we set $D^{\text{DCVRP}}_{\max} = D_{\max} - M$. The idea behind the $-M$ edges is to force them into the solution. The shipment of very high weights on the incident extra nodes enforces that each train can visit at most one of these extra nodes. Together these two modifications enforce that each route "jumps" exactly once to its starting node and goes back to the

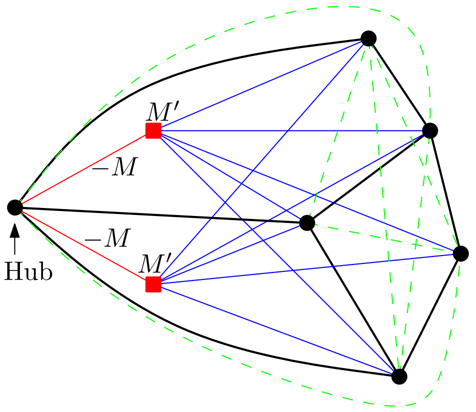


**Figure 5.5.1:** *Transformation from TRP to DCVRP for two trains. The original graph consists of all circular nodes together with the solid black edges. First the green dotted edges are introduced to make the graph complete. Then all edge length are increased by $T^s_{couple} \cdot \bar{v}$ and nodes with multiple shipments are expanded to cliques (not shown). Then the two new red square nodes with demand $M'$ are added and connected via the (red) edges of length $-M$ to the hub. The blue edges have length zero and make possible a "jump" to the starting node of a route.*

hub node from there. The correctness of transformation $\psi$ is established in the following lemma.

**Lemma 5.5** *Let a TRP instance $I_{TRP}$ be given. Let $\sigma_{DCVRP}$ be an optimal solution to the DCVRP instance $\Psi(I_{TRP})$ of cost $c$. Then, the $X$-part $\sigma^x_{TRP} = (R^x, \rho^x)$ of the optimal TRP solution has cost $c + K \cdot M$ and can be reconstructed from $\sigma_{DCVRP}$ in linear time. The same statement holds for the $Y$-part of the solution.*

**Proof.** We first argue that $\sigma_{\text{DCVRP}}$ has the following form: It consists of $K$ cycles, such that the $i$-th cycle can be written as $C_i = (\hat{h}, v^e_i, v_{i,j_1}, \ldots, v_{i,j_{l_i}}, \hat{h})$. The reason for this is that the supply of $M'$ of each extra node together with the capacity constraints enforce that exactly one extra node $v^e_i$ is on each circuit $C_i$. The negative lengths of the edges $(\hat{h}, v^e_i)$ enforce that the extra nodes must be directly after (or before) $\hat{h}$ on the circuits because one such negative cost edge is taken per circuit.

To construct the $i$-th route of $\sigma_{\text{TRP}}$ we first set $\rho^x(s) = i$ for all shipments in $C_i$. Then we set the $i$-th route to $(v_{i,j_1}, \ldots, v_{i,j_{l_i}}, \hat{h})$, replacing nodes that arose from an expansion to an $i$-clique by the original station node and deleting consecutive occurrences of a node. We then reconstruct the original paths by replacing edges that do not exist in $N^{\text{TRP}}$ by the shortest path in $N^{\text{TRP}}$ between the two end nodes.

From the description of the transformation it follows that the feasibility of $\sigma_{\text{DCVRP}}$ guarantees that $\sigma^x_{\text{TRP}}$ is feasible w.r.t. $L_{\max}$ and $D_{\max}$ and the covering of the shipments. Note that we assume a couple time also at the stations where trains start their journey. As for the optimality, assume $\sigma^x_{\text{TRP}}$ is not an optimal solution. Then let $S'$ be the $X$-part of the optimal solution to the TRP consisting of $K$ routes. This solution can be transformed into $K$ cycles by reverting the above construction. It is straight-forward to check that these cycles form a cheaper feasible solution for the DCVRP instance. $\square$

Note that the bipartite component of the gadget can be slimmed down: All we need is that there is a perfect matching between each subset $Q \subset V$ of size $K$ and the extra nodes. Thus, for a subset of $K$ nodes in $N$, it is sufficient to insert only the edges $(v_i, v^e_i), i \in \{1, \ldots, K\}$ in the bipartite component. Also the extra edges that are introduced to make the graph complete can be partly removed: Each edge between two shipment nodes, for which the pick up of both would already lead to a violation of the capacity or the distance constraint need not be introduced.

## Separation Heuristics

The core part of every branch and cut algorithm is the design of a separation algorithm that effectively separates a given fractional point from the convex hull of integer solutions. The general separation problem is NP-complete and this still holds for most known classes of valid inequalities including the rounded capacity inequalities. For this reason, we focus on effective *separation heuristics* that try to find violated inequalities of Type (5.5.3c) but do not guarantee to find one if one exists. As the cutting plane generation is embedded into a branch and bound framework, this does not compromise the correctness of the algorithm. These inequalities comprise two subtypes, capacity and distance constraints.

We have based our implementation on the branch and cut code by Ralphs et al. [107] for the vehicle routing problem. This has the advantage that we could use the already implemented separation heuristics for the capacity constraints. Ralphs et al. report that most classes of valid inequalities for the vehicle routing problem that have been explored in the literature although theoretically interesting prove to be either ineffective or very difficult to compute in practice. Therefore, they focus on rather simple separation heuristics for the capacity constraint, see [107] for a more detailed description. The capacity constraints being handled by the existing code we focus on the new distance constraints for our separation heuristics.

As we are only interested in instances that arise from a transformation $\psi$, we describe how to find the cuts in the graph $N_3 = (V_3, E_3)$ which is the original network of the TRP-instance after the fourth step of the transformation, i.e., before the gadget is added. In this graph an integral solution consists of a set of paths that start from the hub node $\hat{h}$. Given a fractional solution $\hat{x}$ we consider the *support graph* $\hat{N}_3 = (V_3, \hat{E}_3), \hat{E}_3 = \{e \in E_3 \mid \hat{x}_e > 0\}$. If we temporarily remove the hub node $\hat{h}$ a support graph decomposes into a set of connected components $\{Q_1, \dots, Q_{k'}\}$, $k' \leq K$. We define the *length* $\ell(Q)$ of such a connected component to be the sum of the lengths of the edges in $Q \cup \{\hat{h}\}$ weighted by $\hat{x}$. Furthermore, we define the *engine number* $e(Q)$ of $Q$ to be the number of engines that enter $Q$ in $\hat{N}_3$. The value $e(Q)$ is defined as $\lfloor \frac{c}{2} \rfloor + 1$, where $c$ is the value of the (weighted, graph theoretic) $[Q, \{\hat{h}\}]$-cut in $\hat{N}_3$.

If for a connected component $Q$ the value $\ell(Q)/e(Q)$ exceeds $D_{\max}^{\text{DCVRP}} - M = D_{\max}$, we introduce a valid inequality of Type (5.5.3c) for $Q$. Setting $\lambda(Q)$ to $e(Q)+1$ gives a locally valid cut. Next, we apply the shrinking heuristic described in [107] to enforce stronger cuts if the previous search was unsuccessful. All these cuts have only local validity in the branch-and-cut search tree, because $\hat{x}$ depends on the branching decisions that enforce or forbid some edges.

For integral solutions we make more efforts to come up with cuts. If the length of a path in such a solution exceeds $D_{\max}$, we introduce a cut with right-hand-side $2\kappa(Q)$. The value $\kappa(Q)$ is a global lower bound on the number of vehicles needed to serve $Q$, and is explained below. We try to enforce stronger cuts by considering only parts of each path: we sequentially add edges along a path until the distance-constraint is violated, and enforce a cut on this smaller subset of nodes. Next, we shorten the path from its source, and add a cut for each subset violating the distance constraint. This procedure is iterated by adding one edge to the previous prefix of the path. As before, these cuts use $\kappa(Q)$ as lower bound on the number of vehicles. Note that if the right hand side of the cut does not change after a shrinking step the new cut dominates the old one in the sense of Definition 4.22.

The value $\kappa(Q)$ is computed independently of $\hat{x}$ by adapting two standard relaxations of the TSP to our needs, the relaxation to 1-trees and the one to the assignment problem. For a node set $Q$, we compute the minimum weight spanning tree on $Q$. If the cost of the spanning tree exceeds $D_{\max}$, the set of nodes of that component represents a cut. In order to find the best possible lower bound on $\lambda(Q)$, we proceed as follows. Let $\ell_T$ be the weight of the tree, $\lambda = 1$. As long as $\frac{\ell_T}{D_{\max}} > \lambda$, we increase $\lambda$ by one, decrease $\ell_T$ by the weight of the heaviest edge in the tree and increase it by the weight of the cheapest not yet considered edge from $Q$ to the hub node $\hat{h}$. The idea is to subdivide the component in many components, each served by one vehicle. The updated value of $\ell_T$ provides a lower bound on the length of the route needed to serve these new components. Hence, the final value of $\lambda$ is a global lower bound on the number of vehicles needed to serve the nodes in $Q$.

If this procedure does not lead to $\lambda > 1$, we apply a second heuristic based on the TSP-relaxation to the assignment problem, see [87]. We build a bipartite graph as follows: Each partition $A$ and $B$ consists of the nodes in $Q$. For every original edge $(u, v)$ we introduce two edges $(u_A, v_B)$ and $(u_B, v_A)$ of cost $c_{\{u,v\}}$. Furthermore, we introduce one new node for each partition. This node represents the hub node. We connect the hub node of partition $A$ to all nodes in $B$ except for the hub node with edges of weight as in the original graph. These edges represent the trip from the last node to the hub. Similarly, we connect the hub node of partition $B$ to all nodes in $A$ (excluding the hub) with edges of weight zero. These edges represent a zero cost edge from the hub to the start of the path. The weight of the minimum weight bipartite matching is a lower bound on the length of the minimum path needed to serve the nodes in $Q$. Hence, if the weight of the bipartite matching exceeds $D_{\max}$, at least two vehicles are needed to serve the nodes in $Q$. In our implementation we use the LEDA assignment algorithm [92]. The value $\kappa(Q)$ results from the best result of the two relaxations.

To sum up, the transformation $\psi$ together with the additional cuts presented in this section allowed us to use standard vehicle routing software to produce solutions to the TRP problem. The results are presented in the experimental Section 5.8. We present our solution approach to the TSSP problem later in Section 5.7.

## 5.6 Model 2

Model 1 has some limitations: First of all, the decomposition approach implies that the solution process for the TRP is blind to the constraints of the TSSP so that the overall solution can perform badly w.r.t. the "secondary" optimization criterion, the hub-capacity. For the same reason, the approach has problems with tight time windows for which many of the TRP solutions will induce infeasible instances for TSSP. A further problem with Model 1 is that it is not designed to handle multiple hubs and trains that go between hubs.

Because of the above reasons we decided to tackle the general train optimization problem from a different angle by using a column generation approach that naturally decomposes the problem into a master and a pricing problem (as discussed in Section 4.7). The work on this column generation approach is not entirely finished at the time of writing this thesis, but we can present some preliminary results.

### 5.6.1 Master Problem

For the presentation of the decomposition approach we employ the terminology introduced in Section 4.7. The master problem can be formulated directly (instead of obtaining it from a Danzig Wolfe decomposition from a compact formulation). We associate variables with routes. Informally, a route $r$ is a walk in the railway network to or from a hub, for which we specify which shipments it takes and when it reaches the nodes of the walk. The routes we consider must be feasible in the sense that the arrival and departure times at the stations are consistent with the track length, the couple time and the average speed; the arrival times at pick up/delivery nodes respect the time windows of the shipments; and finally the maximum train load is respected. For a shipment $s$ and a route $r$ we write $s \in r$ if $r$ picks up or delivers $s$. The cost $c_r$ of any route is defined as the sum of the cost of the edges in it and a contribution to $C_{\text{engine}}$, as explained below. The volume $\text{vol}(r)$ is the total volume of shipments picked up (or delivered): $\text{vol}(r) = \sum_{s \in r} \text{vol}(s)$. We denote by $\text{start}(r)$ and $\text{end}(r)$ the start and end node of $r$. We give a more formal definition of a route later in the pricing section.

For the description of the master problem the current notion of a route is precise enough.

For the master problem, a route to the hub is encapsulated in a Boolean variable $x_r^t$, where $t$ stands for the arrival time at the hub. We discretize time at the hub so that the number of these variables is finite. Let $T$ be the set of points in time and $t \in T$. Similarly, the Boolean variable $y_r^t$ encodes whether route $r$ is taken starting at time $t$ from the hub. We also allow paths between the hubs which we represent as Boolean variables $h_r^{\mathrm{d}:t}$. Here $r$ simply stands for the source and destination hub and the set of shipments taken. Time $t$ stands for the departure time at the source hub. Let $\delta$ be the time it takes to travel between the hubs. Then, for a variable $h_r^{\mathrm{d}:t}$ we write equivalently $h_r^{\mathrm{a}:t+\delta}$, i.e., we specify its arrival time instead of its departure time. Following the policy of SBB Cargo we do not consider pick-ups or deliveries on hub paths. Finally, we introduce for each shipment $s \in S$ Boolean variables $d_s$ that model the possibility of transporting $s$ directly with a dedicated engine from its source to its destination. Such a path is not associated to any time since it must always be possible to deliver a shipment on the direct path respecting the time windows. Therefore, the $d_s$-variables guarantee that an initial reduced master problem that contains all such $d_s$ variables is feasible. Abusing notation slightly, we denote by $R$ the set of all routes, irrespective of their type; furthermore, all summations are meant to be over feasible routes separately for the summands. This means that a sum of type $\sum_{r \in R, t \in T} x_r^t + h_r^{a:t}$ is to be read as the sum of all $X$-variables that correspond to feasible $X$-routes with arbitrary arrival time at any hub plus the sum of all $Y$-variables that correspond to feasible $Y$-routes with arbitrary departure time from any hub. All sums of this type in the constraints should be seen as a shorthand notation for two separate sums.

The pickup and delivery time windows and the capacity constraint being handled in the pricing subproblems, the master problem is a set partitioning model that has to ensure that $X$-routes and $Y$-routes that depend on each other are time consistent, that the capacity at the hub is not exceeded and that the two commodities, engines and cars are used consistently. In comparison to a “typical”-set partitioning problem our master problem is quite complicated. We use the following formulation for the master problem.

$$
\begin{array}{llll}
\min & \displaystyle\sum_{r\in R, t\in T} c_r x_r^t + c_r y_r^t + c_r h_r^{\text{d:}t} + \sum_{s\in S} c_s d_s & & \\
(\hat{\pi}_s) & d_s + \displaystyle\sum_{t\in T, r\in R: s\in r} x_r^t & \geq\ 1 & \forall s \in S \\
(\check{\pi}_s) & d_s + \displaystyle\sum_{t\in T, r\in R: s\in r} y_r^t & \geq\ 1 & \forall s \in S \\
(\phi_{sh}) & \displaystyle\sum_{t\in T, \text{end}(r)=h, s\in r} x_r^t + h_r^{\text{a:}t} & & \\
 & - \displaystyle\sum_{t\in T, \text{start}(r)=h, s\in r} y_r^t + h_r^{\text{d:}t} & =\ 0 & \forall s\in S, h\in H \\
(\hat{\sigma}_{hst}) & \displaystyle\sum_{\substack{r: s\in r, \text{end}(r)=h \\ t_1\geq t}} x_r^{t_1} + h_r^{\text{a:}t_1} & & \\
 & - \displaystyle\sum_{\substack{r: s\in r, \text{start}(r)=h \\ t_2\geq t + T^h_{\text{shunt}}}} y_r^{t_2} + h_r^{\text{d:}t_2} & \leq\ 0 & \forall t\in T, h\in H, s\in S \\
\Bigg[\,(\check{\sigma}_{hst}) & \displaystyle\sum_{\substack{r: s\in r, \text{start}(r)=h \\ t_1\leq t}} y_r^{t_1} + h_r^{\text{d:}t_1} & & \\
 & - \displaystyle\sum_{\substack{r: s\in r, \text{end}(r)=h \\ t_2\leq t - T^h_{\text{shunt}}}} x_r^{t_2} + h_r^{\text{a:}t_2} & \leq\ 0 & \forall t\in T, h\in H, s\in S\,\Bigg] \\
(\chi_{th}) & \displaystyle\sum_{\substack{t_1\leq t \\ \text{end}(r)=h}} \text{vol}(r)\, x_r^{t_1} + \text{vol}(r)\, h_r^{\text{a:}t_1} & & \\
 & - \displaystyle\sum_{\substack{t_2\leq t \\ \text{start}(r)=h}} \text{vol}(r)\, y_r^{t_2} + \text{vol}(r)\, h_r^{\text{d:}t_2} & \leq\ \text{cap}_h & \forall t\in T, h\in H \\
(\beta_{ht}) & \displaystyle\sum_{\substack{r\in R, \text{end}(r)=h \\ t'\leq t}} x_r^{t'} + h_r^{\text{a:}t'} & & \\
 & - \displaystyle\sum_{\substack{r\in R, \text{start}(r)=h \\ t'\leq t + T^h_{\text{shunt}}}} y_r^{t'} + h_r^{\text{d:}t'} & \geq\ 0 & \forall t\in T, h\in H \\
 & x_r^t, y_r^t, h_r^{\cdot:t}, d_s \in \{0,1\} & & \forall r\in R, t\in T, s\in S
\end{array}
$$

Let us discuss the model. To keep things simple, we refer to the constraints by the name of the associated dual variables. The $\pi$-constraints are the set covering part of the model that state that each shipment has to be picked up and delivered. We chose a set covering model instead of a set partitioning model (with equality constraints) because in this way the LPs tend to be easier to solve. Furthermore, the $\pi$ dual variables are restricted in sign, which leads to an easier cost structure for the pricing problems. See [126] for a discussion of the advantages of set covering over set partitioning models.

The $\phi$-constraints are global inflow-outflow constraints for the shipments. Together with the $\sigma$-constraints they ensure the time consistent inflow-outflow of shipments. The $\hat{\sigma}$-constraints enforce that a shipment that arrives after time $t$ has a corresponding outgoing train after time $t + T^h_{\text{shunt}}$. The constraint is designed in such a way that also for fractional routes of a shipment $s$ each fractional incoming route has a corresponding fractional outgoing route. The $\check{\sigma}$-constraints represent the symmetric statement for outgoing trains. In fact, these constraints are nothing else than a variation of the classical *generalized flow conservation constraints for networks with intermediate storage* for flows over time problems, see for example [69]. Out of the three types of constraints $\check{\sigma}$, $\hat{\sigma}$, and $\phi$ every pair of types implies the third type. Therefore, it suffices to include the $\hat{\sigma}$- and the $\phi$-constraints in the formulation.

The $\beta$-constraints play a similar role for the engines as the $\sigma$-constraints do for the shipments. Here, we allow that engines stay in the hub. This can lead to complications if we charge the engine costs uniformly to $X$- and $Y$-paths. One way to solve this is to charge all engine costs to $X$-paths and to connect an artificial node with a zero-length track to the hub, from which all "superfluous" engines can start.

The $\chi$-constraints limit the capacity for each hub $h \in H$ to $\text{cap}_h$ cars in each time slot.

There is one subtlety that we do not discuss here: Shipments with a hub as a source or destination complicate the model. For this special case, constraints of type $\pi, \phi, \sigma$ and $\chi$ have to be adapted. It is more or less straight-forward to do this, but the modifications[1] make the model unnecessarily hard to read. We do however take care of these cases in our implementation.

[1]Note that it does not suffice to connect extra stations by zero-length tracks to the hubs. Such a construction would affect the $\beta$-constraints. In particular, we would potentially incur the engine costs for the additional trips to the extra nodes.

## 5.6.2 Pricing

As discussed in Section 4.7, the pricing problem consists of finding a feasible route with negative reduced cost. From the above formulation it is clear that we have separate pricing problems for the different types of variables. The easiest case are the direct paths: The reduced cost of a direct path associated with variable $d_s$ is

$$C_{\text{engine}} + \sum_{e \in r} c_e - (\hat{\pi}_s + \check{\pi}_s) \quad , \tag{5.6.1}$$

where $r$ is the route associated with the direct path for shipment $s$. As there are only $|S|$ direct paths it makes sense to include all of them in the initial reduced master problem and to keep them in the formulation. This has the side effect that the initial RLPM is always feasible, just as all other RLPMs if we keep the direct paths variables in the formulation.

More interesting are the $X$- and $Y$-routes. The reduced cost of an $X$-route $r$ to hub $h'$ at time $t'$ is

$$\begin{aligned} C_{\text{engine}} + \sum_{e \in r} c_e - \sum_{s \in r} (\hat{\pi}_s + \phi_{sh'}) - \sum_{s \in r, t \leq t'} \hat{\sigma}_{h'st} \\ \left[ + \sum_{s \in r, t \geq t' + T^h_{\text{shunt}}} \check{\sigma}_{h'st} \right] - \text{vol}(r) \sum_{t \geq t'} \chi_{th'} - \sum_{t \geq t'} \beta_{h't} \end{aligned} \tag{5.6.2}$$

where

$$\hat{\pi}_s, \check{\pi}_s, \beta_{ht} \geq 0 \text{ and } \chi_{th}, \hat{\sigma}_{hst}, \check{\sigma}_{hst} \leq 0 \text{ and } \phi_{sh} \in \mathbb{R} \quad . \tag{5.6.3}$$

This cost structure has the attractive property that for fixed $t'$ and $h'$ all costs are either constants or can be charged to edges in the network or to the pick-up of shipments. This motivates the definition of the following problem.

**Definition 5.6 ($X$-pricing)** *Given a network $N$, a hub $h' \in H$, an arrival time $t'$, parameters $L_{\max}$, $T^s_{couple}$, $\bar{v}$, a set of shipments $S$, and a* reduced cost *rc*$(s)$ *for each shipment $s \in S$. Find a minimum cost walk $w$ in $N$ from hub $h'$ to an arbitrary node and a set of shipments $S_r \subseteq S$, such that*

1. *for all $s \in S_r$ source$(s) \in w$,*
2. *$\sum_{s \in S_r}$ vol$(s) \leq L_{\max}$,*
3. *$t'$ – time$_w\big(h', \text{source}(s)\big) \geq depart_S(s)$, where time$_w\big(h', \text{source}(s)\big)$ denotes the time it takes an engine to go on the tracks in $w$ from $h'$ to the first occurrence of source$(s)$ according to the parameters $\bar{v}$, $\ell$, and $T^s_{couple}$.*

*The cost of a walk is the sum of $C_{engine}$, the edge costs of $w$ and the reduced costs of the shipments in $S_r$.*

A route $r$ is now more precisely defined as such a pair $(w, S_r)$. This problem can be understood as a variation of a resource constrained shortest path problem with time windows. It is non-elementary in the sense that nodes can be visited more than once but it is elementary in the sense that shipments can be picked up only once. Also note that even though the edge costs are nonnegative, the reduced costs of the shipments can be negative. Therefore, the problem is NP-complete even without the load limit and the time windows by an easy reduction from the shortest (elementary) path problem without nonnegativity restriction: It suffices to replace each negative cost edge by a cost zero path consisting of two edges such that the new node in the middle gets a single shipment, the reduced cost of which equals the length of the edge in the shortest path problem.

We propose to solve the $X$-pricing problem by a particular label correcting shortest path algorithm augmented with some extra information to guarantee shipment elementariness. We do not give the details of the algorithm here because it is a relatively straight-forward extension of existing label correcting algorithms. The main framework is similar to the ones in [17, 57]. The main design decision is to store the following information in a label: reduced cost, primal cost, time, capacity and the shipments that have been picked up.

As illustrated in [17, 57], the most important ingredients for an efficient label correcting algorithm are efficient *dominance rules*. In short, a dominance rule gives criteria as to when a given label $\lambda_1$ dominates a label $\lambda_2$ in the sense that $\lambda_2$ can be removed from the current list of active labels because for any feasible solution that arises from $\lambda_2$ there is a corresponding feasible solution arising from $\lambda_1$ that has an objective value that is at least as good. We developed several dominance rules that take into consideration the capacity constraint, the current objective value and the time windows. It is also possible to develop dominance rules for complete time slots. A simple such rule is as follows. Let hub $h'$ be fixed. If $t_2 > t_1$ and for all $s \in S$ it holds that $\sum_{t \leq t_2} \hat{\sigma}_{h'st} \geq \sum_{t \leq t_1} \hat{\sigma}_{h'st}$, i.e., $\sum_{t_1 < t \leq t_2} \hat{\sigma}_{h'st} = 0$ and $\sum_{t_1 \leq t < t_2} \beta_{h't} = 0$ we know that the complete time slot $t_1$ is dominated by $t_2$ in the sense that an optimal solution for $t_2$ cannot be worse than an optimal solution for $t_1$.

In our implementation we solve one pricing problem for each pair of hub and point in time. A more involved approach could solve the pricing problem for all time slots in one application of a label-correcting algorithm. In that case one could also apply dominance rules for labels from different time slots.

The reduced cost of a $Y$-route $r$ from hub $h'$ at time $t'$ is:

$$\sum_{e\in r} c_e - \sum_{s\in r}(\check{\pi}_s - \phi_{sh'}) + \sum_{s\in r, t\le t'-T^h_{\text{shunt}}} \hat{\sigma}_{h'st} \\ \left[+\sum_{s\in r, t\ge t'} \check{\sigma}_{h'st}\right] + \text{vol}(r)\sum_{t\ge t'} \chi_{th'} + \sum_{t\ge t'-T^h_{\text{shunt}}} \beta_{h't} \tag{5.6.4}$$

The $Y$-pricing problem is completely symmetric to the $X$-pricing problem. Therefore, we do not discuss it here.

The reduced cost of an $H$-route $r$ from hub $h'$ to hub $h''$ with departure time $t'$ and arrival time $t''$ is

$$\text{cost}(h', h'') + \sum_{s\in r}(\phi_{sh'} - \phi_{sh''}) - \sum_{s\in r, t\le t''} \hat{\sigma}_{h''st} + \sum_{s\in r, t\le t'-T^h_{\text{shunt}}} \hat{\sigma}_{h'st} \\ \left[-\sum_{t\ge t'} \check{\sigma}_{h'st} + \sum_{s\in r, t\ge t''+T^h_{\text{shunt}}} \check{\sigma}_{h''st}\right] \\ -\text{vol}(r)\sum_{t\ge t''} \chi_{th''} + \text{vol}(r)\sum_{t\ge t'} \chi_{th'} \\ -\sum_{t\ge t''} \beta_{h''t} + \sum_{t\ge t'-T^h_{\text{shunt}}} \beta_{h't} \;. \tag{5.6.5}$$

Here cost$(h', h'')$ denotes the fixed cost for the trip between the hubs $h'$ and $h''$. Also for the $H$-route it holds that all reduced costs are either constants for fixed $t'$, $h'$, and $h''$ or can be charged to the pick-up of shipments. As the walk through the graph for an $H$-route is always the direct connection between the two involved hubs, the $H$-pricing problem simplifies to a (comparatively) simple knapsack problem. To solve it we use the algorithm (and the code) by Pisinger [102], which we adapt to handle fractional profits as described in Ceselli and Righini [23].

## 5.6.3 Acceleration Techniques

The literature on Column Generation abounds with techniques to accelerate CG-algorithms. It seems that the number of acceleration techniques that one can apply to a single problem is limited more by the willingness to implement, test, and evaluate them than by the number of different such techniques. In [45] Desaulniers, Desrosiers and Solomon propose a whole catalogue of techniques that have proved helpful in various applications. Similar to the style of their paper we briefly discuss the techniques that we use.

Pre-Processing Strategies. As already mentioned above, we condensed the original SBB-Cargo network to a smaller network: To this end, we first calculate the all-pairs shortest path among the nodes with shipments and the hubs. Edges that do not occur on any such shortest path can be safely ignored. In the resulting graph we contract degree-two nodes if they are neither a hub nor a source nor a destination of any shipment.

Aggregation. A commonly used technique for routing problems is to aggregate demands. This technique is of limited use in our case, because there are no shipments with identical source and destination. It is however possible to aggregate shipments with identical source only in the $X$-pricing and shipments with identical destination only in the $Y$-pricing. This leads to a moderate speed-up of the pricing steps.

Heuristic Pricing. As explained in Section 4.7 it suffices for the correctness of the algorithm that the pricing problem returns any column with negative reduced cost if there is one. For this reason, it is beneficial to use pricing heuristics or to stop a pricing calculation prematurely if a column with negative reduced cost has already been found. We apply both techniques by managing a *column pool* that contains candidates for negative reduced cost columns. Before the pricers are called the column pool is checked for negative reduced cost columns. Additionally, we stop the label-correcting algorithm if a large enough set of negative reduced cost columns has already been found.

Perturbation. For the calculation of the reduced master problem we perturb the right hand side of the $\sigma$-constraints by small random values. As predicted in the literature [44] this leads to a significant speed-up of the LP-solving steps.

Column Elimination. In order to keep the reduced master problem at a reasonable size we subject the columns to aging. If a column keeps being nonbasic for a given number of pricing iterations it is removed from the RLPM and added to the column pool. This technique gives a speed-up for a well-chosen threshold value. However, it trades off the speed that it takes to solve a single RLPM versus the number of iterations it takes to solve the whole problem and can therefore even be detrimental to the solving process.

Stabilization. A common problem in column generation is that the dual variables oscillate and assume extreme values (if compared to the dual values at the optimal solution). This behavior can be partly explained by the property of the simplex algorithm that it finds an extreme point of the optimal face. Different schemes to remedy this problem have been introduced. We experimented with the one by Rousseau, Gendreau and Feillet [110] that sets up a sequence of linear programs with random objective function to find an interior point of the optimal face. Our preliminary experience with this approach is that it indeed stabilizes the dual values and that the objective value seems to converge faster. Unfortunately, this better convergence comes at the cost of highly increased solution times for the additional linear programs that need to be solved.

### 5.6.4 Heuristics

We experimented with a few heuristics. The currently most successful one is a variation of the simple *dive-and-fix* heuristic [133, 110] augmented with column generation steps and some problem specific rules. The heuristic iterates the following steps until an integral solution has been found.

1. Select a candidate column $c$ with the highest fractional value such that $c$ is time-consistent with the so far rounded up columns and round up $c$.
2. Round down all columns in the RLPM that are not time-consistent with $c$.
3. Stop the pricer for $c$'s column type from generating columns that include shipments of $c$.
4. Resolve the restricted master problem with the new bounds using the dual simplex algorithm.
5. Do several pricing and resolving iterations to include new columns into the RLPM.
6. If the solution is integral stop. Otherwise go back to Step 1.

Additionally, after each fixing step we execute a local search heuristic that tries to complete the current partial integral solution with other fractional columns from the RLPM (which it also shifts in time) and newly generated columns.

Our heuristic is unconventional if compared to other heuristics in column generation settings that rely more on the compact formulation or metaheuristics that are initialized and guided by the column generation process, see [41]. In our case we have a master problem that is “unusually complicated”, in which a feasible fractional solution already guarantees that all fractional $X$-routes have a compatible counterpart $Y$-route, i.e., the fractional solutions are already time consistent and do not exceed the hub capacity. This property would be lost if we tried to build our own routes out of the information in a compact formulation.

On the other hand, a different problem arises that is usually not a topic in Column Generation: By rounding up and down fractional routes it can happen that the RLPM becomes infeasible. In this case we apply a technique called Farkas Pricing, which has not been applied in column generation so far to the best of our knowledge. It is introduced in the SCIP library by Achterberg [3], which we use in our implementation. We briefly explain it in the next section.

### 5.6.5 Farkas Pricing

Consider a restricted linear programming master problem in general form that is infeasible.

$$\begin{aligned} \min cx \\ b \leq Ax \leq d \\ e \leq x \leq f \end{aligned} \tag{RLPM$_{\text{gen}}$}$$

As in the proof of the Farkas’ Lemma 4.5 we can set the objective function to 0 and consider the dual linear program

$$\begin{aligned} \max u_b b - u_d d + r_e e - r_f f \\ u_b A - u_d A + r_e - r_f = 0 \\ u_b, u_d, r_e, r_f \geq 0 \ \ . \end{aligned}$$

This linear program is feasible, which is certified by $(u_b, u_d, r_e, r_f) = 0$ and must therefore be unbounded, as the primal problem is infeasible. From the complementary slackness conditions it is clear that out of the two bounds associated with each constraint and each variable only one can be nonzero if these are different. If they are the same, still only one needs to be nonzero. Therefore, we can set $u = u_b - u_d$ and $r = r_e - r_f$. Then we have the following set of (in)equalities that certifies the primal infeasibility.

$$\begin{aligned} u_b b - u_d d + r_e e - r_f f > 0 \\ uA + r = 0 \ \ . \end{aligned}$$

As we have only a restricted linear programming master problem this does not imply that the underlying problem is infeasible. The aim of Farkas pricing is to add further variables such that the resulting RLPM is feasible again. An addition of a variable corresponds to the addition of a further column to $A$. In our case we have all the nonbasic variables at their lower bound $e = 0 < f$. Therefore, it follows from complementary slackness that $r_f = 0$ and $r_e \geq 0$, so that $r \geq 0$. Suppose we find a new variable corresponding to a column $a_i$ such that $-ua_i < 0$. Then for the infeasibility certificate above to carry on we need that $r_i = -ua_i < 0$ which is a contradiction to $r_i \geq 0$ and thus destroys this infeasibility certificate. This does not imply that the new RLPM is feasible. Still, it is clear from the finiteness of the number of variables that this procedure must find a primal feasible solution in a finite number of steps if the primal is indeed feasible. To get a column $a_i$ with $-ua_i < 0$ we call the same pricing algorithm as before except that we set the objective function to 0. The resulting reduced costs $0-ua_i$ are exactly what is needed here. The SCIP library is designed in such a way that it automatically switches to Farkas pricing if an RLMP becomes infeasible.

### 5.6.6 Branching

One issue not discussed so far is the question of *branching rules* and selection rules for *branching candidates*. In column generation one often applies branching rules that substantially differ from branching rules for classical branch and bound. In the classical approach, one chooses a fractional Boolean variable, fixes it to one in the first subproblem and to zero in the second. While fixing a variable to one can be sensible in a CG setting, fixing it to zero might incur problems in the pricing steps because it could be difficult to stop the pricer from regenerating a rounded down column. Moreover, such a step is not very meaningful when most variables have values close to zero. Branching rules for column generation usually compute from a fractional solution the values of variables in a compact formulation and branch on these. For example, we implemented a branching rule that first computes the fractional assignments of shipments to the hubs (the resulting value can be seen as a variable of a compact formulation) and then branches on this fractional assignment. Finally, all branching rules that are used for column generation for the vehicle routing problem can be used in our setting, see [37, 75] for details. As with our current implementation the solution of the root node of the branch and price tree takes a long time, we only did a few preliminary experiments with different branching rules. Instead, we focussed on obtaining integral solutions already in the root node. Our experimental findings with Model 2 are summarized in Section 5.8.

## 5.7 Shunting and Scheduling

In this section we take a closer look at the shunting and scheduling problems that arise in our problem context: the TSSP, a simplified variant of it and finally a problem connected to the optimization of the precise sequence of shunting operations in the shunting yard. The simplified version of TSSP can be formulated as follows. Given a solution to the TRP problem as sets of incoming and outgoing trains, decide in which sequence the trains should be scheduled to arrive and depart, such that the capacity of the shunting-yard at the hub is not exceeded. This means that we consider the sequencing variation of TSSP. We show that this problem is NP-complete, even in a very restricted setting. Then we discuss how to solve the TSSP problem by an ILP-formulation. Finally, we consider the problem of optimally grouping the shunting operations in the shunting yard.

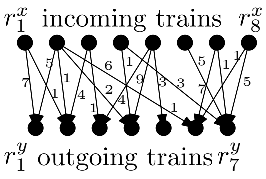
### 5.7.1 Hardness of Directed Minimum Cut Linear Arrangement

The sequencing problem for incoming and outgoing trains turns out to be NP-hard, already in a very simple version.

**Corollary 5.7 (of Theorem 5.9)** *It is NP-hard to decide if a collection of incoming and outgoing trains can be sequenced such that the capacity of the shunting yard is sufficient, even if every incoming train consists of precisely 3 cars, and every outgoing train of precisely 2 cars.*

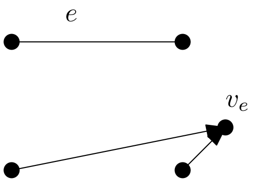
Given the composition of the incoming trains $R^x = \{r_1^x, \ldots, r_m^x\}$ and the outgoing trains $R^y = \{r_1^y, \ldots, r_n^y\}$, the sequencing task at hand can be depicted by the bipartite graph $G_{\text{io}} = (U \cup V, E)$ in Figure 5.7.1(a), the *in-out graph*. The incoming trains correspond to nodes in $U$, the outgoing trains to nodes in $V$. Each edge $e = (r_i^x, r_j^y)$ has a *volume* vol$(e)$ that corresponds to the number of cars that train $r_j^y$ receives from train $r_i^x$. Observe that this value is well-defined as each shipment is picked-up and delivered by exactly one train as specified by the $\rho$-functions in a TRP solution. We model precedence constraints by directed edges, for every car from its arriving train to its departing train, expressing that a car needs to arrive (with its train) before it can depart. We call $G_{\text{io}}$ a *uniformly directed bipartite graph*, because all edges are directed from $U$ to $V$. Alternatively to the volume information on the edges we can also restrict ourselves to in-out graphs with unit weight edges and allow parallel edges to model the volume.

The sequencing task corresponds to finding a *linear arrangement* of the graph $G$, i.e., an embedding of the graph onto the horizontal line, such that all edges

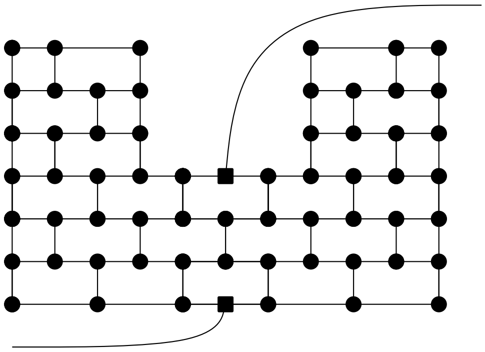


$r_1^x$ incoming trains $r_8^x$
$r_1^y$ outgoing trains $r_7^y$

(a) in-out graph $G_{\text{io}}$



(b) the transformation of an undirected edge to a pair of directed edges to a new node



(c) a small "U-wall" of [90, 94]. The square nodes connect to other square nodes of U-walls

**Figure 5.7.1:** *Illustrations for Minimum Cut Linear Arrangement*

are directed from left to right. For such an arrangement, the maximal number of edges crossing any vertical line is the *(cut-) width*, and it corresponds to the maximal number of cars residing in the shunting-yard. The width of a graph $G$ is given by the minimal width of a linear arrangement of $G$. This means that it is not necessary to consider the extra shunting time $T^H_{\text{shunt}}$ here because it represents a constant offset for the departure times that has no effect on the sequencing problem. Conversely, any uniformly directed bipartite graph can be understood as an in-out graph. Hence Corollary 5.7 follows indeed from Theorem 5.9 below.

Let $L \colon U \cup V \to \{1, \ldots, n\}$ be an optimal linear arrangement of the uniformly directed bipartite graph $G = (U \cup V, E)$. We can assume that in $L$ every outgoing train departs as early as possible that is, as soon as all its cars are available. Conversely, there is no use in scheduling an incoming train to arrive before some of its cars are needed. Together this means that given the sequence of the incoming trains it is easy to compute an optimal sequence of the outgoing trains, and vice versa.

Without the directions and the restriction to bipartite graphs, this problem is known as the “minimum cut linear arrangement”, a well studied NP-complete problem [62, GT44] that was shown to remain NP-hard for graphs of degree 3 [90], and even planar graphs of degree 3 [94]. We extend these results in the following way.

**Lemma 5.8** *For any constant $c > 0$ it is NP-hard to approximate minimum cut linear arrangement with an additive error of $c$, even on planar graphs with degree 3.*

**Proof.** By reduction from the NP-hard problem “minimum cut linear arrangement for planar graphs” [94]. We follow closely the reduction presented in [94]. Let $G$ be a planar graph, and $\ell$ the bound on its width. We construct $G'$ by taking a U-wall (see Figure 5.7.1(c)) of nodes with degree 3 for every node of $G$. $G'$ has the property that no two U-walls can significantly overlap in any linear arrangement. (This idea goes back to [90].) The edges of $G$ are replaced by edges in $G'$ connecting nodes of the inner parts of the two U-walls, the square nodes in Figure 5.7.1(c). As limit $L$ for the width of $G'$ we use the cut-width of the U-walls (which equals their height) plus the bound $\ell$ on the width of $G$. Now from any linear arrangement that obeys this limit $L$ we can reconstruct an arrangement of the original graph $G$ that has width $\ell$. To extend the result in the sense of the lemma, we “multiply” the construction by a factor $c$, i.e., we use $c$-times bigger U-walls, and replace every original edge by $c$ new edges. If there is a linear arrangement of the original graph of width $\ell$, the constructed graph $G'$ has width $cL$. Conversely, even from an arrangement of the new graph of width

$cL + c - 1$, we can reconstruct a linear arrangement because the U-walls still cannot overlap significantly, and this linear arrangement has width $c\ell + c - 1$. Because every original edge is represented by $c$ parallel edges, every cut is divisible by $c$, and hence this linear arrangement actually has width $c\ell$, hence the original graph $G$ has width $\ell$. □

**Theorem 5.9** *It is NP-hard to decide if a uniformly directed bipartite planar graph of out-degree 3 and in-degree 2 admits a linear arrangement of width $\ell$.*

**Proof.** By reduction from the problem of approximating the width of a planar graph with an additive error of 7 of Lemma 5.8. Let $G$ be the undirected planar graph and $L$ be the width limit defining an instance of that problem. Then $G$ either has width $\leq L$ or $\geq L + 7$, and it is NP-hard to distinguish these two cases. We construct a graph $G'$ by replacing every edge $e$ with a pair of edges directed toward a new node $v_e$, see Figure 5.7.1(b). This graph $G'$ is also known as the node-edge incidence graph with the links directed from nodes to edges (or vice-versa, this is just symmetric). We set the width limit $\ell = L + 6$.

Any optimal linear arrangement of $G'$ will place all the edge-nodes $v_e$ as far left as possible, because not doing so can only increase the width. The nodes of $G$ are also nodes of $G'$, such that the above observation allows us to directly map arrangements of $G$ to arrangements of $G'$ and vice versa. Then directly to the left of an original node $v$, the width of $G'$ is the same as the width of $G$. Only to the right of it, it is increased by twice the number of neighbors of $v$ in $G$ that are arranged left of $v$. By construction $v$ has at most 3 neighbors in $G$. For a neighbor $u$ of $v$ in $G$ that is arranged left of $v$, the directed edge $(u, v_e)$ continues up to $v_e$, and there is the additional edge $(v, v_e)$.

Concluding we see that if $G$ has width $\leq L$, then $G'$ has a linear arrangement of width $\leq \ell = L + 6$, but if the width of $G$ is $\geq L + 7 > \ell$, then $G'$ has width $\geq L + 7 > \ell$. □

This hardness result is complemented by the following consideration.

**Theorem 5.10** *Every uniformly directed bipartite graph with maximum degree 2 admits a linear arrangement of width 4, and it takes linear time to determine the minimal width of such a graph.*

**Proof.** A graph of maximum degree 2 decomposes into cycles and paths. A single edge has width 1, two directed edges have width 2, a path has width 3, and a cycle has width 4 (consider the last incoming train, it adds 2 cars to a shunting-yard containing two cars). □

### 5.7.2 Solving the TSSP Problem in Practice

As the instances of the TSSP problem that arise in our setting are not too large, we can solve them by a simple ILP formulation.

For this formulation we discretize the time horizon into $\tau$ points in time $T = \{0, \dots, \tau - 1\}$ and we make use of the in-out graph $G_{\text{io}}$ as defined in Section 5.7.1. We introduce Boolean variables $a_r^t$ and $d_{r'}^t$ that model arrival (and departure) of the trains $r \in R^x$ ($r' \in R^y$, respectively) at times $t \in T$. Here, we assume that the shunting time $T_{\text{shunt}}^H$ is given in time slots. We refer to $E$ as the edge set of the in-out graph $G_{\text{io}}$.

$$
\begin{array}{llrll}
\min & C & & & \\
\text{s.t.} & a_r^t & \leq & a_r^{t+1} & \forall r \in R^x, t \in T \quad \text{(5.7.1a)} \\
& d_r^t & \leq & d_r^{t+1} & \forall r \in R^y, t \in T \quad \text{(5.7.1b)} \\
& a_r^t & \geq & d_{r'}^{t+T_{\text{shunt}}^H} & \forall t \in \{0, \dots, \tau - T_{\text{shunt}}^H\}, \\
& & & & \forall (r, r') \in E \quad \text{(5.7.1c)} \\
& d_{r'}^t & = & 0 & \forall r' \in R^y, \\
& & & & \forall t \in \{0, \dots, T_{\text{shunt}}^H - 1\} \quad \text{(5.7.1d)} \\
& \sum\limits_{\substack{e \in E \\ e=(r,r')}} \text{vol}(e)(a_r^t - d_{r'}^t) & \leq & C & \forall t \in T \quad \text{(5.7.1e)} \\
& a_r^{t_i} & = & 0 & \forall r \in R^x : \text{arrive}_H(r) \overset{!}{>} t_i \quad \text{(5.7.1f)} \\
& d_{r'}^{t_i} & = & 1 & \forall r' \in R^y : \text{dep}(r') \overset{!}{<} t_i \quad \text{(5.7.1g)} \\
& a_r^0 = 0, \quad a_r^{\tau-1} = 1 & & & \forall r \in R^x \\
& d_{r'}^0 = 0, \quad d_{r'}^{\tau-1} = 1 & & & \forall r' \in R^y \quad \text{(5.7.1h)} \\
& \text{all } a_\cdot, d_\cdot \in \{0, 1\} & & & \quad \text{(5.7.1i)}
\end{array}
$$

Equations (5.7.1a), (5.7.1b) and (5.7.1h) impose that, for every edge $e$, the variables $a_e^\cdot$ and $d_e^\cdot$ form a monotone sequence starting with 0 and ending with 1. The idea is that the train arrives (or departs, respectively) at the time when the 0-1 transition takes place, i.e, for an $X$-route $r^x \in R^x$ we set $\text{arrive}_H(r^x) = t'$ if $a_{r^x}^{t'+1} - a_{r^x}^{t'} = 1$ and symmetrically $\text{dep}_H(r^y) = t''$ if $d_{r^y}^{t''+1} - d_{r^y}^{t''} = 1$ for a $Y$-route $r^y \in R^y$. Constraints (5.7.1c) and (5.7.1d) enforce that an outgoing train can only depart if all its cars have arrived and that $T_{\text{shunt}}^H$ time units are available for shunting those cars. Constraints (5.7.1e) represent the capacity constraint over all time slots, which is the objective value. Constraints (5.7.1f) and (5.7.1g) introduce time constraints for the earliest arrival / latest departure of trains, i.e., from the time windows we infer a constraint of type $\text{arrive}_H(r) > t'$ on the arrival (departure) times at the hub and express this in the form of Constraints (5.7.1f) and (5.7.1g).

Our experiments show that for the problem instances that arise from solutions to the TRP on our instances, in a few minutes we can calculate a shunting schedule for a given routing to and from the hub that minimizes the necessary hub capacity and respects the time windows.

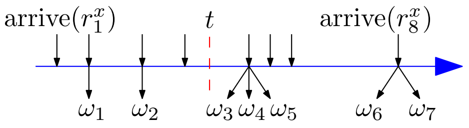
### 5.7.3 Optimal Grouping of Shunting Operations

In this section, we take a closer look at the shunting operation. Beforehand we assumed that it always takes a constant additional time $T^H_{\text{shunt}}$ to compose an outgoing train, which is clearly a rough model that ignores the concrete sequence of shunting operations that is necessary to compose the train. In fact, it is an interesting algorithmic problem to come up with methods that find a good sequence of these operations for a given shunting yard and a specification of incoming and outgoing trains. Surprisingly, this task turned out to be quite difficult for various reasonable models of what a good sequence precisely means. One important reason for that is that shunting yards differ in their layout and in the way in which they are operated.

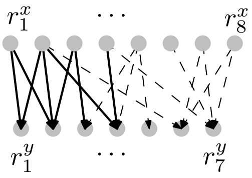
In the literature the question of shunting is addressed in a few publications [72, 103, 39, 40]. From some discussions with practitioners I can report that these techniques are not used in reality. Apart from the problem with the different layouts there is a further problem to the implementation of these methods: All presented schemes assume that there is a given fleet of incoming trains that waits in the shunting yard when shunting starts and that this fleet is to be shunted such that the outgoing trains are composed in one shot. In reality, shunting starts well before all incoming trains have arrived.

Here, we consider a simple model for this dynamic aspect, *grouped shunting*, in which we periodically decide to use one of the static shunting methods to shunt the outgoing trains for which all cars have arrived at the shunting yard. This results in a scheduling problem that is algorithmically interesting. However, we do not claim that this method is always the method of choice for a real shunting yard.

The problem setting is as follows. We assume here that we have already found a good order for the trains to arrive at the shunting yard. More precisely, let us assume that we have computed all targeted arrival times $\mathcal{I} =$


arrive($r_1^x$)
$t$
arrive($r_8^x$)
$\omega_1$ $\omega_2$ $\omega_3$ $\omega_4$ $\omega_5$ $\omega_6$ $\omega_7$

(a) time line with times of incoming and departing trains



(b) $G_t \subset G_{\text{io}}$ represents a possible configuration of cars in the shunting yard at time $t$ (solid edges), cf. Fig. 5.7.1(a).

**Figure 5.7.2:** *grouped shunting example*

$\{\text{arrive}_h(r_1), \ldots, \text{arrive}_h(r_m)\}$ of incoming trains $R^x$, for example by the methods of the last section. From this, we compute the earliest possible departure times $\mathcal{O} = \{\omega_1, \ldots, \omega_n\}$ of the outgoing trains $R^y$ as follows. The earliest possible departure time $\omega_i$ of an outgoing train $r_i^y$ is the latest arrival time $\text{arrive}_h(r_j^x)$ of an incoming train $r_j^x$ that has cars for $r_i^y$: $\omega_i = \max_{j:(r_j^x, r_i^y) \in E} \text{arrive}_h(r_j^x)$ with respect to the edge set $E$ of the in-out graph $G_{\text{io}}$. Note that these earliest possible departure times do not include the time needed for shunting in contrast to the actual departure time that are calculated by the algorithm that we present in this section. It follows that the earliest possible departure times are a subset of the arrival times ($\mathcal{O} \subseteq \mathcal{I}$), see Figure 5.7.2(a). The trains are indexed w.l.o.g. in the order of their arrival times/earliest possible departure times.

At time $t$ all cars that are in the shunting yard correspond to a subgraph $G_t$ of $G_{\text{io}}$, see Figure 5.7.2(b) for an example. If we start a shunting phase at time $t$, the set of all cars $O_t$ in the shunting yard at time $t$ belonging to complete outgoing trains are composed. In Figure 5.7.2(b) the set $O_t$ corresponds to all nodes in the bottom partition that have all their adjacent edges in $G_t$, i.e., $r_1^y$ and $r_2^y$. The remaining cars are left in the shunting yard. We further assume that the time needed for shunting $O_t$ depends on the number of cars in $O_t$, denoted by $|O_t|$. We assume that the shunting time is given by a monotone, concave function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n)$ is the time needed to shunt $n$ cars. Note that the concavity just states that a static shunting task for some set of cars cannot take longer than breaking this set up into subsets and sequentially perform the static shunting on these subsets. This property trivially holds for all sensible static shunting methods. Given a shunting operation starting at time $t$, the outgoing trains are composed in the time interval $[t, t + f(|O_t|)]$, and during that time no other shunting operation can take place.

Our task is to decide how to group the shunting operations, i.e., at which points in time we should start to shunt. Observe that the cars that are in the shunting yard at time $t$ depend on the grouping decisions before $t$. For the objective of makespan minimization $(C_{\max})$ we call the problem the *optimal grouping problem with makespan objective*. The makespan refers to the end of the last shunting phase. We give a dynamic programming algorithm for this problem.

For each $\omega_i \in \mathcal{O}$, the algorithm maintains a state $W(\omega_i) = (t', v')$ with the following properties. Time $t'$ is a point in time within the interval $I_i = [\omega_i, \omega_{i+1})$ and $v'$ represents the minimum number of cars waiting to be shunted at $t'$. Together, the pair $(t', v')$ represents a *partial solution until (interval)* $i$, that is, a solution to the problem restricted to the intervals up to $I_i$, which has ended shunting before or at time $t'$ and has $v'$ cars waiting to be shunted. The interval for the last point $\omega_{\max} \in \mathcal{O}$ is defined as $I_{\max} = [\omega_{\max}, \infty)$. For convenience, we write $t = W^t(\omega)$ and $v = W^v(\omega)$ for $W(\omega) = (t, v)$.

The key idea of the algorithm is that it suffices to store a single state for each interval. We express this by a dominance rule for two states of the same interval. The state $(t, v)$ *dominates* $(t', v')$ if and only if $t + f(v) < t' + f(v')$.

The following lemma makes the usefulness of dominance precise.

**Lemma 5.11 (Dominance Rule)** *Consider a solution $\sigma$ with makespan $C^{\sigma}_{\max}$ which starts a shunting phase with $v$ cars at time $t$ of interval $I_i$. Assume further that a state $(t', v')$, $t' \in I_i$ exists that dominates $(t, v)$. Then, a solution $\sigma'$ exists, which starts shunting with $v'$ cars at time $t'$ and has a makespan of less than or equal to $C^{\sigma}_{\max}$.*

**Proof.** As $(t', v')$ dominates $(t, v)$, we can construct a solution $\sigma'$ as follows. The existence of $(t', v')$ guarantees that a partial solution $P$ until $i$ exists. We build $\sigma'$ by using $P$ up to $t'$. At $t'$ we start a shunting phase that ends at $e' = t' + f(v')$, i.e., before $e = t + f(v)$, where the corresponding shunting phase in $\sigma$ ends (by definition of dominance). If $t$ is the start of the last shunting phase in $\sigma$ we already have a complete solution with a shorter makespan. Otherwise, the rest of the new solution consists of the grouping decisions in $\sigma$ at or after $e'$. Let $s_{\text{next}}$ be the start of the first shunting phase at or after $e'$. Note that $s_{\text{next}} \geq e > e'$ since $(t', v')$ dominates $(t, v)$. At $s_{\text{next}}$, the solution $\sigma'$ has exactly the same number of cars waiting for shunting as $\sigma$ has, since $t$ and $t'$ are in the same interval. The cars available at $s_{\text{next}}$ in $\sigma$ and $\sigma'$ are just the weights of outgoing trains in the interval from $[t, s_{\text{next}})$ resp. $[t', s_{\text{next}})$. These two values are identical. $\square$

A solution $\sigma$ induces a set of states in the intervals in which it starts and ends shunting and in the intervals in which it waits. For the starting and ending phases this is the exact time at which the shunting starts or ends together with the number of cars available for shunting at these times. A solution that waits in an interval $I_i$ induces the state $(\omega_i, v)$, where $v$ is the number of cars available for shunting at $\omega_i$. We say that a *solution dominates a state s* if it induces a state $s'$ in the interval of $s$ that dominates $s$. Similarly, we say that a solution $\sigma$ dominates a solution $\sigma'$ if $\sigma$ induces a state that dominates a state of $\sigma'$. Because of Lemma 5.11, it is sufficient to consider undominated solutions when searching for an optimal solution. We introduce the same notation for partial solutions until $i$, which only induce states in intervals $I_j,\ j \leq i$.

Note that dominance for the last interval $[\omega_{\max}, \infty)$ is equivalent to a better makespan. Therefore, an undominated solution is an optimal one. Furthermore, an undominated partial solution can be extended to an undominated optimal solution by the same arguments as in Lemma 5.11.

The dynamic program proceeds as follows, see Algorithm 5 for a precise formulation. First, we initialize prefix sums $S(\omega)$ for each event point. These sums stand for the cumulated number of cars of all outgoing trains up to time $\omega$. Then we iterate over the events chronologically and update the $W$ values. The crucial observation is that shunting at time $t = W^t(\omega)$ means that we keep the shunting yard busy for at least $\sigma = f(W^v(\omega))$ time. Let $\omega'$ be the event point directly before $t' = t + \sigma$. To find this event $\omega'$, we need a dictionary on $\mathcal{O}$ that supports predecessor queries. If we decide to start a shunting phase at $t$, then there is a feasible solution with state $(t', S(\omega') - S(\omega))$ in the interval of $\omega'$. We use the dominance rule to find out if this state should replace the current state in the interval of $\omega'$. In order to account for the possibility of not shunting directly after $t'$, we also have to update all states in intervals after $\omega'$, which we do implicitly in line 2 before accessing $W(\omega)$. After the last iteration, the values $W$ reflect an optimal solution. In order to find the minimum makespan, we need to add one extra state after the last event. In this state we calculate the finish time after the additional shunting operation at the end, i.e., the makespan.

**Theorem 5.12** *Algorithm 5 solves the grouping problem with makespan objective in $O(n \log n)$ time.*

**Proof.** We prove the correctness of the algorithm by the following invariant:

*At the end of the $i$-th iteration of the forall loop 1 the following two properties hold:*

**Algorithm 5**: Optimal grouping

```
   // Initialize Prefix Sums (in linear time in the obvious way)
   forall $\omega \in \mathcal{O}$ do $S(\omega) \longleftarrow \sum_{i:\omega_i \leq \omega} \mathrm{vol}(r_i^y)$
   // Initialize States
   $W(\omega_1) \longleftarrow (\omega_1, \mathrm{vol}(r_1^y))$
   $C_{\max} \longleftarrow \infty, v_{\mathrm{old}} \longleftarrow 0$
   // Iterate
1  forall $\omega \in \mathcal{O}$ in chronological order do
2      $(t,v) \longleftarrow W(\omega) \longleftarrow$ DOMINANCE$\big((\omega, v_{\mathrm{old}} + \mathrm{vol}(r_i^y)), W(\omega)\big)$
       $t' \longleftarrow t + f(v)$
3      if $t' < \omega_{\max}$ then
4          $\omega' \longleftarrow$ PREDECESSOR$(t')$
5          $W(\omega') \longleftarrow$ DOMINANCE$\Big(W(\omega'), \big(t', \omega + S(\omega') - S(\omega)\big)\Big)$
       else
6          $C_{\max} \longleftarrow \min\big\{C_{\max}, t' + f(S(\omega_{\max}) - S(\omega))\big\}$
       $v_{\mathrm{old}} \longleftarrow v$
   return $C_{\max}$
```

**INV1**$(i)$ *For all intervals $I_j$, with $j \leq i$ the state $(t,v) = W(\omega_j)$ is not dominated by any partial solution until $i$.*

**INV2**$(i)$ *No undominated partial solution until $k$ exists that dominates $W(\omega_k)$. and starts its last shunting phase before $\omega_i$ and ends this phase in interval $I_k, k > i$.*

The correctness of the invariant implies the correctness of the algorithm because non-domination implies optimality.

We prove the invariants by induction on $i$. For $i = 0$ there is nothing to prove. Consider iteration $i > 0$ and the corresponding state $(t,v) = W(\omega_i)$. For INV1$(i)$ we have to set $(t,v)$ to a state that is not dominated. Such a state corresponds to a specific partial solution until $i$. If that solution ends a shunting phase in $I_i$ then $W(\omega_i)$ is already set correctly by INV2$(i-1)$. If this is not the case, then this solution ends a shunting phase before $I_i$ and waits in $I_i$. In this case $(\omega_i, W^v(\omega_{i-1}) + \mathrm{vol}(r_i^y))$ is a non dominated state and it is assigned to $W(\omega_i)$ in line 2. This makes use of the fact that $W^v(\omega_{i-1})$ is undominated until $i-1$ because of INV1$(i-1)$. After line 2 the state $W(\omega)$ cannot be dominated by another state and INV1$(i)$ holds.

Line 5 creates the state that corresponds to a start of a shunting phase in $i$ and updates the interval in which the phase ends. After this update INV2$(i)$ holds:

We have to check the property for all undominated partial solutions until $k$ that start a shunting phase in $I_i$, for the others it is clear from INV2$(i-1)$. We know from INV1$(i)$ that in $I_i$ the state $W(\omega_i)$ cannot be dominated. Therefore, any undominated solution that starts shunting in $I_i$ has to end this shunting phase in the interval of $\omega'$, see Lines 4 and 5. This implies that we do the only necessary update to preserve the second property.

We can use a balanced search tree for the predecessor queries which guarantees a running time of $O(n \log n)$. $\square$

## 5.8 Experiments

In this section we report on the experimental results with the three models.

### 5.8.1 Instance

The planners of SBB Cargo Express Service provided us with real data, i.e., the actual railway network and an (averaged) supply and demand matrix of a day in Summer 2005. As already mentioned, the condensed network has 121 nodes and 332 edges. In Figures 5.8.1 and 5.8.2 we show the original network (green edges, all nodes) together with the condensed network that we extracted from it (black edges, black and blue nodes). In total, there are around 200 shipments that are transported almost every day. These data represent the supply and demand averages over the workdays of a week. However, within one week, the supply and demand changes (slightly), that is, on a fixed day, not all of these shipments are really present. Note that one would expect that the averaged instance has higher cost than the average weekday cost because the former contains the union of all shipments present in the weekdays.

In order to evaluate the quality of our solutions we evaluated the cost of the current hand-made schedules in our model. These schedules are for the current situation with a single hub. We are well aware that this does not necessarily equal the exact real cost of such a solution. Both in our model and for this evaluation we set the cost of an engine to an estimative big-M-like value of 1000 in comparison to the unit cost we charge for a driven kilometer. Note that small to medium changes of this value cannot influence the structure of the optimal solution as long as the distance that can be saved by employing an additional engine is well below this value. The resulting costs of the hand-made solutions are shown in Table 5.1.

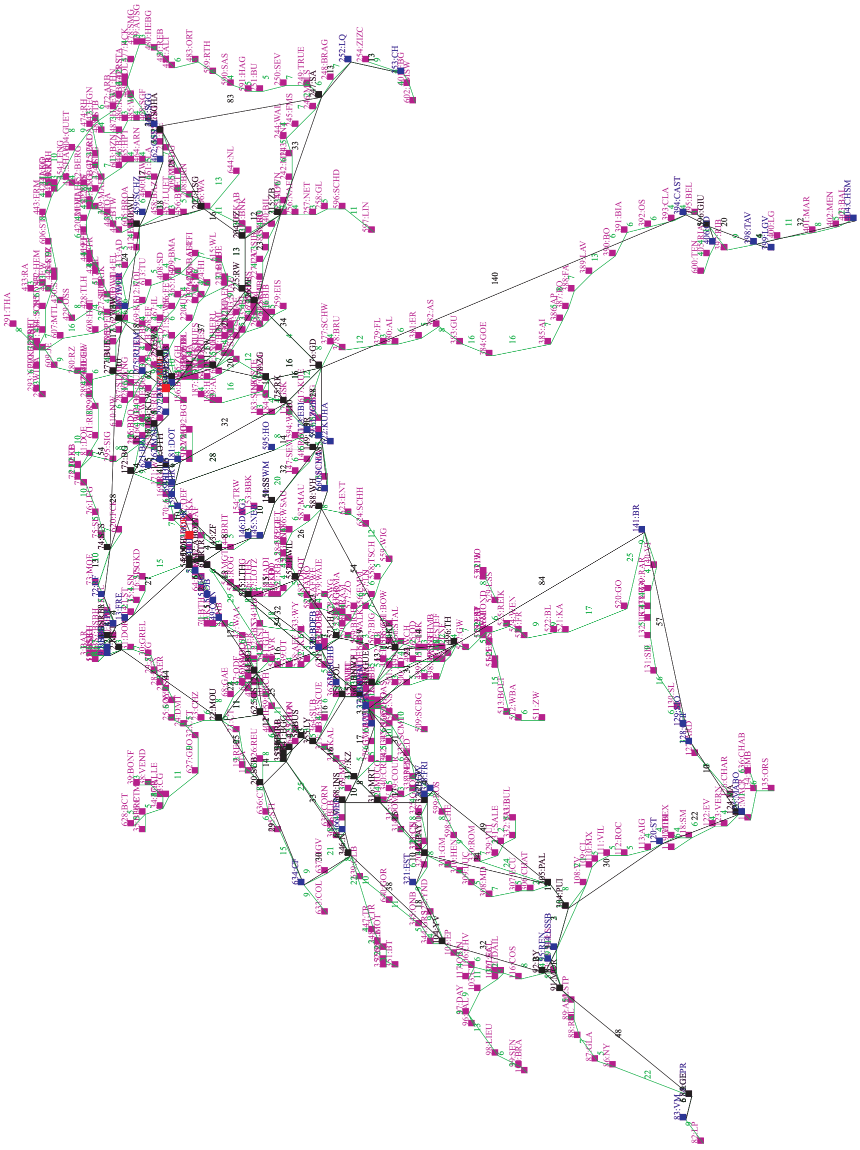


**Figure 5.8.1:** *The original railway network together with the condensed network that we extracted. The stations are given together with their SBB codes. Blue nodes are stations with shipments in the Cargo Express Service, black nodes are stations without shipments that were retained in the condensed network to keep it sparse. The red nodes correspond to hubs. The green nodes and edges are stations that are not retained in the condensed network.*

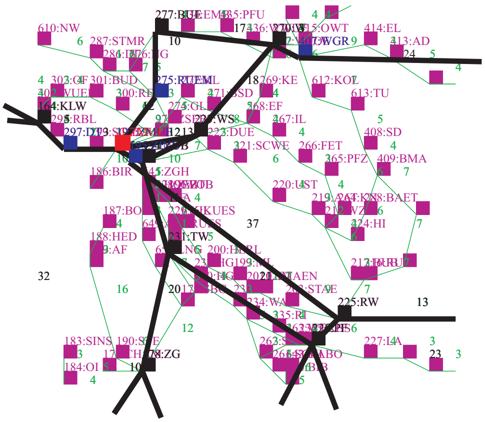


**Figure 5.8.2:** *An enlarged detail of the complete network of Figure 5.8.1.*

**Table 5.1:** *Estimated costs of the current hand-made schedules in our model*

| Monday | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|
| 48534 | 52086 | 51078 | 52086 | 52100 |

If we compare the size of our problem instance to the size of the largest VRP instances that can be solved exactly as presented in a typical VRP survey paper [19] for the "easier" standard vehicle routing problem, it turns out that the SBB instance has more shipments than each of the instances presented there. Also, the underlying network has more nodes than each of the instances in this publication except for `E151-12c`. This emphasizes the fact that the SBB instance is a challenging one, even more so as we try to solve the routing and the scheduling problem together. We do not expect however to find an exact solution for our instance.

## 5.8.2 Model 0

As already mentioned above we implemented Model 0 using the OPL Modeling language [71]. The model can be found in Appendix A.2. On a toy instance with 14 nodes, 24 edges and 11 shipments we did not get any feasible solution on Machine B (see Appendix A.1) in 40 hours. This result can be seen as a justification to develop and implement more involved approaches like Model 1 and 2.

**Table 5.2:** *Parameter settings for Model 1*

| $L_{\max}$ | $D_{\max}$ | $T^H_{\text{shunt}}$ | $T^S_{\text{couple}}$ | $\bar{v}$ |
|---|---|---|---|---|
| 25 cars | 288km | 15min | 8min | 90 km/h |

### 5.8.3 Model 1

We implemented Model 1 using SYMPHONY 5, a branch and cut framework by Ralphs et al. [106]. We used CPLEX 9 as LP solver for SYMPHONY, and LEDA 4.5 for computing the minimum spanning trees and the assignment problems.

#### TRP Instance

Out of the SBB Cargo data we extracted the $X$- and the $Y$-instance for the TRP. The complete data set is confidential but we will discuss the most important properties here. We set the parameters to the values show in Table 5.2.

The setting $\bar{v} = 90km/h$ overestimates the actual speed of the freight trains, as we found out later. To produce the aggregated set of shipments we aggregated in a first step all shipments with identical source for the $X$-instance and all shipments with identical destination for the $Y$-instance. This aggregation turns out to be too drastic, as it produces shipments that exceed the maximum train load. Therefore, in a second step we undid some of the aggregations. This was done by hand in cooperation with the SBB Cargo planners, as various rules apply as to when two shipments are usually taken together. This led us to an $X$-instance on a nearly complete graph with 33 nodes and 40 shipments and a $Y$-instance on a nearly complete graph with 34 nodes and 44 shipments. We set the size of the train fleet to 24.

#### TRP solution

As the train fleet is fixed we can express the solution cost in kilometers. The distances include a “kilometer equivalent” of 12km for each pickup at a station. We carried out our experiments on Machine A (see Appendix A.1). For the $X$-instance it took our optimization code 30 minutes to find the first feasible solution of cost 2510km, after 5 hours it found the best solution of cost 2280km before we stopped the calculation after 26 hours. At this point the lower bound certified that our best solution is at most 17% off the optimal solution. For the $Y$-instance the solution time was 130 minutes for the first feasible solution of cost 3363km.

The best solution found has cost 3197km and is 33% above the lower bound after 63 hours. Compared to our experience with Model 0 and other similar models this means that the decomposition allows us to simplify the routing part in such a way that we can find solutions to comparatively big instances in reasonable time.

### 5.8.4 TSSP Instance and Overall Solution

The two solutions to the TRP that we found constitute the input to the TSSP. With the actual time windows of SBB Cargo they constitute an infeasible instance for the TSSP because there are 4 $X$-routes that arrive after $Y$-routes that depend on them depart. At this point, the drawback of the decomposition becomes clear. To overcome this problem we chose the second best $Y$-solution that was found (cost 3452km), which reduced the number of incompatible trains to 1. We then analyzed the solution pair by hand and solved the last incompatibility by forbidding two track segments in the $X$-instance. This lead to a solution to the $X$-instance of cost 2440km. It follows that in our model the overall cost of this solution is 30892, which is well below the costs of the hand-made solutions. However the problem with this (partial) solution arises from the TSSP instance that it represents: The resulting TSSP instance could be solved in a few minutes on Machine A (see Appendix A.1). The necessary hub-capacity is 252. This high value is unrealistic for the size of the actual shunting yard of the SBB network. One reason that it is so high is that we tried to integrate most of the shipments that are currently transported by direct trains into the hub-spoke system. A second reason is that our decomposition approach only considers the hub-capacity in the second step and ignores it for the TRP.

In order to partially overcome such problems we integrated a limited interactivity into our model that would allow the planner to forbid or fix edges. We used this mode to obtain the “hand-optimized” solution above. It allows the planner to incorporate to some degree external constraints into the model that are not present in the formulation.

To sum up, we were able to find a semi-realistic solution to the one-hub problem but also hit the limits of this decomposition approach. The experience with this model (together with SBB’s migration to a multi-hub system) motivated us to develop the column generation model.

### 5.8.5 Model 2

We implemented Model 2 using the SCIP library by Achterberg [3]. We are the first to use SCIP for a column generation approach.

**Table 5.3:** *Parameter settings for Model 2*

| $L_{\max}$ | $T^H_{\text{shunt}}$ | $T^S_{\text{couple}}$ | $\bar{v}$ | $\text{cap}_h$ |
|---|---|---|---|---|
| 25 cars | 54min | 27min | 60 km/h | 100 |

### Preliminary results

In this section we report on some preliminary results of Model 2.

First, the full instance, on which the CG approach works, is considerably larger than the aggregated instances for the TRP. We run our instances with parameters that we partly rechecked with the SBB cargo planners and set to conservative values, see Table 5.3.

Moreover, we allow to use an additional hub now, as intended by the SBB planners. With our current implementation we reach the tailing-off phase in the root node of the branch and price tree after a calculation time of around three days on Machine C (see Appendix A.1). At this time the value of the relaxation is 20525, the value of the dual bound via Lemma 4.8 is 15920.

The dive and fix heuristic of Section 5.6.4 finds an integral solution of cost 34861. This value emphasizes the quality of our solution in the model, if we compare it to the values around 50000 of the hand-made solutions. It is important to point out here that the hand-made solution is for a single hub, whereas we allow to use two hubs. In general, it is too early to conclude that our solution is superior to the hand-made ones even if the objective values are promising. This has to be verified in cooperation with the SBB planners.

## 5.9 Related Work

The full problem that we modeled in this chapter and attacked from different angles is special enough to be new in the sense that it has never been studied before in the literature. On the other hand, some of its components and related problems have received considerable attention in the literature.

As for the routing part, the TRP defined in this chapter can be seen as a special vehicle routing problem. The vehicle routing problem (VRP) itself has been studied in many variants, see the book edited by Toth and Vigo [124] for a survey or the annotated bibliography by Laporte [84]. As we saw in Section 5.5.3, the TRP can be transformed to a DCVRP problem. Among the VRP problems DCVRP has received comparatively little attention. Most of the publications of exact algorithms date back to the 80ies [27, 85, 86]. There are several implementations for the general vehicle routing problem, commercial as well as free ones, see [70, 105] a survey. One of the few free and open ones is the code by Ralphs et al. [107], on which we base the implementation of Model 1. Ralphs' implementation is itself based on his SYMPHONY branch and cut framework [106]. Another branch and cut implementation for the vehicle routing problem is by Blasum and Hochstättler [16].

Column Generation plays a prominent role for vehicle routing problems, see [46, 117, 19] for a survey.

There are several publications related to the shunting of trains [89, 125]. However, most of these refer to a different problem, the shunting of unused passenger trains that are parked in a shunting yard and undergo cleaning and maintenance checks there. This setting is completely different from ours. In [15] the authors consider the shunting of trams in the morning which also differs from shunting freight trains. In [88] shunting without a hump in a different model is considered. In [39, 40] the authors model a problem that is similar to ours. However, they do not consider the dynamic aspect and only partly the capacity restriction. Their algorithm can be understood as one of the black-box static shunting algorithms used in Section 5.7.3.

The paper by van Wezel and Riezenbos [125] also discusses why so many planning tasks in railway optimization are still performed by hand in spite of numerous optimization efforts in the Algorithms and Operations Research community. They come to the conclusion that apart from the quality of the mathematical model itself, also robustness and flexibility issues, software engineering problems, and psychological questions play an important role.

## 5.10 Summary of Results

In this chapter we have seen how a sequence of models has been developed to capture an involved scheduling and routing problem. From a practical point of view, this seems to be a typical phenomenon: Rather than developing the ideal model in one shot it often takes some iterations and feedback cycles to come up with a useful model. The promising results of the column generation approach suggest that indeed the practical applicability of our approach is in reach.

On the theoretical side we have shown how the mincut-linear arrangement problem, one of the showcase problems for the application of divide-and-conquer in approximation algorithms [116], is at the core of the sequencing problem at the hub. Furthermore, we have provided an exact solution to a scheduling problem that is inspired by the shunting operation at the hub.

The results of this chapter that cover Model 0 and 1 are joint work with Michael Gatto. The theoretical results from Section 5.7 are joint work with Michael Gatto and Riko Jacob. The implementation of Model 2 is at the time of writing this thesis an ongoing project with Tobias Achterberg, Alberto Ceselli, Michael Gatto, Marco E. Lübbecke, and Heiko Schilling. The topics of this chapter will also be covered with a different focus in the dissertation of Michael Gatto.

## 5.11 Open Problems

There is a multitude of problems connected to the topics of this chapter that can be the subject of further research. Apart from the further development of mathematical models and the application of advanced column generation techniques for the train routing problem, also algorithms that consider the exact shunting operations at a hub are interesting. As already mentioned this is a challenging task because shunting yards differ in their exact layout and their way of operation. Furthermore, we have not considered the engine driver assignment at all. Finally, also robustness issues are an important aspect, which can lead to interesting variations of the problems presented here.

# Chapter 6

# OVSF Code Assignment

Morpheus: “What can you see, Neo?”
Neo: “It’s strange... the code is somehow different...”
(from The Matrix, Reloaded)

## 6.1 Introduction

In the last years the field of telecommunications has raised a multitude of interesting new algorithmic questions. In this chapter we treat a code reassignment problem that arises in the Wideband Code Division Multiple Access method (W-CDMA) of the Universal Mobile Telecommunications System (UMTS, for more details see [74, 83]). More precisely, we focus on its multiple access method Direct Sequence Code Division Multiple Access (DS-CDMA). The purpose of this access method is to enable all users in one cell to share the common resource, i.e., the bandwidth. In DS-CDMA this is accomplished by a spreading and scrambling operation. Here we are interested in the spreading operation that spreads the signal and separates the transmissions from the base-station to the different users. More precisely, we consider spreading by Orthogonal Variable Spreading Factor (OVSF-) codes [4, 74], which are used on the downlink (from the base station to the user) and the dedicated channel (used for special signaling) of the uplink (from user to base station). These codes are derived from a code tree. The OVSF-code tree is a complete binary tree of height $h$ that is constructed in the following way: The root is labeled with the vector $(1)$, the left child of a node labeled $a$ is labeled with $(a, a)$, and the right child with $(a, -a)$. Each user in one cell is assigned a different OVSF-code. The key property that separates the signals sent to the users is the *mutual orthogonality* of the users' codes. All assigned codes are mutually orthogonal if and only if there is at most one assigned code on each root-to-leaf path. In DS-CDMA users request different data rates and get OVSF-codes of different levels. The data rate is inversely proportional to the length of the code. In particular, it is irrelevant which code on a level a user gets, as long as all assigned codes are mutually orthogonal. We say that an assigned code in any node in the tree *blocks* all codes in the subtree rooted at that node and all codes on the path to the root, see Figure 6.1.1 for an illustration.

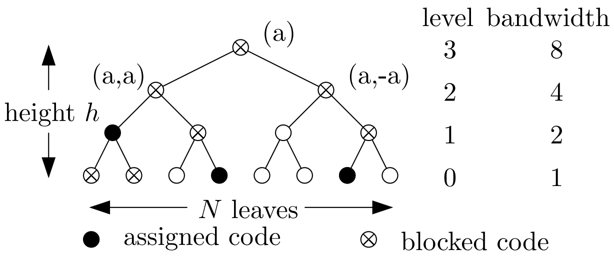


**Figure 6.1.1:** *A code assignment and blocked codes.*

As users connect to and disconnect from a given base station, i.e., request and release codes, the code tree can get fragmented. It can happen that a code request for a higher level cannot be served at all because lower level codes block *all* codes on this level. For example, in Figure 6.1.1 no code can be inserted on level two without reassigning another code, even though there is enough available bandwidth. This problem is known as *code blocking* or *code-tree fragmentation* [83, 93]. One way of solving this problem is to reassign some codes in the tree (more precisely, to assign different OVSF-codes of the same level to some users in the cell). In Figure 6.1.2 some user requests a code on level two, where all codes are blocked. Still, after reassigning some of the already assigned codes as indicated by the dashed arrows, the request can be served. Here and in many of the following figures, we only depict the relevant parts (subtrees) of the single code tree.

The process of reassigning codes necessarily induces signaling overhead from the base station to the users whose codes change. This overhead should be kept small. Therefore, a natural objective already stated in [93, 109] is to serve all code requests as long as this is possible, while keeping the number of reassignments as small as possible. As long as the total bandwidth of all simultaneously active code requests does not exceed the total bandwidth, it is always possible to serve them. The problem has been studied before with a focus on simulations. In [93] the problem of reassigning the codes for a single additional request is

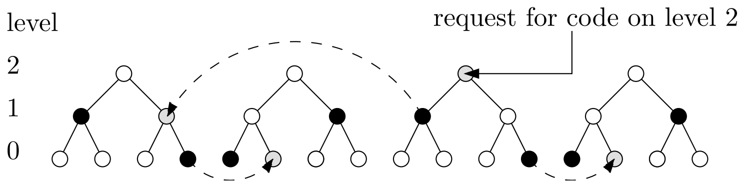


**Figure 6.1.2:** *A code insertion on level 2 into a single code tree $T$, shown without the top levels.*

introduced. The Dynamic Code Assignment (DCA) algorithm presented in [93] is claimed to be optimal. We prove that this algorithm is not always optimal and analyze natural versions of the underlying code assignment (CA) problem. Our intention is to present a rigorous analysis of this problem.

### 6.1.1 Related Work

It was a paper by Minn and Siu [93] that originally drew our attention to this problem. The one-step offline code assignment problem is defined together with an algorithm that is claimed to solve it optimally [93]. As we show in Section 6.3.1, this claim is not correct. Many of the follow-up papers like [9, 22, 24, 60, 61, 78, 109] acknowledge the original problem to be solved by Minn and Siu and study some other aspects of it. Assarut et al. [9] evaluate the performance of Minn and Siu's DCA-algorithm, and compare it to other schemes. Moreover, they propose a different algorithm for a more restricted setting [8]. Others use additional mechanisms like time multiplexing or code sharing on top of the original problem setting in order to mitigate the code blocking problem [22, 109]. A different direction is to use a heuristic approach that solves the problem for small input instances [22]. Kam, Minn and Siu [78] address the problem in the context of bursty traffic and different Quality of Service (QoS). They come up with a notion of "fairness" and also propose to use multiplexing. Priority based schemes for different QoS classes can be found in [25], similar in perspective are [60, 61].

Fantacci and Nannicini [55] are among the first to express the problem in its online version, although they have quite a different focus. They present a scheme that is similar to the compact representation scheme in Section 6.5, without focusing on the number of reassignments. Rouskas and Skoutas [109] propose a greedy online-algorithm that minimizes, in each step, the number of additionally blocked codes, and provide simulation results but no analysis. Chen and Chen [26] propose a best-fit least-recently used approach, also without analysis.

After our first publication on OVSF code assignment [50] some more results were found. Tomamichel [121] shows that the general offline CA problem is NP-complete and there exist instances of code trees for which any optimal offline greedy algorithm needs to reassign more than one code per insertion/deletion request.

### 6.1.2 Model and Notation

We consider the combinatorial problem of assigning codes to users. The codes are the nodes of an (OVSF-) code tree $T = (V, E)$. Here $T$ is a complete binary tree of height $h$. The set of all users using a code at a given moment in time can be modeled by a *request vector* $r = (r_0 \dots r_h) \in \mathbb{N}^{h+1}$, where $r_i$ is the number of users requesting a code on level $i$ (with bandwidth $2^i$). The levels of the tree are counted from the leaves to the root starting at level 0. The level of node $v$ is denoted by $l(v)$.

Each request is assigned to a position (node) in the tree, such that for all levels $i \in \{0 \dots h\}$ there are exactly $r_i$ codes on level $i$. Moreover, on every path $p_j$ from a leaf $j$ to the root there is at most one code assigned. We call every set of positions $F \subset V$ in the tree $T$ that fulfills these properties a *code assignment*. If we want to emphasize the feasibility of $F \subset V$ we also use the term *feasible code assignment*. For ease of presentation we denote the set of *codes* by $F$. Throughout this chapter, a code tree is the tree together with a code assignment $F$. If a user connects to the base station, the resulting additional request for a code represents a *code insertion* (on a given level). If some user disconnects, this represents a *deletion* (at a given position). A new request is dropped if it cannot be served. This is the case if its acceptance would exceed the total bandwidth. By $N$ we denote the number of leaves of $T$ and by $n$ the number of assigned codes $n = |F| \leq N$. After an insertion on level $l_t$ at time $t$, any CA-algorithm must change the code assignment $F_t$ into $F_{t+1}$ for the new request vector $r' = (r_0, \dots, r_{l_t} + 1, \dots, r_h)$. The size $|F_{t+1} \setminus F_t|$ corresponds to the number of *reassignments*. This implies that for an insertion, the new assignment is counted as a reassignment. We define the number of reassignments as the cost function. Deletions are not considered in the cost function. They are charged to the insertions. We can do that without any asymptotic overhead since every code can be deleted at most once. When we want to emphasize the combinatorial side of the problem we call a reassignment a *movement* of a code. A maximal subtree of unblocked codes is called a *gap tree* (cf. Figure 6.5.6 (a) in Section 6.5).

We state the original CA problem studied by Minn and Siu together with some of its natural variants:

**one-step offline CA** Given a code assignment $F$ for a request vector $r$ in an OVSF code tree $T$ and a code request for level $l$. Find a code assignment $F'$ for the new request vector $r' = (r_0, \ldots, r_l + 1, \ldots, r_h)$ with minimum number of reassignments.

**general offline CA** Given a sequence $S$ of code insertions and deletions of length $m$. Find a sequence of code assignments in an OVSF code tree $T$ such that the total number of reassignments is minimum, assuming the initial code tree is empty.

**online CA** The code insertion and deletion requests are served as they arrive without knowledge of the future requests. The cost function is again the total number of reassignments over the whole request sequence.

**insertion-only online CA** This is the online CA with insertions only.

### 6.1.3 Summary of Results

The results presented in this chapter are joint work with Thomas Erlebach, Riko Jacob, Matúš Mihal'ák, Gábor Szabó and Peter Widmayer. As already discussed in the introduction, it is not always easy to divide the work done together and we do not want to lose in readability by leaving out some of the relevant results. An extended abstract of these results is presented in [53], and has and will be presented with different focus in the theses of Gábor Szabó [120] and Matúš Mihal̆ák.

This chapter consists of two main parts: One on the one-step offline CA and one on the online CA. These parts are preceded by a section, in which we discuss some general properties of the problems.

In the first part we begin with a counter-example to the DCA-algorithm. We proceed with an NP-completeness proof in Section 6.3.2. This result is my main contribution to this chapter. In Section 6.3.3 we present an exact algorithm for one-step offline CA with a running time that is exponential in the height $h$ of the tree. We show that a natural greedy algorithm already mentioned in [93] achieves approximation ratio $h$. The involved proof of this result will appear in the thesis of Matúš Mihal'ák. Finally, we consider the fixed parameter tractability of one-step offline CA.

In the second part we tackle the online-problem. It is a more natural version of the problem, because we are interested in minimizing the signaling overhead over a sequence of operations rather than for a single operation only.

We present a $\Theta(h)$-competitive algorithm and show that the greedy strategy that minimizes the number of reassignments in every step is not better than

$\Omega(h)$-competitive in the worst case. This means that even an optimal algorithm for one-step CA, which solves an NP-complete problem in every step, is only $\Omega(h)$ competitive. Also another strategy proposed in the literature delivers no more than $\Omega(h)$-competitiveness but is optimal in an insertion only scenario, as we show in Section 6.5.3. Finally, we sketch an online-algorithm with constant competitive ratio that uses resource augmentation, where we give its code tree one more level than the adversary. The details of this algorithm can be found in the thesis of Gábor Szabó [120] and in [53].

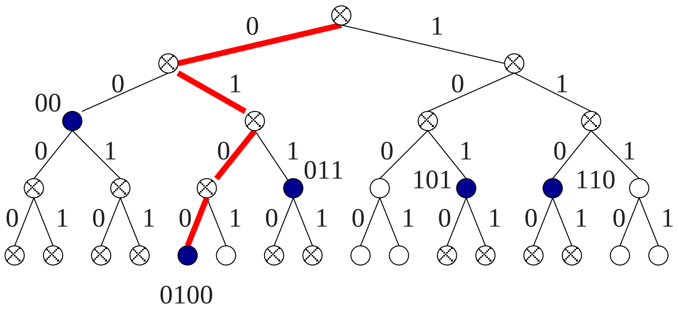


**Figure 6.2.1:** *Correspondence of code assignments in tree of height* $4$ *with codes on levels* $\{$*0,1,1,1,2*$\}$ *and prefix free codes of lengths* $\{$*4,3,3,3,2*$\}$

## 6.2 Properties of OVSF Code Assignment

In this section we present important properties of the OVSF code assignment problem that are relevant for the understanding of the following sections.

### 6.2.1 Feasibility

One might ask whether there always exists a feasible code assignment for a new code request. We present the necessary conditions for a feasible code assignment. In later sections we will always assume that a code assignment is possible for the current set of codes.

Given an assignment $F$ of $n$ codes in an OVSF code tree $T$ according to the request vector $r = (r_0, \ldots, r_h)$ and a new code request on level $l_i$, the question is whether a code assignment $F'$ exists for the new request vector $r' = (r_0, \ldots, r_{l_i} + 1, \ldots, r_h)$. Every assigned code on level $l$ has its unique path from the root to a node of length $h - l$. The path can be encoded by a word $w \in \{0, 1\}^{h-l}$ that describes the left/right decisions on this path. The orthogonality property amounts to demanding that these words form a binary prefix free code. Given a prefix free code set $C_{\text{pf}}$ with code lengths $\{h - l_1, \ldots, h - l_{n+1}\}$ (where $l_i$ is the level of code $i \in \{1, \ldots, n + 1\}$) we can clearly assign codes on levels $l_i$ by following the paths described by the code words in $C_{\text{pf}}$ (see Figure 6.2.1). This shows that a code assignment $F'$ for codes on levels $l_1, \ldots, l_{n+1}$ exists if and only if there exists a binary prefix free code set of given code lengths $\{h - l_1, \ldots, h - l_{n+1}\}$.

We use the Kraft-McMillan inequality to check the existence of a prefix free code set of given code lengths.

**Theorem 6.1** *[5] A binary prefix free code set with code lengths $a_1, \dots, a_m$ exists if and only if*

$$\sum_{i=1}^{m} 2^{-a_i} \leq 1. \tag{6.2.1}$$

If we multiply Equation (6.2.1) by $2^h$ and we consider the number of codes $r_{l_i}$ that are requested on level $l_i$ we get the following corollary.

**Corollary 6.2** *Given an OVSF code tree $T$ of height $h$ with $N = 2^h$ leaves and a request vector $r = (r_0, \dots, r_h)$ a feasible code assignment exists if and only if*

$$\sum_{i=0}^{h} r_i \cdot 2^i \leq N.$$

Corollary 6.2 shows that checking the existence of a feasible code assignment given the request vector can be done in linear time.

## 6.2.2 Irrelevance of Higher Level Codes

We show that an optimal algorithm for the one-step CA problem moves only codes on levels lower than the requested level $l_r$. A similar result was already given in [93]. In [93] the authors mention without proof that the optimal algorithm does not need to move codes on higher levels than the requested level. We give the proof of a similar statement here.

**Lemma 6.3** *Let $c$ be an insertion on level $l_r$ into a code tree $T$. Then for every code reassignment $F'$ that inserts $c$ and that moves a code on level $l \geq l_r$ there exists a code reassignment $F''$ that inserts $c$ and moves fewer codes, i.e., with $|F'' \setminus F| < |F' \setminus F|$.*

**Proof.** Let $x \in F$ be the highest code that is reassigned by $F'$ on a level above the level $l_r$ and let $S$ denote the set of codes moved by $F'$ into the subtree $T_x$ rooted at node $x$. We denote by $R$ the rest of the codes that are moved by $F'$ (see Figure 6.2.2). The cost of $F'$ is $|S| + |R|$. The code reassignment $F''$ is defined as follows: let $y$ be the position where $F'$ moves the code $x$, then $F''$ will move the codes in $S$ into the subtree $T_y$ rooted at $y$ and leave the code $x$ in $T_x$ and move the rest of the codes in $R$ in the same way as $F'$. The cost of $F''$ is at least one less than the cost of $F'$ since it does not move the code $x$. In Figure 6.2.2 the cost of $F'$ is 6 and the cost of $F''$ is 5. $\square$

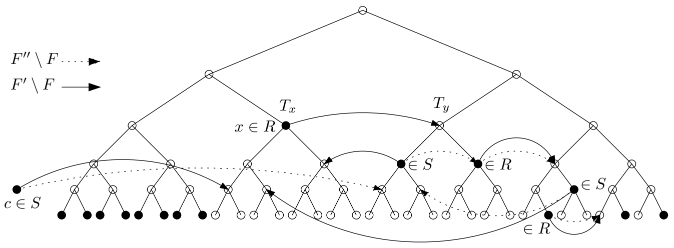


**Figure 6.2.2:** *Non-optimality of a code assignment $F'$ that reassigns codes also on higher levels than the requested level.*

## 6.3 One-Step Offline CA

In this section we present results for the one-step CA problem.

### 6.3.1 Non-Optimality of Greedy Algorithms

First we look at possible greedy algorithms for the one-step offline CA. A straightforward greedy approach is to select for a code insertion a subtree with minimum cost that is not blocked by a code above the requested level, according to some cost function. All codes in the selected subtree must then be reassigned. So in every step a top-down greedy algorithm chooses the maximum bandwidth code that has to be reassigned, places it at the root of a minimum cost subtree, takes out the codes in that subtree and proceeds recursively. The DCA-algorithm in [93] works in this way. The authors propose different cost functions, among which the "topology search" cost function is claimed to solve the one-step offline CA optimally. Here we show the following theorem:

**Theorem 6.4** *Any top-down greedy algorithm $A_{tdg}$ depending only on the current assignment of the considered subtree is not optimal.*

As all proposed cost functions in [93] depend only on the current assignment of the considered subtree, this theorem implies the non-optimality of the DCA-algorithm.

**Proof.** Our construction considers the subtrees in Figure 6.3.1 and the assignment of a new code to the root of the tree $T_0$. The rest of the subtrees that are not shown are supposed to be fully assigned with codes on the leaf level, so that no optimal algorithm moves codes into those subtrees. Tree $T_0$ has a code of bandwidth $2k$ on level $l$ and depending on the cost function has or does not have a code with bandwidth $k$ on level $l-1$. The subtree $T_1$ contains $k-1$ codes at leaf level and the rest of the subtree is empty. The subtrees $T_2$ and $T_3$ contain $k$ codes at leaf level interleaved with $k$ free leaves. As we will show in Corollary 6.9 any optimal one-step algorithm can be forced to produce such an assignment. This original assignment rules out all cost functions that do not put the initial code at the root of $T_0$. We are left with two cases:

**case 1:** The cost function evaluates $T_2$ and $T_3$ as cheaper than $T_1$. In this case we let the subtree $T_0$ contain only the code with bandwidth $2k$. Algorithm $A_{\text{tdg}}$ reassigns the code with bandwidth $2k$ to the root of the subtree $T_2$ or $T_3$, which causes one more reassignment than assigning it to the root of $T_1$, hence the algorithm fails to produce the optimal solution.

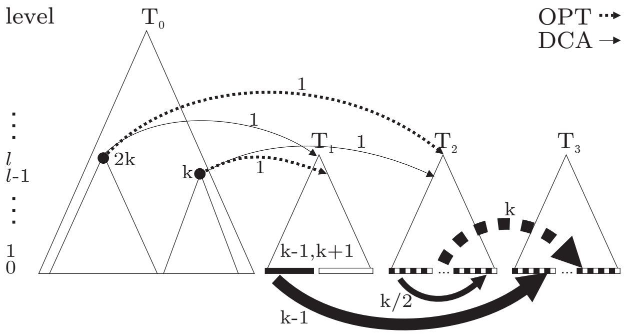


**Figure 6.3.1:** *Example for the proof of Theorem 6.4.*

**case 2:** The cost function evaluates $T_1$ as cheaper than $T_2$ and $T_3$. In this case we let the subtree $T_0$ have both codes. $A_{\text{tdg}}$ moves the code with bandwidth $2k$ to the root of $T_1$ and the code with bandwidth $k$ into the tree $T_2$ or $T_3$, see solid lines in Figure 6.3.1. The number of reassigned codes is $3k/2 + 2$. But the minimum number of reassignments is $k + 3$, which is achieved when the code with bandwidth $k$ is moved in the empty part of $T_1$ and the code with bandwidth $2k$ is moved to the root of $T_2$ or $T_3$, see dashed lines in Figure 6.3.1.

$\square$

### 6.3.2 NP-Hardness

We prove the decision variant of the one-step offline CA to be NP-complete. The canonical decision variant of it is to decide whether a new code insertion can be handled with cost less or equal to a number $c_{\max}$, which is also part of the input. First of all, we note that the decision-variant is in NP, because we can guess an optimal assignment and verify in polynomial time if it is feasible and if its cost is lower or equal to $c_{\max}$. The NP-completeness is established by a reduction from the three-dimensional matching problem (3DM) that we restate here for completeness (cf. [62]):

**Problem 6.5** *(3DM) Given a set $M \subseteq W \times X \times Y$, where $W, X$ and $Y$ are disjoint sets having the same number $q$ of elements. Does $M$ contain a perfect matching, i.e., a subset $M' \subseteq M$ such that $|M'| = q$ and no two elements of $M'$ agree in any coordinate?*

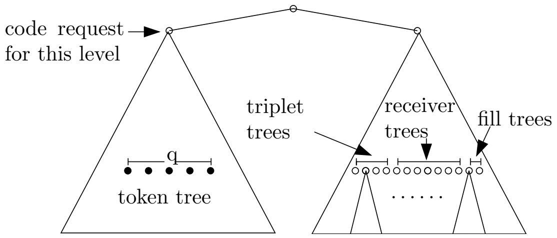
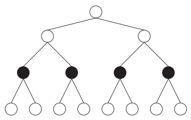


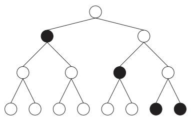
**Figure 6.3.2:** *Sketch of the construction*

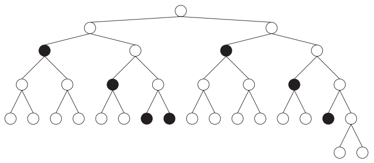
Let the elements of the ground sets $W, X, Y$ be indexed from 1 to $q$. To simplify the presentation, we introduce the *indicator vector* of a triplet $(w_i, x_j, y_k)$ as a zero-one vector of length $3q$ that is all zero except at the indices $i, q+j$ and $2q+k$. The idea of the reduction is to view the triplets as such indicator vectors and to observe that the problem 3DM is equivalent to finding a subset of $q$ indicator vectors out of the indicator vectors in $M$ that sum up to the all-one vector.

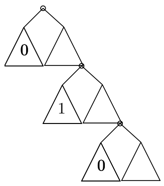
Figure 6.3.2 shows an outline of the construction that we use for the reduction. An input to 3DM is transformed into an initial feasible assignment that consists of a token tree on the left side and different smaller trees on the right. A code insertion request is given at the level indicated in the figure. The construction is set up in such a way that the code must be assigned to the root of the left tree, the *token tree*, in order to minimize the number of reassignments. Similarly, the $q$ codes that are forced to move from the left to the right tree must be assigned to the roots of *triplet trees*. The choice of the $q$ triplet trees reflects the choice of the corresponding triplets of a matching. All codes in the chosen triplet trees find a place without any additional reassignment if and only if these triplets really represent a 3D matching.

Let us now look into the details of the construction. The token tree consists of $q$ codes positioned arbitrarily on level $l_{\text{start}}$ with sufficient depth, for example depth $\lceil \log(|M| + 21q^2 + q) \rceil + 1$. The triplet trees have their roots on the same level $l_{\text{start}}$. They are constructed from the indicator vectors of the triplets. For each of the $3q$ positions of the vector such a tree has four levels – together called a *layer* – that encode either zero or one, where the encodings of zero and one are shown in Figure 6.3.3 (a) and (b). Figures 6.3.3 (c) and (d) show how layers are stacked using *sibling trees* (the sibling tree of a zero-tree is identical to that of a one-tree shown in the figure). We have chosen the zero-trees and one-trees such that both have the same number of codes and occupy the same bandwidth, but are still different.


(a) The zero-tree


(b) The one-tree


(c) A layer, consisting of a one-tree and its sibling



(d) Stacking layers

**Figure 6.3.3:** *Encoding of zero and one*

The receiver trees are supposed to receive all codes in the chosen triplet trees. These codes fit exactly in the free positions, if and only if the chosen triplets form a 3DM, i.e., if their indicator vectors sum up to the all-one vector. This equivalence directly tells us, how many codes the trees must receive on which level: On every layer the receiver trees must take $q-1$ zero-trees, $1$ one-tree and $q$ sibling-trees, so that on the four levels of each layer there must be exactly $0, q+1, 5q-3$ resp. $q+2$ free codes (plus $q$ extra codes on the very last level). For each one of these $3q \cdot 7q + q = 21q^2 + q$ codes we build one receiver tree. The receiver tree for a code on level $l'$ is a tree with root on level $l_{\mathrm{start}}$ with the following properties. It has one free position on level $l'$, the rest of the tree is full and it contains $21q+2$ codes, i.e., one more code than a triplet tree. Clearly, such a tree always exists in our situation.

Finally, the fill trees are trees that are completely full and have one more code than the receiver trees. They fill up the level $l_{\mathrm{start}}$ in the sibling-tree of the token tree.

An interesting question is, whether this transformation from 3DM to the one-step offline CA can be done in polynomial time. This depends on the input encoding of our problem. We consider the following natural encodings:

- a zero-one vector that specifies for every node of the tree whether there is a code or not,

- a sparse representation of the tree, consisting only of the positions of the assigned codes.

Obviously, the transformation cannot be done in polynomial time for the first input encoding, because the generated tree has $2^{12q+l_{\text{start}}}$ leaves. For the second input encoding the transformation is polynomial, because the total number of generated codes is polynomial in $q$, which is polynomial in the input size of 3DM. Besides, we should rather not expect an NP-completeness proof for the first input encoding, because this would suggest—together with the dynamic programming algorithm in this paper—$n^{O(\log n)}$-algorithms for all problems in NP.

We now state the crucial property of the construction in a lemma:

**Lemma 6.6** *Let $M$ be an input for $3DM$ and $\phi$ the transformation described above. Then $M \in 3DM$ if and only if $\phi(M)$ can be done with $\alpha = 21q^2+2q+1$ reassignments.*

**Proof.** Assume there is a 3DM $M' \subset M$. Now consider the reassignment that assigns the code insertion to the root of the token tree, and the tokens to the $q$ roots of the triplet trees that correspond to the triplets in $M'$. We know that the corresponding indicator vectors sum up to the all-one vector, so that all codes in the triplet trees that need to be reassigned fit exactly in the receiver trees. In total, $1 + q + (21q + 1)q = \alpha$ codes are (re-)assigned.

Now assume there is no matching. This implies that every subset of $q$ indicator vectors does not sum up to the all-one vector. Assume for a contradiction that we can still serve $\phi(M)$ with at most $\alpha$ reassignments. Clearly, the initial code insertion must be assigned to the left tree, otherwise we need too many reassignments. The $q$ tokens must not trigger more than $(21q + 1)q$ additional reassignments. This is only possible if they are all assigned to triplet trees, which triggers exactly $(21q+1)q$ necessary reassignments. Now no more reassignments are allowed. But we know that the corresponding $q$ indicator vectors do not sum up to the all-one vector, in particular, there must be one position that sums up to zero. In the layer of this position the receiver-trees receive $q$ zero-trees and no one-tree instead of $q - 1$ zero trees and one one-tree. But by construction the extra zero-tree cannot be assigned to the remaining receiver trees of the one-tree. It cannot be assigned somewhere else either, because this would cause an extra reassignment on a different layer. This is why an extra reassignment is needed, which brings the total number of (re-)assignments above $\alpha$. □

One could wonder whether an optimal one-step offline CA algorithm can ever attain the configuration that we construct for the transformation. We prove below in Corollary 6.9 that we can force such an algorithm into any configuration. To sum up, we have shown the following theorem:

**Theorem 6.7** *The decision variant of the one-step offline CA is NP-complete for an input given by a list of positions of the assigned codes and the code insertion level.*

**Enforcing arbitrary configurations** We show that for any configuration $C'$ and any optimal one-step algorithm $A$ there exists a sequence of code insertions and deletions of polynomial length, so that $A$ ends up in $C'$ on that sequence. Notice that any optimal one-step algorithm reassigns codes only if it has to, i.e., it places a code without any additional reassignments if this is possible, and it does not reassign after a deletion. The result even applies to any algorithm $A$ with these properties.

We start with the empty configuration $C_0$. The idea of the proof is to take a detour and first attain a full-capacity configuration $C_{\text{full}}$ and then go from there to $C'$. The second step is easy: It suffices to delete all the codes in $C_{\text{full}}$ that are not in $C'$; $A$ must not do any reassignments during these deletions. First, we show that we can force $A$ to produce an arbitrarily chosen configuration $C_{\text{full}}$ that uses the full tree capacity.

**Theorem 6.8** *Any one-step optimal algorithm $A$ can be led to an arbitrary full configuration $C_{\text{full}}$ with $n$ assigned codes by a request sequence of length $m < 3n$.*

**Proof.** Recall that $h$ denotes the height of the code tree. We proceed top-down: On every level $l'$ with codes in $C_{\text{full}}$ we first fill all its unblocked positions using at most $2^{h-l'}$ code insertions on level $l'$. $A$ just fills $l'$ with codes. Then we delete all codes on $l'$ that are not in $C_{\text{full}}$ and proceed recursively on the next level.

We have to argue that we do not insert too many codes in this process. To see this, observe that we only insert and delete codes above the $n$ codes in $C_{\text{full}}$, and we do this at most once in every node. Now if we consider the binary tree the leaves of which are the codes in $C_{\text{full}}$, then we see that the number of insert operation is bounded by $n + n - 1$, where $n - 1$ is the number of inner nodes of this tree. Together with the deletions we obtain the statement. □

We come back to arbitrary configurations.

**Corollary 6.9** *Given a configuration tree $C'$ of height $h$ with $n$ assigned codes, there exists a sequence $\sigma_1, \ldots, \sigma_m$ of code insertions and deletions of length $m < 4nh$ that forces $A$ into $C'$.*

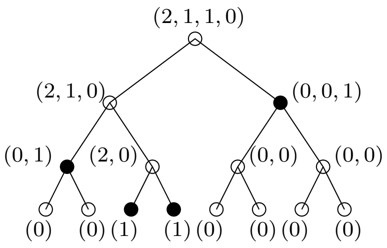


**Figure 6.3.4:** *Node signatures.*

**Proof.** We define $C_{\text{full}}$ from $C'$ by filling the gap trees in $C'$ (as high as possible) with codes. Each code causes at most one gap tree on every level, hence we need at most $h$ codes to fill the gap trees for one code. Altogether, we need at most $nh$ codes to fill all gap trees. According to Theorem 6.8, we can construct a sequence of length $m < 3nh$ that forces $A$ into $C_{\text{full}}$. Then we delete the padding codes and end up in $C'$. Altogether we need at most $4nh$ requests for code insertion and deletion. □

## 6.3.3 Exact $n^{O(h)}$ Algorithm

In this section we solve the one-step offline CA problem optimally using a dynamic programming approach. The key idea of the resulting algorithm is to store the right information in the nodes of the tree and to build it up in a bottom-up fashion.

To make this construction precise, we define a *signature* of a subtree $T_v$ with root $v$ as an $l(v) + 1$-dimensional vector $s^v = (s_0^v, \ldots, s_{l(v)}^v)$, in which $s_i^v$ is the number of codes in $T_v$ on level $i$, see Figure 6.3.4. A signature $s$ is *feasible* if there exists a subtree $T_v$ with a feasible code assignment that has signature $s$. The information stored in every node $v$ of the tree consists of a table, in which all possible feasible signatures of an arbitrary tree of height $l(v)$ are stored together with their *cost for* $T_v$. Here the cost of such a signature $s$ for $T_v$ (usually $s \neq s^v$) is defined as the minimum number of codes in $T_v$ that have to move away from their old position in order to attain some tree $T'_v$ with signature $s$. To attain $T'_v$ it can be necessary to also move into $T_v$ codes from other subtrees but we do not count these movements for the cost of $s$ for $T_v$.

Given a code tree $T$ with all these tables computed, one can compute the cost of any single code insertion from the table at the root node $r$: Let $s^r = (s_0^r, \ldots, s_h^r)$ be the signature of the whole code tree before insertion, then the cost of an insertion at level $l$ is the cost of the signature $(s_0^r, \ldots, s_l^r + 1, \ldots, s_h^r)$ in this table plus one. This follows because the minimum number of codes that are moved away from their positions in $T$ is equal to the number of reassignments minus one.

The computation of the tables starts at the leaf level, where the cost of the one-dimensional signatures is trivially defined. At any node $v$ of level $l(v)$ the cost $c(v, s)$ of signature $s$ for $T_v$ is computed from the cost incurred in the left subtree $T_l$ of $v$ plus the cost incurred in the right subtree $T_r$ plus the cost at $v$. The costs $c(l, s')$ and $c(r, s'')$ in the subtrees come from two feasible signatures with the property $s = (s'_0 + s''_0, \ldots, s'_{l(v)-1} + s''_{l(v)-1}, s_{l(v)})$. Any pair $(s', s'')$ of such signatures corresponds to a possible configuration after the code insertion. The best pair for node $v$ gives $c(v, s)$. Let $s^v = (s_0^v, \ldots, s_{l(v)}^v)$ be the signature of $T_v$, then it holds that

$$c(v,s) = \begin{cases} c(l,(0,\ldots,0)) + c(r,(0,\ldots,0)) & \text{for } s_{l(v)} = 1 \\ \min_{\{s',s'' \mid (s',0)+(s'',0)=s\}} & \\ \qquad\qquad (c(l,s') + c(r,s'')) & \text{for } s_{l(v)} = 0, s_{l(v)}^v = 0 \\ 1 & \text{for } s_{l(v)} = 0, s_{l(v)}^v = 1 \end{cases} .$$

The costs of all signatures $s$ for $v$ can be calculated simultaneously by combining the two tables in the left and right children of $v$. Observe for the running time that the number of feasible signatures is bounded by $(n+1)^h$ because there cannot be more than $n$ codes on any level. The time to combine two tables is $O(n^{2h})$, thus the total running time is bounded by $O(2^h \cdot n^{2h})$.

**Theorem 6.10** *The one-step offline CA can be optimally solved in time $O(2^h \cdot n^{2h})$ and space $O(h \cdot n^h)$.*

### 6.3.4 $h$-Approximation Algorithm

In this section we propose and analyze a greedy algorithm for one-step offline CA, i.e., for the problem of assigning an initial code insertion $c_0$ into a code tree $T$ with given code assignment $F$. The idea of the greedy algorithm $A_{\text{greedy}}$ is to assign the code $c_0$ to the root $g$ of the subtree $T_g$ that contains the fewest assigned codes among all possible subtrees. From Lemma 6.3 we know that no optimal algorithm reassigns codes on higher levels than the current one; hence the possible subtrees are those that do not contain assigned codes on or above their root. Then the greedy algorithm takes all codes in $T_g$ (denoted by $\Gamma(T_g)$) and reassigns them recursively in the same way, always processing codes of higher level first.

At every time $t$ algorithm $A_{\text{greedy}}$ has to assign a set $C_t$ of codes into the current tree $T^t$. Initially, $C_0 = \{c_0\}$ and $T^0 = T$. For a given position, code or code insertion $c$, its level is denoted by $l(c)$.

**Algorithm 6**: $A_{\text{greedy}}$

```
$C_0 \leftarrow \{c_0\}$; $T^0 \longleftarrow T$
$t \longleftarrow 0$
while $C_t \neq \emptyset$ do
    $c_t \longleftarrow$ element with highest level in $C_t$
    $g \longleftarrow$ the root of a subtree $T_g^t$ of level $l(c_t)$ with
    the fewest codes in it and no code on or above its root
    // assign $c_t$ to position $g$
    $T^{t+1} \longleftarrow (T^t \setminus \Gamma(T_g^t)) \cup \{g\}$
    $C_{t+1} \longleftarrow (C_t \cup \Gamma(T_g^t)) \setminus \{c_t\}$
    $t \longleftarrow t + 1$
end
```

In [93] a similar algorithm is proposed as a heuristic for one-step offline CA. We prove that $A_{\text{greedy}}$ has approximation ratio $h$. This bound is asymptotically tight: In the following examples we show that $A_{\text{greedy}}$ can be forced to use $\Omega(h) \cdot \text{OPT}$ (re-)assignments (see Figure 6.3.5), where OPT refers to the optimal number of (re-)assignments. A new code $c_{\text{new}}$ is assigned by $A_{\text{greedy}}$ to the root of $T_0$ (which contains the least number of codes). The two codes on level $l-1$ from $T_0$ are reassigned as shown in the figure, one code can be reassigned into $T_{\text{opt}}$ and the other one goes recursively into $T_1$. In total, $A_{\text{greedy}}$ does $2 \cdot l + 1$ (re-)assignments while the optimal algorithm assigns $c_{\text{new}}$ into the root of $T_{\text{opt}}$ and reassigns the three codes from the leaf level into the trees $T_1, T_2, T_3$, requiring only $4$ (re-)assignments. Obviously, for this example $A_{\text{greedy}}$ is not better than $(2l+1)/4$ times the optimal. In general, $l$ can be $\Omega(h)$.

For the upper bound we compare $A_{\text{greedy}}$ to the optimal algorithm $A_{\text{opt}}$. $A_{\text{opt}}$ assigns $c_0$ to the root of a subtree $T_{x_0}$, the codes from $T_{x_0}$ to some other subtrees, and so on. Let us call the set of subtrees to the root of which $A_{\text{opt}}$ moves codes the *opt-trees*, denoted by $\mathcal{T}_{\text{opt}}$, and the arcs that show how $A_{\text{opt}}$ moves the codes the *opt-arcs* (cf. Figure 6.3.6). By $V(\mathcal{T}_{\text{opt}})$ we denote the set of nodes in $\mathcal{T}_{\text{opt}}$.

A sketch of the proof is as follows. First, we show that in every step $t$ $A_{\text{greedy}}$ has the possibility to assign the codes in $C_t$ into positions inside the opt-trees. This possibility can be expressed by a code mapping $\phi_t : C_t \to V(\mathcal{T}_{\text{opt}})$. The

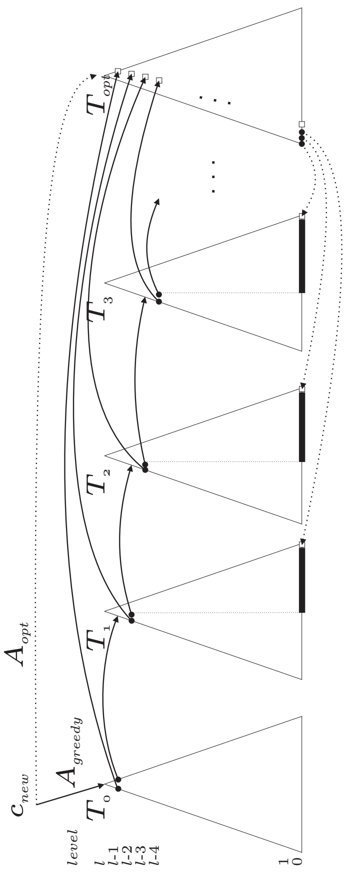


**Figure 6.3.5:** *Lower bound example for $A_{greedy}$.*

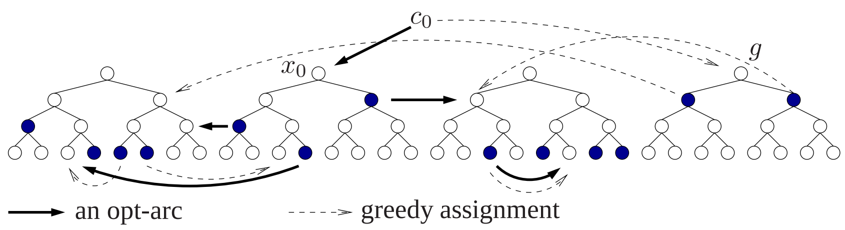


**Figure 6.3.6:** *$A_{opt}$ moves codes to assign a new code $c_0$ using opt-arcs. The opt-trees are subtrees to the root of which $A_{opt}$ moves codes. Here, the cost of the optimal solution is 5. The greedy algorithm has cost 6.*

key-property is now that in every step of the algorithm there is the theoretical choice to complete the current assignment using the code mapping $\phi_t$ and the opt-arcs as follows: Use $\phi_t$ to assign the codes in $C_t$ into positions in the opt-trees and then use the opt-arcs to move codes out of these subtrees of the opt-trees to produce a feasible code assignment. We will see that this property is enough to ensure that $A_{\text{greedy}}$ incurs a cost of no more than OPT on every level.

In the process of the algorithm it can happen that we have to change the opt-arcs in order to ensure the existence of $\phi_t$. To model the necessary changes we introduce $\alpha_t$-arcs that represent the changed opt-arcs after $t$ steps of the greedy algorithm.

To make the proof-sketch precise, we need the following definitions:

**Definition 6.11** *Let $\mathcal{T}_{opt}$ be the set of the opt-trees for a code insertion $c_0$ and let $T^t$ (together with its code assignment $F^t$) be the code tree after $t$ steps of the greedy algorithm $A_{greedy}$. An $\alpha$-*mapping *at time $t$ is a mapping $\alpha_t : M_{\alpha_t} \rightarrow V(\mathcal{T}_{opt})$ for some $M_{\alpha_t} \subseteq F^t$, such that $\forall v \in M_{\alpha_t} : \; l(v) = l(\alpha_t(v))$ and $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ is a code assignment.*

Note that in general $F^t$ is not a code assignment for all codes since it does not contain the codes in $C^t$. The set $\alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$ represents the resulting code assignment (that again does not contain the codes in $C^t$) after reassignment of the codes $M_{\alpha_t} \subseteq F^t$ by $\alpha_t$.

**Definition 6.12** *Let $T^t$ be a code tree, $x, y$ be positions in $T^t$ and $\alpha_t$ be an $\alpha$-mapping. We say that $y$* depends *on $x$ in $T^t$ and $\alpha_t$, if there is a path from $x$ to $y$ using only tree-edges from a parent to a child and $\alpha_t$-arcs. By $\mathrm{dep}_t(x)$ we denote the set of all positions $y$ that depend on $x$ in $T^t$ and $\alpha_t$. We say that an $\alpha_t$-arc $(u, v)$ depends on $x$ if $u \in \mathrm{dep}_t(x)$.*

For an illustration of this definition, see Figure 6.3.7.

**Definition 6.13** *At time $t$ a pair $(\phi_t, \alpha_t)$ of a code mapping $\phi_t : C_t \rightarrow V(\mathcal{T}_{opt})$ and an $\alpha$-mapping $\alpha_t$ is called an* independent mapping *for $T^t$, if the following properties hold:*

1. *$\forall c \in C_t$ the levels of $\phi_t(c)$ and $c$ are the same (i.e. $l(c) = l(\phi_t(c))$).*

2. *$\forall c \in C_t$ there is no code in $T^t$ at or above the roots of the trees in $\mathrm{dep}_t(\phi_t(c))$.*

3. *the code movements realized by $\phi_t$ and $\alpha_t$ (i.e. the set $\phi_t(C_t) \cup \alpha_t(M_{\alpha_t}) \cup (F^t \setminus M_{\alpha_t})$) form a code assignment.*

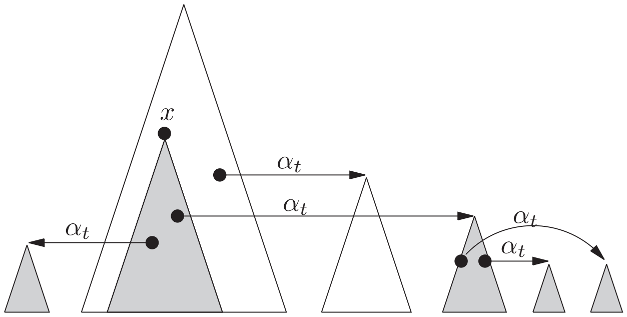


**Figure 6.3.7:** *The filled subtrees represent all the positions that depend on $x$.*

4. *every node in the domain $M_{\alpha_t}$ of $\alpha_t$ is contained in $\mathrm{dep}_t(\phi_t(C_t))$ (i.e., no unnecessary arcs are in $\alpha_t$).*

Note that $\phi_t$ and $\alpha_t$ can equivalently be viewed as functions and as collections of arcs of the form $(c, \phi_t(c))$ and $(u, \alpha_t(u))$, respectively. We write $\mathrm{dep}_t(\phi_t(C_t))$ for the set $\bigcup_{c \in C_t} \mathrm{dep}_t(\phi_t(c))$. Note that if a pair $(\phi_t, \alpha_t)$ is an independent mapping for $T^t$, then $\mathrm{dep}_t(\phi_t(C_t))$ is contained in opt-trees and every node in $\mathrm{dep}_t(\phi_t(C_t))$ can be reached on exactly one path from $C_t$ (using one $\phi_t$-arc and an arbitrary sequence of tree-arcs, which always go from parent to child, and $\alpha_t$-arcs from a code $c \in \Gamma(T^t)$ to $\alpha_t(c)$).

Now we state a lemma that is crucial for the analysis of the greedy strategy.

**Lemma 6.14** *For every set $C_t$ in algorithm $A_{greedy}$ the following invariant holds:*

$$\textit{There is an independent mapping } (\phi_t, \alpha_t) \textit{ for } T^t. \tag{6.3.1}$$

**Proof.** The proof is done by induction on $t$ and shows how to construct an independent mapping $(\phi_{t+1}, \alpha_{t+1})$ from $(\phi_t, \alpha_t)$ by case analysis. The detailed proof can be found in [49] and will be included in the thesis of Matúš Mihaľák. □

We remark that Lemma 6.14 actually applies to all algorithms that work level-wise top-down and choose a subtree $T_g^t$ for each code $c_t \in C_t$ arbitrarily under the condition that there is no code on or above the position $g$.

We can express the cost of the optimal solution by the opt-trees:

**Lemma 6.15** *(a) The optimal cost is equal to the number of assigned codes in the opt-trees plus one, and (b) it is equal to the number of opt-trees.*

**Proof.** Observe for (a) that $A_{\text{opt}}$ moves all the codes in the opt-trees and for (b) that $A_{\text{opt}}$ moves one code into the root of every opt-tree. □

**Theorem 6.16** *The algorithm $A_{\textit{greedy}}$ has an approximation ratio of $h$.*

**Proof.** $A_{\text{greedy}}$ works level-wise top-down. We show that on every level $l$ the greedy algorithm incurs a cost of at most OPT. Consider a time $t_l$ where $A_{\text{greedy}}$ is about to start a new level $l$, i.e., before $A_{\text{greedy}}$ assigns the first code on level $l$. Assume that $C_{t_l}$ contains $q_l$ codes on level $l$. Then $A_{\text{greedy}}$ places these $q_l$ codes in the roots of the $q_l$ subtrees on level $l$ containing the fewest codes. The code mapping $\phi_{t_l}$ that is part of the independent mapping $(\phi_{t_l}, \alpha_{t_l})$, which exists by Lemma 6.14, maps each of these $q_l$ codes to a different position in the opt-trees. Therefore, the total number of codes in the $q_l$ subtrees with roots at $\phi_{t_l}(c)$ (for $c$ a code on level $l$ in $C_{t_l}$) is at least the number of codes in the $q_l$ subtrees chosen by $A_{\text{greedy}}$. Combining this with Lemma 6.15(a), we see that on every level $A_{\text{greedy}}$ incurs a cost (number of codes that are moved away from their position in the tree) that is at most $A_{\text{opt}}$'s total cost. □

## 6.4 Fixed Parameter Tractability of the Problem

In this section we consider the fixed parameter tractability of the parameterized one-step offline Code Assignment problem, see also Definition 2.2. Parameterized problems are described by languages $L \subseteq \Sigma^* \times \mathbb{N}$. If $(x, k) \in L$, we refer to $k$ as the parameter.

We assume that our problem is given by a pair $(x, k)$, where $x$ encodes the code insertion on level $l$ and the current code assignment and $k$ is the parameter. We assume the encoding of the code assignment in the zero-one vector form $x_1, \ldots, x_{2^{h+1}-1}$ saying for every node of the tree whether there is an assigned code. Denote for the purpose of this section by $n$ the size of the input, i.e., $n := |x| = 2^{h+1} - 1$.

We consider various variants of parameters for the problem. The most natural ones are the number of moved codes $m$ or the level $l$ of the code insertion. To show the fixed parameter tractability, we reuse the ideas of the exact dynamic programming algorithm, which stores at every node a table of all possible signatures.

We first show that the problem is fixed parameter tractable, if the parameters are both $m$ and $l$, i.e., we show an algorithm solving the problem in time $O(f(m, l)p(n))$ for some polynomial $p(n)$.

For a code insertion into the code tree for level $l$, we know that we only move codes from lower levels than $l$. Hence, when building the tables at nodes, we need to consider only those signatures that differ on levels $0, \ldots, l-1$ from the signature of the current subtree. From the assumption that we move at most $m$ codes, we have that on each of these levels, the considered signature can differ by at most $m$. Hence, the number of considered signatures in every node is at most $(2m+1)^l$. To compute all the tables, we need to combine all the tables from the children nodes, i.e., we have to consider $(2m+1)^{2l}$ pairs for every node. From this we get a running time of $O(2^h(2m+1)^{2l})$, which is certainly of the form $f(m,l)p(n)$.

For the case, where we have only $l$ as the parameter, we immediately get that we move from every subtree $T_v$ at most $2^l$ codes, hence we bound the number of codes moved in every subtree by a parameter (we note that we did not bound the overall number of moved codes) $m = 2^l$.

Consider now the case, where only $m$ is the parameter. Since we move at most $m$ codes within the tree, we know that at most $m$ codes come into the subtree and at most $m$ go away from the subtree. Hence, assigning for each such possibility a level out of $0, \ldots, l$, we get an upper bound of at most $(l+1)^{2m}$ signatures to be considered at every node on level $l$. Since $l+1 \leq h$ for $l = 0, \ldots, h-1$ we get at every node at most $h^{2m} = \log n^{2m}$ signatures. From [108] we can use the inequality $(\log n)^m \leq (3m \log m)^m + n$ to express the size of each table in the form $g(m) + n$. To compute the table for every node, we need time $n(g(m)+n)^2$ which is certainly of the form $f(m)p(n)$.

We summarize the results of this section in the following theorem.

**Theorem 6.17** *The one-step offline CA problem is fixed parameter tractable for the following parameters:*

- *the level $l$ of the code insertion and*
- *the number $m$ of moved codes.*

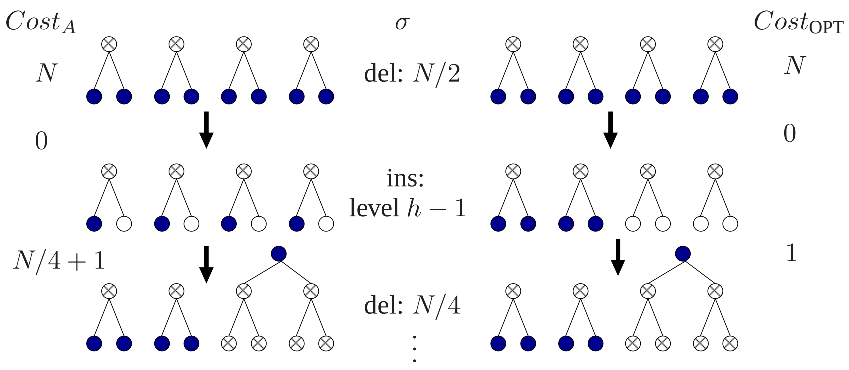


**Figure 6.5.1:** *Lower bound for the online assignment problem.*

## 6.5 Online CA

Here we study the CA problem in an online setting, see Chapter 2. We assume that insertions do not exceed the total available bandwidth.

In the case of the online CA problem the requests for code insertions and deletions must be handled one after another, i.e., the $i$th request must be served before the $i+1$st request is known. An online algorithm ALG for the CA problem is $c$-*competitive* if there is a constant $\alpha$ such that for all finite input sequences $I$,

$$\text{ALG}(I) \leq c \cdot \text{OPT}(I) + \alpha \ .$$

We give a lower bound on the competitive ratio, analyze several algorithms and present a resource augmented algorithm with constant competitive ratio.

**Theorem 6.18** *No deterministic algorithm $A$ for the online CA problem can be better than* $1.5$*-competitive.*

**Proof.** Let $A$ be any deterministic algorithm for the problem. Consider $N$ leaf insertions. The adversary can delete $N/2$ codes (every second) to get the situation in Figure 6.5.1.

Then a code insertion at level $h-1$ causes $N/4$ code reassignments. We can proceed with the left subtree of full leaf codes recursively and repeat this process $(\log_2 N - 1)$ times. The optimal algorithm $A_{\text{opt}}$ assigns the leaves in the first step in such a way that it does not need any reassignment at all. Thus, $A_{\text{opt}}$ needs $N + \log_2 N - 1$ code assignments. Algorithm $A$ needs $N + T(N)$ code assignments, where $T(N) = 1+N/4+T(N/2)$ and $T(2) = 0$. Clearly, $T(N) = \log_2 N - 1 + \frac{N}{2}(1-2/N)$. If $C_A \leq c \cdot C_{\text{OPT}}$ then $c \geq \frac{3N/2+\log_2 N-2}{N+\log_2 N-1} \longrightarrow_{N\to\infty} 3/2$. □

### 6.5.1 Compact representation algorithm

This algorithm maintains the codes in the tree $T$ sorted and compact. For a given node/code $v \in T$ we denote by $l(v)$ its level and by $w(v)$ its string representation, i.e., the description of the path from the root to the node/code, where $0$ means left child and $1$ right child as in Section 6.2.1. We use the lexicographic ordering when comparing two string representations. By $U$ we denote the set of unblocked nodes of the tree. We maintain the following invariants:

$$\forall \text{ codes } u, v \in F: \quad l(u) < l(v) \Rightarrow w(u) < w(v), \tag{6.5.1}$$

$$\forall \text{ nodes } u, v \in T: \quad l(u) \leq l(v) \ \wedge\ u \in F \ \wedge\ v \in U \\ \Rightarrow\ w(u) < w(v). \tag{6.5.2}$$

This states that we want to keep the codes in the tree ordered from left to right according to their levels (higher level assigned codes are to the right of lower level assigned codes) and compact (no unblocked code to the left of any assigned code on the same level).

In the following analysis we show that this algorithm is not worse than $O(h)$ times the optimum for the offline version. We also give an example that shows that the algorithm is asymptotically not better than this.

**Theorem 6.19** *Algorithm $A_{compact}$ satisfying invariants (6.5.1) and (6.5.2) performs at most $h$ code reassignments per insertion or deletion.*

**Proof.** We show that for both insertion and deletion we need to make at most $h$ code reassignments. When inserting a code on level $l$, we look for the rightmost unassigned position on that level that maintains the invariants (6.5.1) and (6.5.2) among codes on level $0, \ldots, l$. Either the found node is not blocked, so that we do not move any codes, or the code is blocked by some assigned code on a higher level $l' > l$ (see Figure 6.5.2). In the latter case we remove this code to free the position for level $l$ and handle the new code insertion on level $l'$ recursively. Since we move at most one code at each level and we have $h$ levels, we move at most $h$ codes for each insertion.

Handling the deletion operation is similar, we just move the codes from right to left in the tree and move at most one code per level to maintain the invariants. □

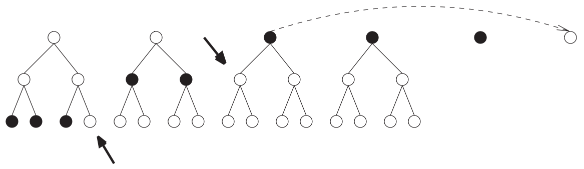


**Figure 6.5.2:** *For a code insertion, Algorithm $A_{compact}$ finds the leftmost position (blocked or unblocked) that has no code on it and no code in the subtree below it. It reassigns at most one code at every level.*

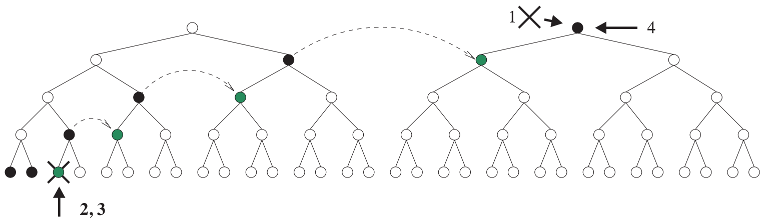


**Figure 6.5.3:** *Code assignments for levels $0, 0, 1, 2, 3, 4, \ldots, h-1$ and four consecutive operations: 1. DELETE(h-1), 2. INSERT(0), 3. DELETE(0), 4. INSERT(h-1).*

**Corollary 6.20** *Algorithm $A_{compact}$ satisfying invariants (6.5.1) and (6.5.2) is $O(h)$-competitive.*

**Proof.** In the sequence $\sigma = \sigma_1, \ldots, \sigma_m$ the number of deletions $d$ must be smaller or equal to the number $i$ of insertions, which implies $d \leq m/2$. The cost of any optimal algorithm is then at least $i \geq m/2$. On the other hand, $A_{\text{compact}}$ incurs a cost of at most $m \cdot h$, which implies that it is $O(h)$-competitive. □

**Theorem 6.21** *Any algorithm $A_I$ satisfying invariant* $(6.5.1)$ *is* $\Omega(h)$*-competitive.*

**Proof.** Consider the sequence of code insertions on levels $0, 0, 1, 2, 3, 4, \ldots, h-1$. For these insertions, there is a unique code assignment satisfying invariant $(6.5.1)$, see Figure 6.5.3. Consider now two requests—deletion of the code at level $h-1$ and insertion of a code on level $0$. Then $A_I$ has to move every code on level $l \geq 1$ to the right to create space for the code assignment on level $0$ and maintain the invariant $(6.5.1)$. This takes $1$ code assignment and $h-2$ reassignments. Consider as the next requests the deletion of the third code on level zero and an insertion on level $h-1$. Again, to maintain invariant $(6.5.1)$,

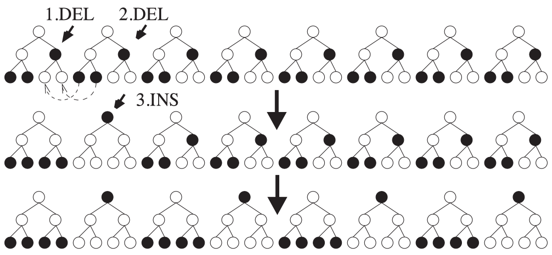


**Figure 6.5.4:** *Requests that a greedy strategy cannot handle efficiently.*

$A_I$ has to move every code on level $l \geq 1$ to the left. This takes again 1 code assignment and $h-2$ reassignments. An optimal algorithm can handle these four requests with two assignments, since it can assign the third code on level zero in the right subtree, where $A_I$ assigns the code on level $h-1$. Repeating these four requests $k$ times, the total cost of the algorithm $A_I$ is then $C_I = h+1+2k(h-1)$, whereas OPT has $C_{\text{OPT}} = h + 1 + 2k$. As $k$ goes to infinity, the ratio $C_A/C_{\text{OPT}}$ becomes $\Omega(h)$. □

### 6.5.2 Greedy strategies

Assume we have a deterministic algorithm $A$ that solves the one-step offline CA problem. This $A$ immediately leads to a greedy online-strategy. As an optimal algorithm breaks ties in an unspecified way, the online-strategy can vary for different optimal one-step offline algorithms.

**Theorem 6.22** *Any deterministic greedy online-strategy, i.e. a strategy that minimizes the number of reassignments for every insertion and deletion, is $\Omega(h)$ competitive.*

**Proof.** Assume that $A$ is a fixed, greedy online-strategy. First we insert $N/2$ codes at level 1. As $A$ is deterministic we can now delete every second level-1 code, and insert $N/2$ level-0 codes. This leads to the situation depicted in Figure 6.5.4. Then we delete two codes at level $l = 1$ (as $A$ is deterministic it is clear which codes to delete) and immediately assign a code at level $l + 1$. As it is optimal (and up to symmetry unique) the algorithm $A$ moves two codes as depicted. The optimal strategy arranges the level-1 codes in a way that it does not need any additional reassignments. We proceed in this way along level 1 in the first round, then left to right on level 2 in a second round, and continue

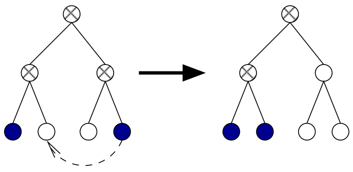


**Figure 6.5.5:** *Reassignment of one code reduces the number of blocked codes from 3 to 2.*

toward the root. Algorithm $A$ moves $N/4$ codes in the first round and assigns $N/2^3$ codes. In general, in every round $i$ the algorithm moves $N/4$ level-0 codes and assigns $N/2^{i+2}$ level-$i$ codes. Altogether, the greedy strategy needs $O(N) + (N/4)\Omega(\log N) = \Omega(N \log N)$ (re-)assignments, whereas the optimal strategy does not need any reassignments and only $O(N)$ assignments. $\square$

## 6.5.3 Minimizing the number of blocked codes

The idea of minimizing the number of blocked codes is mentioned in [109] but not analyzed at all. In every step the algorithm tries to satisfy the invariant:

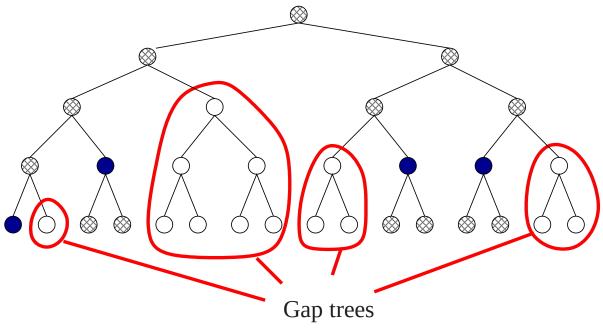
$$\text{The number of blocked codes in } T \text{ is minimum.} \tag{6.5.3}$$

In Figure 6.5.5 we see a situation that does not satisfy the invariant (6.5.3). Moving a code reduces the number of blocked codes by one. We prove that this approach is equivalent to minimizing the number of gap trees on every level (Lemma 6.24). Recall that a gap tree is a maximal subtree of unblocked codes.

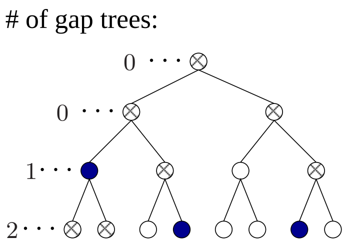
**Definition 6.23** *The level of the root of a gap tree is called the* level of the gap tree. *The vector $q = (q_0, \ldots, q_h)$, where $q_i$ is the number of gap trees on level $i$, is called the* gap vector *of the tree $T$.*

See Figure 6.5.6 for an example of the definition. Invariant (6.5.3) implies that there is at most one gap tree on every level. If there are two gap trees on level $l$ we can move the sibling tree of one of the gap trees to fill the other gap tree, reducing the number of blocked codes by at least one (see Figure 6.5.5). The following lemma states that there is indeed an equivalence between having a minimal number of gap trees and having a minimum number of blocked codes.

**Lemma 6.24** *Let $T$ be a code tree for requests vector $\sigma$. Then $T$ has at most one gap tree on every level if and only if $T$ has a minimum number of blocked codes.*



(a) Gap trees—maximal subtrees of unblocked codes



$q = (2, 1, 0, 0)$ — gap vector

(b) Gap vector

**Figure 6.5.6:** *Definition of gap trees and gap vector*

**Proof.** Suppose $T$ has a minimum number of blocked codes. If $T$ had two gap trees $T_u, T_v$ on level $l$, then we could move the codes in sibling tree $T_{u'}$ of $u$ into $T_v$, which would save at least one blocked code (the parent of $u$ would become unblocked), a contradiction.

Now suppose that $T$ has at most one gap tree on every level. We will show that this property uniquely defines the gap vector for the given request vector $\sigma$. Then we show that all code assignments with the same gap vector have the same number of blocked codes. The two statements together prove the second implication.

For the first statement observe that the free bandwidth capacity of $T$ can be expressed as

$$\text{cap} = \sum_{i=0}^{h} q_i 2^i \quad .$$

As $q_i \leq 1$, the gap vector is the binary representation of the number cap and therefore the gap vector $q$ is uniquely defined by $\sigma$ for trees with at most one gap tree per level. For the second statement note that the gap vector determines also the number of blocked codes:

$$\#\text{blocked codes} = (2^{h+1} - 1) - \sum_{i=0}^{h} q_i (2^{i+1} - 1) \quad .$$

Thus, every tree for requests $\sigma$ with at most one gap tree at every level has the same number of blocked codes. □

Now we are ready to define the algorithm $A_{\text{gap}}$ (Algorithm 7). As we will show, on insertions $A_{\text{gap}}$ never needs any extra reassignments.

**Algorithm 7**: Algorithm $A_{\text{gap}}$

```
invariant: The number of blocked codes is minimum.

Insert:
Assign the new code into the smallest gap tree where it fits.
Delete:
delete code from tree
if deletion creates a second gap tree on some level then
    move one of their sibling subtrees into the second gap tree
    Treat all newly created second gap trees on higher levels recursively.
end
```

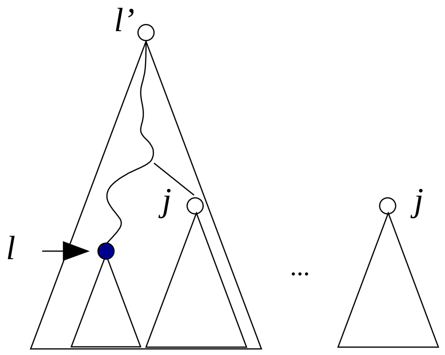


**Figure 6.5.7:** *Two gap trees on a lower level than $l'$ violate the minimum chosen height of the gap tree.*

**Lemma 6.25** *The algorithm $A_{gap}$ has always a gap tree of sufficient height to assign a code on level $l$ and at every step the number of gap trees at every level is at most one.*

**Proof.** We know that there is sufficient capacity to serve the request, i.e., $cap \geq 2^l$. We also know that $cap = \sum_i q_i 2^i$. Since $q_i \leq 1$ for all $i$, there exists a gap tree on level $j \geq l$.

Next, consider an insertion into the smallest gap tree of level $l'$ where the code fits. New gap trees can occur only on levels $j$, $l \leq j < l'$ and only within the gap tree on level $l'$. Also, at most one new gap tree can occur on every level. Suppose that after creating a gap tree on level $j$, we have more than one gap tree on this level. Then, since $j < l'$, we would assign the code into this smaller gap tree, which contradicts our assumption (Figure 6.5.7). Therefore, after an insertion there is at most one gap tree on every level.

Consider now a deletion of a code. The nodes of the subtree of that code become unblocked, i.e., they belong to some gap tree. At most one new gap tree can occur in the deletion operation (and some gap trees may disappear). Thus, when the newly created gap tree is the second one on the level, we fill the gap trees and then we recursively handle the newly created gap tree on a higher level. In this way the gap trees are moved up. Because we cannot have two gap trees on level $h - 1$, we end up with a tree with at most one gap tree on each level. □

The result shows that the algorithm is optimal for insertions only. It does not need any extra code movements, contrary to the compact representation algorithm. Similarly to the compact representation algorithm, this algorithm is

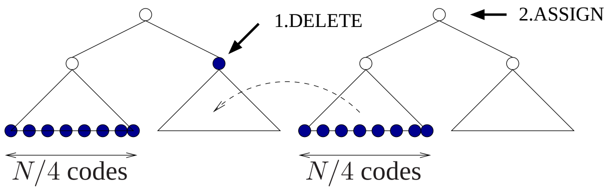


**Figure 6.5.8:** *Worst case number of movements for algorithm $A_{gap}$.*

$\Omega(\log N)$-competitive.

**Theorem 6.26** *Algorithm $A_{gap}$ is $\Omega(h)$-competitive.*

**Proof.** The proof is basically identical with the proof of Theorem 6.22. □

The algorithm $A_{\text{gap}}$ has even a very bad worst case number of code movements. Consider the four subtrees on level $h-2$, where the first one has $N/4$ leaf codes inserted, its sibling has a code on level $h-2$ inserted and the third subtree has again $N/4$ leaf codes inserted (Figure 6.5.8). After deletion of the code on level $h-2$, $A_{\text{gap}}$ is forced to move $N/4$ codes. This is much worse than the worst case for the compact representation algorithm. Nevertheless, it would be interesting to investigate the best possible upper bound that can be proved for the competitive ratio of $A_{\text{gap}}$.

### 6.5.4 Resource augmented online-algorithm

In this section we give the sketch of a resource augmented online-strategy *2-gap*, see also Definition 2.1. In the case of the OVSF online code assignment problem the resource is the total available bandwidth. The strategy *2-gap* uses a tree $T'$ of bandwidth $2b$ to accommodate codes whose total bandwidth is $b$. By the nature of the code assignment we cannot add a smaller amount of additional resource. *2-gap* uses only an amortized constant number of reassignments per insertion or deletion.

Algorithm *2-gap* is similar to the compact representation algorithm of Section 6.5.1 (insisting on the ordering of codes according to their level, Invariant (6.5.1)), only that it allows for up to 2 gaps at each level $\ell$ (instead of only one for aligning), to the right of the assigned codes on $\ell$. The algorithm for inserting a code at level $\ell$ is to place it at the leftmost gap of $\ell$. If no such gap exists, we reassign the leftmost code of the next higher level $\ell+1$, creating 2 gaps (one of them is filled immediately by the new code) at $\ell$. We repeat this procedure toward the root. We reject an insertion if the nominal bandwidth $b$ is exceeded. For deleting a code $c$ on level $\ell$ we move the rightmost code on level $\ell$ into the position $c$, keeping all codes at level $\ell$ to the left of the gaps of $\ell$. If this results in 3 consecutive gaps, we reassign the rightmost code of level $\ell + 1$, in effect replacing two gaps of $\ell$ by one of $\ell + 1$. Again we proceed toward the root.

A detailed description of this algorithm can be found in the thesis of Gábor Szabó [120] and in [53]. The following theorem gives a performance guarantee for the algorithm:

**Theorem 6.27 ([120],[53])** *Let $\sigma$ be a sequence of $m$ code insertions and deletions for a code-tree of height $h$, such that at no time the bandwidth is exceeded. Then the above online-strategy uses a code-tree of height $h + 1$ and performs at most $2m + 1$ code assignments and reassignments.*

**Corollary 6.28** *The above strategy is 4-competitive for resource augmentation by a factor of 2.*

**Proof.** Any sequence of $m$ operations contains at least $m/2$ insert operations. Hence the optimal offline solution needs at least $m/2$ assignments, and the above resource augmented online-algorithm uses at most $2m+1$ (re-)assignments, leading to a competitive ratio of $4$. $\square$

## 6.6 Open Problems

In this chapter we derived a multitude of results. Some open problems remain unanswered.

- Is there a constant approximation algorithm for the one-step offline CA problem?
- Can the gap between the lower bound of 1.5 and the upper bound of $O(h)$ for the competitive ratio of the online CA be closed?
- Is there an instance where the optimal general offline algorithm has to reassign more than an amortized constant number of codes per insertion or deletion?

# Chapter 7

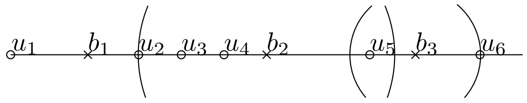
# Joint Base Station Scheduling

Sometimes our circuits get shorted
By external interference.
Signals get crossed
And the balance distorted
By internal incoherence.
(Rush - Vital Signs)

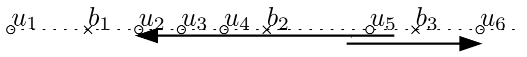
## 7.1 Introduction

In this chapter we consider different combinatorial aspects of a problem that arises in the context of load balancing in time division networks.

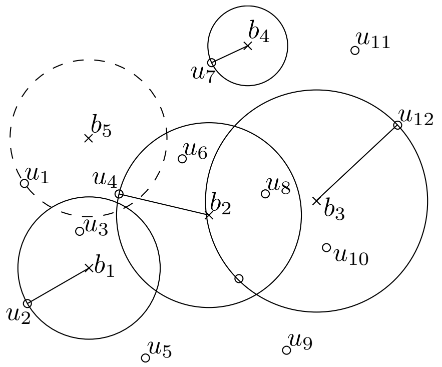
The general setting is that users with mobile devices are served by a set of base stations. In each time slot (round) of the time division multiplexing each base station serves at most one user. Traditionally, each user is assigned to a single base station that serves her until she leaves the cell of the base station or until her demand is satisfied. The amount of data that a user receives depends on the strength of the signal that she receives from her assigned base station and on the interference, i.e., all signal power that she receives from other base stations. In [43], Das et al. propose a novel approach: Clusters of base stations jointly decide which users they serve in which round in order to increase network performance. Intuitively, this approach increases throughput, when in each round neighboring base stations try to serve pairs of users such that the mutual interference is low. We turn this approach into a discrete scheduling problem in one and two dimensions (see Figure 7.1.1), the Joint Base Station Scheduling problem (JBS).



(a) A possible situation in some time slot (round). Base station $b_2$ serves user $u_2$, $b_3$ serves user $u_6$. Users $u_3, u_4$ and $u_5$ are blocked and cannot be served. Base station $b_1$ cannot serve $u_1$ because this would create interference at $u_2$.



(b) Arrow representation of (a).



(c) A possible situation in some time slot in the 2D case. Users $u_2, u_4, u_7$ and $u_{12}$ are served. Base station $b_5$ cannot serve user $u_1$, because this would create interference at $u_4$ as indicated by the dashed circle.

**Figure 7.1.1:** *The JBS-problem in one and two dimensions.*

In one dimension (see Figure 7.1.1(a)) we are given a set of $n$ users as points $\{u_1, \ldots, u_n\}$ on a line and we are given positions $\{b_1, \ldots, b_m\}$ of $m$ base stations. Note that such a setting could correspond to a scenario where the base stations and users are located along a straight road. In our model, when a base station $b_j$ serves a user $u_i$, this creates interference for other users in an interval of length $2|b_j - u_i|$ around the midpoint $b_j$. In any round each base station can serve at most one user such that at the position of this user there is no interference from any other base station. The goal is to serve all users in as few rounds as possible. In two dimensions users and base stations are represented as points in the plane. When base station $b_j$ serves user $u_i$ this creates interference in a disk with radius $\|b_j - u_i\|_2$ and center $b_j$ (see Figure 7.1.1(c)).

The one-dimensional problem is closely related to interval scheduling problems, except that the particular way how interference operates leads to directed intervals (arrows). For these, we allow their tails to intersect (intersecting tails correspond to interference that does not affect the users at the heads of the arrows). We present results on this special interval scheduling problem. Similarly, the problem is related to interval graphs, except that we have conflict graphs of arrows together with the conflict rules defined by the interference (*arrow graphs*).

### 7.1.1 Related Work

Das et al. [43] propose an involved model for load balancing that takes into account different fading effects and calculates the resulting signal to noise ratios at the users for different schedules. In each round only a subset of all base stations is used in order to keep the interference low. The decision on which base stations to use is taken by a central authority. The search for this subset is formulated as a (nontrivial) optimization problem that is solved by complete enumeration and that assumes complete knowledge of the channel conditions. The authors perform simulations on a hexagonal grid, propose other algorithms, and reach the conclusion that the approach has the potential to increase throughput.

There is a rich literature on interval scheduling and selection problems (see [54, 118] and the references given therein for an overview). Our problem is more similar to a setting with several machines where one wants to minimize the number of machines required to schedule all intervals. A version of this problem where intervals have to be scheduled within given time windows is studied in [30]. Inapproximability results for the variant with a discrete set of starting times for each interval are presented in [28].

### 7.1.2 Model and Notation

In this section we define the problems of interest. Our model of computation is the real RAM machine. The operands involved (positions on the line or in the plane) could be restricted also to rational numbers, but we use real operands to preserve the geometric properties of interval and disk intersections. In the one-dimensional case we are given a set $B = \{b_1, \ldots, b_m\} \subset \mathbb{R}$ of base station positions and a set $U = \{u_1, \ldots, u_n\} \subset \mathbb{R}$ of user positions on the line in left-to-right order. Conceptually, it is more convenient to think of the interference region that is caused by some base station $b_j$ serving a user $u_i$ as an *interference arrow* of length $2|b_j - u_i|$ with midpoint $b_j$ pointing to the user, as shown in Figure 7.1.1(b). The interference arrow for the pair $(u_i, b_j)$ has its head at $u_i$ and its midpoint at $b_j$. We denote the set of all arrows resulting from pairs $P \subseteq U \times B$ by $\mathcal{A}(P)$. If it is clear from the context, we call the interference arrows just *arrows*. If more than one user is scheduled in the same round then each of them must not get any interference from any other base station. Thus, two arrows are *compatible* if no head is contained in the other arrow; otherwise, we say that they are in *conflict*. Formally, the head $u_i$ of the arrow for $(u_i, b_k)$ is contained in the arrow for $(u_j, b_l)$ if $u_i$ is contained in the closed interval $[b_l - |u_j - b_l|, b_l + |u_j - b_l|]$. If we want to emphasize which user is affected by the interference from another transmission, we use the term *blocking*, i.e., arrow $a_i$ blocks arrow $a_j$ if $a_j$'s head is contained in $a_i$.

As part of the input we are given only the base station and user positions. The arrows that show which base station serves which user are part of the solution. For each user we have to decide from which base station she is served. This corresponds to a selection of an arrow for her. Furthermore, we have to decide in which round each selected arrow is scheduled under the side constraint that all arrows in one round must be compatible. For this purpose it is enough to label the arrows with colors that represent the rounds.

For the two-dimensional JBS problem we have positions in $\mathbb{R}^2$ and *interference disks* $d(b_i, u_j)$ with center $b_i$ and radius $\|b_i - u_j\|_2$ instead of arrows. We denote the set of interference disks for the user base-station pairs from a set $P$ by $\mathcal{D}(P)$. Two interference disks are in conflict if the user who is served by one of the disks is contained in the other disk; otherwise, they are compatible. The problems can now be stated as follows:

**1D-JBS**

**Input:** User positions $U = \{u_1, \ldots, u_n\} \subset \mathbb{R}$ and base station positions $B = \{b_1, \ldots, b_m\} \subset \mathbb{R}$.

**Output:** A set $P$ of $n$ user base-station pairs such that each user is in exactly one pair, and a coloring $C : \mathcal{A}(P) \rightarrow \mathbb{N}$ of the set $\mathcal{A}(P)$ of corresponding arrows such that any two arrows $a_i, a_j \in \mathcal{A}(P),\ a_i \neq a_j$, with $C(a_i) = C(a_j)$ are compatible.

**Objective:** Minimize the number of colors used.

**2D-JBS**

**Input:** User positions $U = \{u_1, \dots, u_n\} \subset \mathbb{R}^2$ and base station positions $B = \{b_1, \dots, b_m\} \subset \mathbb{R}^2$.

**Output:** A set $P$ of $n$ user base-station pairs such that each user is in exactly one pair, and a coloring $C : \mathcal{D}(\mathcal{P}) \rightarrow \mathbb{N}$ of the set $\mathcal{D}(\mathcal{P})$ of corresponding disks such that any two disks $d_i, d_j \in \mathcal{D}(\mathcal{P}),\ d_i \neq d_j$, with $C(d_i) = C(d_j)$ are compatible.

**Objective:** Minimize the number of colors used.

For simplicity, we will write $c_i$ instead of $C(a_i)$ in the rest of the chapter. From the problem definitions above it is clear that both the 1D- and the 2D-JBS problems consist of a *selection problem* and a *coloring problem*. In the selection problem we want to select one base station for each user in such a way that the arrows (disks) corresponding to the resulting set $P$ of user base-station pairs can be colored with as few colors as possible. We call a selection $P$ *feasible* if it contains exactly one user base-station pair for each user. Determining the cost of a selection is then the coloring problem. This can also be viewed as a problem in its own right, where we no longer make any assumption on how the set of arrows (for the 1D problem) is produced. The conflict graph $G(A)$ of a set $A$ of arrows is the graph in which every node corresponds to an arrow and there is an edge between two nodes if the corresponding arrows are in conflict. We call such conflict graphs of arrows *arrow graphs*. The *arrow graph coloring problem* asks for a proper coloring of such a graph. It is similar in spirit to the coloring of interval graphs. As we will see in Section 7.2.1, the arrow graph coloring problem can be solved in time $O(n \log n)$. We finish this section with a simple lemma that leads to a definition:

**Lemma 7.1** *For each 1D-JBS instance there is an optimal solution in which each user is served either by the closest base station to her left or by the closest base station to her right.*

**Proof.** This follows by a simple exchange argument: Take any optimal solution that does not have this form. Then exchange the arrow where a user is not served by the closest base station in some round against the arrow from the closest base station on the same side (which must be idle in that round). Shortening an arrow without moving its head can only resolve conflicts. Thus, there is also an optimal solution with the claimed property. □

The two possible arrows by which a user can be served according to this lemma are called *user arrows*. It follows that for a feasible selection one has to choose one user arrow from each pair of user arrows.

### 7.1.3 Summary of Results

The work presented in this chapter was done in collaboration with Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Gábor Szabó and Peter Widmayer. Extended abstracts of these results are published in [52] and [51], and have and will be presented with different focus in the theses of Gábor Szabó [120] and Matúš Mihaľ ák.

We prove that arrow graphs are perfect and can be colored optimally in $O(n \log n)$ time. For the one-dimensional JBS problem with evenly spaced base stations we give a polynomial-time dynamic programming algorithm. For another special case of the one-dimensional JBS problem, where $3k$ users must be served by $3$ base stations in $k$ rounds, we give a polynomial-time optimal algorithm. As a last variant we consider the decision problem of whether an instance can be served in $k$ rounds. We derive a 2-approximation algorithm for JBS based on an LP rounding. In the two-dimensional case deciding whether all users can be served in one round is doable in polynomial time. The general 2D-JBS problem is shown to be NP-complete. Finally, we analyze an approximation algorithm for a constrained version of the 2D-JBS problem, and present lower bounds on the quality of some natural greedy algorithms for the general two-dimensional JBS problem.

My main contribution concerns the one-dimensional problem, in particular, the algorithm for evenly spaced base stations and the analysis of the graph class of *arrow graphs*. For this reason, the emphasis in this chapter is on 1D-JBS.

## 7.2 1D-JBS

As mentioned earlier, solving the 1D-JBS problem requires selecting an arrow for each user and coloring the resulting arrow graph with as few colors as possible. To understand when a selection of arrows leads to an arrow graph with small chromatic number, we first study the properties of arrow graphs in relation to existing graph classes. Next we analyze special cases of 1D-JBS that are solvable in polynomial time. At the end of this section we present a dynamic program that solves the decision version of the 1D-JBS problem in time $n^{O(k)}$, where $k$ is the number of rounds, and we show a $2$-approximation algorithm. The big open problem remains the complexity of the general 1D-JBS problem: Is it NP-complete or is it polynomially solvable?

### 7.2.1 Relation to Other Graph Classes

In order to gain a better understanding of arrow graphs, we first discuss their relationship to other known graph classes.[1] We refer to [20, 119] for definitions and further information about the graph classes mentioned in the following.

First, it is easy to see that arrow graphs are a superclass of interval graphs: Any interval graph can be represented as an arrow graph with all arrows pointing in the same direction.

An arrow graph can be represented as the intersection graph of triangles on two horizontal lines $y = 0$ and $y = 1$: Simply represent an arrow with left endpoint $\ell$ and right endpoint $r$ that points to the right (left) as a triangle with corners $(\ell, 0)$, $(r, 0)$, and $(r, 1)$ (with corners $(r, 1)$, $(\ell, 1)$, and $(\ell, 0)$ respectively). With this representation two triangles intersect if and only if the corresponding arrows are in conflict, see Figure 7.2.1 for an example. Intersection graphs of triangles with endpoints on two parallel lines are known in the literature as PI$^*$ graphs. They are a subclass of trapezoid graphs, which are the intersection graphs of trapezoids that have two sides on two fixed parallel lines. Trapezoid graphs are in turn a subclass of co-comparability graphs, a well-known class of perfect graphs. Therefore, the containment in these known classes of perfect graphs implies the perfectness of arrow graphs. Consequently, the size of a maximum clique in an arrow graph equals its chromatic number.

As arrow graphs are a subclass of trapezoid graphs, we can apply known

[1]The connections between arrow graphs and known graph classes such as PI$^*$ graphs, trapezoid graphs, co-comparability graphs, AT-free graphs, and weakly chordal graphs were observed by Ekki Köhler, Jeremy Spinrad, Ross McConnell, and R. Sritharan at the seminar “Robust and Approximative Algorithms on Particular Graph Classes”, held in Dagstuhl Castle during May 24–28, 2004.

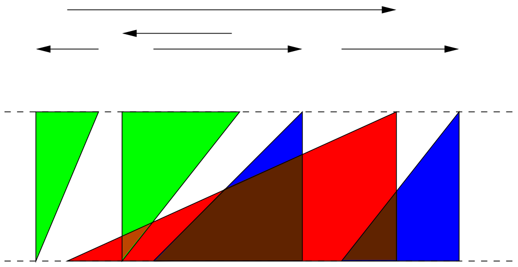


**Figure 7.2.1:** *An arrow graph (top) and its representation as a PI* graph (bottom).*

efficient algorithms for trapezoid graphs to arrow graphs. Felsner et al. [58] give algorithms with running-time $O(n \log n)$ for chromatic number, weighted independent set, weighted clique, and clique cover in trapezoid graphs with $n$ nodes, provided that the trapezoid representation is given. The coloring algorithm provided in [58] is similar to the following *greedy coloring* algorithm.

We assume for simplicity that the arrows $A = \{a_1, \ldots, a_n\}$ are given in left-to-right order of their left endpoints. This sorting can also be seen as the first step of the greedy coloring algorithm. The algorithm scans the arrows from left to right in this sorted order. In step $i$ it checks whether there are colors that have already been used and that can be assigned to $a_i$ without creating a conflict. If there are such candidate colors, it considers, for each such color $c$, the rightmost right endpoint $r_c$ among the arrows that have been assigned color $c$ so far. To $a_i$ is assigned the color $c$ for which $r_c$ is rightmost (breaking ties arbitrarily). If there is no candidate color, the algorithm assigns a new color to $a_i$.

We show that this greedy algorithm produces an optimal coloring by showing that any optimal solution can be transformed into the solution produced by the algorithm.

**Lemma 7.2** *Let $C$ be an optimal coloring for a set of arrows $A = \{a_1, \ldots, a_n\}$. The coloring $C$ can be transformed into the coloring produced by the greedy algorithm without introducing new colors.*

**Proof.** We show the lemma by induction on the index of the arrows. The induction hypothesis is: *There exists an optimal coloring that agrees with the greedy coloring up to arrow $k-1$.* The induction start is trivial. In the $k$th step let $C = (c_1, \ldots, c_n)$ be such an optimal coloring and let $H = (h_1, \ldots, h_n)$ be the greedy coloring, i.e., we have $h_1 = c_1, h_2 = c_2, \ldots, h_{k-1} = c_{k-1}$. We consider the coloring $C' = (c'_1, \ldots, c'_n)$ that is obtained from $C$ by exchanging the colors $c_k$ and $h_k$ for the arrows $a_k, \ldots, a_n$. More precisely, we define

$$c'_i = \begin{cases} c_i, & \text{if } i < k \text{ or } c_i \notin \{c_k, h_k\} \\ h_k, & \text{if } i \geq k \text{ and } c_i = c_k \\ c_k, & \text{if } i \geq k \text{ and } c_i = h_k. \end{cases}$$

By definition we have $c'_k = h_k$, and it remains to show that $C'$ is a proper coloring and, therefore, the induction hypothesis is also true for $k$. If $c_k = h_k$ we have $C' = C$, which is a proper coloring. Otherwise, we have to show that all pairs of arrows $a_i, a_j$ that are in conflict receive different colors in $C'$, i.e., $c'_i \neq c'_j$. If $i, j < k$ or $k \leq i, j$ this is obvious by the fact that $C$ is a coloring. Hence, we assume $i < k < j$; the case $j = k$ is implied by $H$ being a proper coloring.

If $h_k$ is a new color, i.e., different from all of $c_1, \ldots, c_{k-1}$, then, because of the greedy algorithm, also $c_k$ is a new color. Hence, it is impossible that we have $c'_i = c'_j$.

Now assume for a contradiction that we indeed have $c = c'_i = c'_j$ and the arrows $a_i$ and $a_j$ are in conflict. By the ordering of the arrows we know that $a_i$ and $a_k$ overlap. Observe that $c \in \{c_k, h_k\}$ because $C$ is a coloring. This leaves us with two cases:

**Case 1** $c = c_k$**:** Since $C$ is a coloring, the arrows $a_i$ and $a_k$ are compatible, i.e. $a_i$ is directed left and $a_k$ is directed right. Such a configuration is depicted in Figure 7.2.2. By the definition of the greedy algorithm, we know that $h_k$ is a color of a compatible arrow. Since $h_k \neq c_k = c_i$, there must exist an arrow $a_l$, $l < k$, that ends not before $a_i$ and has color $h_k$, i.e. $c_l = h_k$ (and $a_l$ is compatible with $a_k$). Since $a_j$ is in conflict with $a_i$ (the head of $a_j$ is within $a_i$), there is also a conflict between $a_j$ and $a_l$. We have $c'_j = c_k$, implying $c_j = h_k$, hence we get the contradiction $c_j = h_k = c_l$ in the optimal coloring $C$.

**Case 2** $c = h_k$**:** Because $H$ is a coloring, $a_i$ and $a_k$ have to be compatible. Since $a_i$ ends before $a_k$ and is in conflict with $a_j$, also $a_j$ is in conflict with $a_k$. Because $c'_j = h_k$ we know by definition of $C'$ that $c_j = c_k$, hence there is a conflict in $C$, a contradiction. $\square$

The running time of the algorithm depends on the time the algorithm spends in every step on identifying an allowed color that was previously assigned to an arrow with the rightmost right endpoint. By maintaining two balanced search trees (one tree for each direction of arrows) storing the most recently colored arrows of the used colors (one arrow per color) in the order of their right endpoints, we can implement this operation in logarithmic time. Together with Lemma 7.2 we get the following theorem.

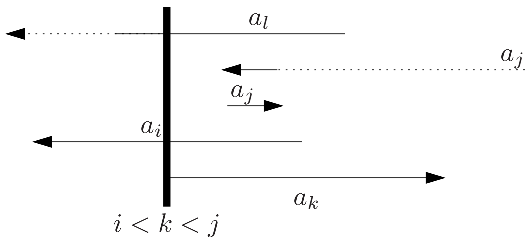


**Figure 7.2.2:** *Possible configuration for the two cases. Dotted lines mean that the arrows could be extended*

**Theorem 7.3** *The greedy algorithm optimally colors a given set of arrows* $\{a_1, \ldots, a_n\}$ *in* $O(n \log n)$ *time.*

We sum up the discussed properties of arrow graphs in the following theorem.

**Theorem 7.4** *Arrow graphs are perfect. In arrow graphs chromatic number, weighted independent set, clique cover, and weighted clique can be solved in time* $O(n \log n)$.

One can also show that arrow graphs are AT-free (i.e., do not contain an asteroidal triple) and weakly chordal.

### 7.2.2 1D-JBS with Evenly Spaced Base Stations

As already mentioned it is still open whether 1D-JBS is NP-complete or not. In order to explore the complexity boundary for the problem we searched for simpler variants, for which there is a polynomial algorithm and for harder variants that are NP-complete. An obvious harder variant is 2D-JBS, which we prove to be NP-complete in Section 7.3.1. A simpler variant is the 1D-JBS problem, in which all base stations are evenly spaced, such that all neighboring base stations have the same distance $d$ from each other. Additionally, we assume that the leftmost user is not further apart than distance $d$ from the leftmost base station, similarly for the rightmost user. We call such users that are further than $d$ apart from any base station *far out users*.

The $m$ base stations partition the line into a set $\{v_0, \ldots, v_m\}$ of *intervals*. We assume that the base stations are given in left to right order. For this setting we can conclude from Lemma 7.1 that no interference arrow intersects more than two intervals, i.e., the influence of a base station is limited to its direct left and right neighboring base station. A solution (selection of arrows) is considered *non-crossing* if there are no two users $u$ and $w$ in the same interval such that $u$ is to the left of $w$, $u$ is served from the right, and $w$ from the left, in two different rounds.

**Lemma 7.5** *For instances of 1D-JBS with evenly spaced base stations, there is always an optimal solution that is non-crossing.*

**Proof.** Consider an optimal solution $s$ that is not non-crossing. We show that such a solution can be transformed into another optimal solution $s'$ that is non-crossing. Let $u$ and $w$ be two users such that $u$ and $w$ are in the same interval, $u$ is to the left of $w$, and $u$ is served by the right base station $b_r$ in round $t_1$ by arrow $a_r$ and $w$ is served by the left base station $b_l$ in round $t_2$ by arrow $a_l$; obviously, $t_1 \neq t_2$. We can modify $s$ such that in round $t_1$ base station $b_r$ serves $w$ and in $t_2$ base station $b_l$ serves $u$. This new solution is still feasible because first of all both the left and the right involved arrows $a_l$ and $a_r$ have become shorter. This implies that both $a_l$ and $a_r$ can only block fewer users. On the other hand, the head of $a_l$ has moved left and the head of $a_r$ has moved right. It is impossible that they are blocked now because of this movement: In $t_1$ this could only happen if there were some other arrows containing $w$, the new head of $a_r$. This arrow cannot come from the left, because then it would have blocked also the old arrow. It cannot come from $b_r$ because $b_r$ is busy. It cannot come from a base station to the right of $b_r$, because such arrows do not reach[2] any point to the left of $b_r$. For $t_2$ the reasoning is symmetric. □

The selection of arrows in any non-crossing solution can be completely characterized by a sequence of $m-1$ *division points*, such that the $i^{th}$ division point specifies the index of the last user that is served from the left in the $i^{th}$ interval. (The case where all users in the $i^{th}$ interval are served from the right is handled by choosing the $i^{th}$ division point as the index of the rightmost user to the left of the interval, or as 0 if no such user exists.) A brute-force approach could now enumerate over all possible $O(n^{m-1})$ division point sequences (*dps*) and color the selection of arrows corresponding to each dps with the greedy algorithm from Section 7.2.1.

## Dynamic Programming

We can solve the 1D-JBS problem with evenly spaced base stations more efficiently by a dynamic programming algorithm that runs in polynomial time. The

[2]Here we use the assumption that the rightmost user is no farther to the right of the rightmost base station than $d$, and that the base stations are evenly spaced.

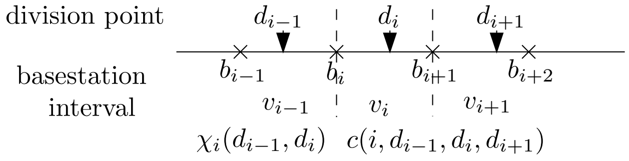


**Figure 7.2.3:** *Dynamic programming approach*

idea of the algorithm is to consider the base stations and thus the intervals in left-to-right order. We consider the cost $\chi_i(d_{i-1}, d_i)$ of an optimal solution up to the $i$th base station conditioned on the position of the division points $d_{i-1}$ and $d_i$ in the intervals $v_{i-1}$ and $v_i$, respectively, see Figure 7.2.3.

**Definition 7.6** *We denote by $\chi_i(\alpha, \beta)$ the minimum number of colors needed to serve users $u_1$ to $u_\beta$ using the base stations $b_1$ to $b_i$ under the condition that base station $b_i$ serves exactly users $u_{\alpha+1}$ to $u_\beta$ and ignoring the users $u_{\beta+1}, \ldots, u_n$.*

Let $\Lambda(v_i)$ denote the set of potential division points for interval $v_i$, i.e., the set of the indices of users in $v_i$ and of the rightmost user to the left of $v_i$ (or 0 if no such user exists). The values $\chi_1(d_0, d_1)$ for $d_0 = 0$ (all users to the left of $b_1$ must be served by $b_1$ in any solution) and $d_1 \in \Lambda(v_1)$ can be computed directly by using the greedy coloring algorithm. For $i \geq 1$, we compute the values $\chi_{i+1}(d_i, d_{i+1})$ for $d_i \in \Lambda(v_i)$, $d_{i+1} \in \Lambda(v_{i+1})$ from the table for $\chi_i(\cdot, \cdot)$. If we additionally fix a division point $d_{i-1}$ for interval $v_{i-1}$, we know exactly which selected arrows intersect interval $v_i$ regardless of the choice of other division points. Observe that this only holds for evenly spaced base stations and no far out users. For this selection, we can determine with the greedy coloring algorithm how many colors are needed to color the arrows intersecting $v_i$. Let us call this number $c(i, d_{i-1}, d_i, d_{i+1})$ for interval $v_i$ and division points $d_{i-1}, d_i$ and $d_{i+1}$. We also know how many colors we need to color the arrows intersecting intervals $v_0$ to $v_{i-1}$. For a fixed choice of division points $d_{i-1}$, $d_i$ and $d_{i+1}$ we can combine the two colorings corresponding to $\chi_i(d_{i-1}, d_i)$ and $c(i, d_{i-1}, d_i, d_{i+1})$: Both of these colorings color all arrows of base station $b_i$, and these arrows must all have different colors in both colorings. No other arrows are colored by both colorings, so $\chi_i(d_{i-1}, d_i)$ and $c(i, d_{i-1}, d_i, d_{i+1})$ agree up to redefinition of colors. We can choose the best division point $d_{i-1}$ and get

$$\chi_{i+1}(d_i, d_{i+1}) = \min_{d_{i-1} \in \Lambda(v_{i-1})} \max \{\chi_i(d_{i-1}, d_i), c(i, d_{i-1}, d_i, d_{i+1})\}$$

The running time is dominated by the calculation of the $c(\cdot)$ values. There are $O(m \cdot n^3)$ such values, and each of them can be computed in time $O(n \log n)$

using the greedy coloring algorithm. The optimal solution can be found in the usual way by tracing back where the minimum was achieved from $\chi_m(x, n)$. Here the $x$ is chosen among the users of the interval before the last base station such that $\chi_m(x, n)$ is minimum. For the traceback it is necessary to store in the computation of the $\chi$ values where the minimum was achieved. The traceback yields a sequence of division points that defines the selection of arrows that gives the optimal schedule. Altogether, we have shown the following theorem:

**Theorem 7.7** *The base station scheduling problem for evenly spaced base stations can be solved in time $O(m \cdot n^4 \log n)$ by dynamic programming.*

Note that the running time can also be bounded by $O(m \cdot u_{\max}^4 \log u_{\max})$, where $u_{\max}$ is the maximum number of users in one interval.

### 7.2.3 $3k$ Users, $3$ Base Stations in $k$ Rounds

In the last section we made the restriction that the input must not contain far out users. One could ask whether the complexity of the problem is changed when far out users are present. In order to explore this direction a bit we define a (very special) variant here, in which far out users are present, but still the problem can be solved in polynomial time: We are given 3 base stations $b_1$, $b_2$ and $b_3$, and $3k$ users with $k$ far out users among them. Far out users are the users to the left of $b_1$ or to the right of $b_3$ whose interference arrows contain $b_2$. We want to find out whether the users can be served in $k$ rounds or not.

This special setting forces every base station to serve a user in every round if there is a $k$-schedule. A far out user has to be served by its unique neighboring base station. Since the arrows of far out users contain $b_2$, all users between $b_1$ and $b_2$ are blocked when the far out users of $b_1$ are served. Hence they have to be served when the far out users of $b_3$ are served. Based on this observation every round contains one of the following types of arrow triplets:

**Type 1:** $b_3$ serves a far out user, $b_2$ serves a user between $b_1$ and $b_2$, and $b_1$ serves a user that is not a far out user.

**Type 2:** $b_1$ serves a far out user, $b_2$ serves a user between $b_2$ and $b_3$, and $b_3$ serves a user that is not a far out user.

For every user, it is uniquely determined whether it will be served in a round of Type 1 or Type 2.

We can schedule the users in the following way. Let $k_1$ and $k_3$ be the number of far out users of $b_1$ and $b_3$ respectively with $k = k_1 + k_3$. First, we serve

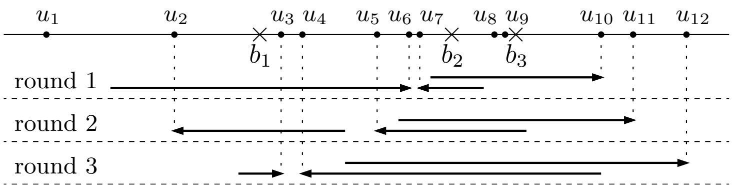


**Figure 7.2.4:** *Far out users $u_{10}$, $u_{11}$ and $u_{12}$ are served by $b_3$ in rounds $1$, $2$ and $3$, respectively. The arrows represent the Type 1 rounds. Users $u_1$, $u_8$ and $u_9$ will be scheduled in a round of Type 2 (not shown).*

the far out users of $b_3$ in rounds $1, \ldots, k_3$ in the order of increasing distance from $b_3$. Next, we match the resulting arrows in a best fit manner with arrows produced by $b_2$ serving users between $b_1$ and $b_2$ (see Figure 7.2.4). For every round $i = 1, 2, \ldots, k_3$, we find the user closest to $b_2$ that can be served together with the corresponding far out user served by $b_3$, and schedule the corresponding transmission in that round. Using this selection strategy the size of the arrows of $b_2$ grows with the number of the round in which they are scheduled. Now we have to serve the remaining $k_3$ users (that are not far out users of $b_1$) with $b_1$. We use a best fit approach again, i.e., for every round $i = 1, 2, \ldots, k_3$, we schedule the user with maximum distance from $b_1$ (longest arrow) among the remaining users. The schedule for the remaining users that form the rounds of Type 2 can be done similarly, starting with the far out users of $b_1$.

**Theorem 7.8** *For the 1D-JBS problem with $3$ base stations and $3k$ users with $k$ far out users deciding whether a $k$-schedule exists can be done in $O(n \log n)$ time.*

**Proof.** The proof can be found in [52] and will appear in the thesis of Matúš Mihaľák. The proof shows that the greedy scheduling strategy finds such a $k$-schedule in time $O(n \log n)$ if one exists. □

## 7.2.4 Exact Algorithm for the $k$-Decision Problem

In this section we present an exact algorithm for the decision variant $k$-1D-JBS of the 1D-JBS problem: For given $k$ and an instance of 1D-JBS, decide whether all users can be served in at most $k$ rounds. We present an algorithm for this problem that runs in $O(m \cdot n^{2k+1} \log n)$ time.

We use the result from Section 7.2.1 that arrow graphs are perfect. Thus the size of the maximum clique of an arrow graph equals its chromatic number.

The idea of the algorithm, which we call $A_{k-\mathrm{JBS}}$, is to divide the problem into subproblems, one for each base station, and then combine the partial solutions to a global one.

For base station $b_i$, the corresponding subproblem $S_i$ considers only arrows that intersect $b_i$ and arrows for which the alternative user arrow[3] intersects $b_i$. Call this set of arrows $A_i$. We call $S_{i-1}$ and $S_{i+1}$ *neighbors* of $S_i$. A solution to $S_i$ consists of a feasible selection of arrows from $A_i$ of cost no more than $k$, i.e. the selection can be colored with at most $k$ colors. To find all such solutions we enumerate all possible selections that can lead to a solution in $k$ rounds. For $S_i$ we store all such solutions $\{s_i^1, \ldots, s_i^I\}$ in a table $T_i$. We only need to consider selections in which at most $2k$ arrows intersect the base station $b_i$. All other selections need more than $k$ rounds, because they must contain more than $k$ arrows pointing in the same direction at $b_i$. Therefore, the number of entries of $T_i$ is bounded by $\sum_{j=0}^{2k} \binom{n}{j} = O(n^{2k})$. We need $O(n \log n)$ time to evaluate a single selection with the greedy coloring algorithm. Selections that cannot be colored with at most $k$ colors are marked as irrelevant and ignored in the rest of the algorithm. We build up the global solution by choosing a set of feasible selections $s_1, \ldots, s_m$ in which all neighbors are compatible, i.e. they agree on the selection of common arrows. It is easy to see that in such a global solution all subsolutions are pairwise compatible.

We can find such a set of compatible neighbors by going through the tables in left-to-right order and marking every solution in each table as *valid* if there is a compatible, valid solution in the table of its left neighbor, or as *invalid* otherwise. A solution $s_i$ marked as valid in table $T_i$ thus indicates that there are solutions $s_1, \ldots, s_{i-1}$ in $T_1, \ldots, T_{i-1}$ that are compatible with it and pairwise compatible. In the leftmost table $T_1$, every feasible solution is marked as valid. When the marking has been done for the tables of base stations $b_1, \ldots, b_{i-1}$, we can perform the marking in the table $T_i$ for $b_i$ in time $O(n^{2k+1})$ as follows. First, we go through all entries of the table $T_{i-1}$ and, for each such entry, in time $O(n)$ discard the part of the selection affecting pairs of user arrows that intersect only $b_{i-1}$ but not $b_i$, and enter the remaining selection into an intermediate table $T_{i-1,i}$. The table $T_{i-1,i}$ stores entries for all selections of arrows from pairs of user arrows intersecting both $b_{i-1}$ and $b_i$. An entry in $T_{i-1,i}$ is marked as valid if at least one valid entry from $T_{i-1}$ has given rise to the entry. Then, the entries of $T_i$ are considered one by one, and for each such entry $s_i$ the algorithm looks up in time $O(n)$ the unique entry in $T_{i-1,i}$ that is compatible with $s_i$ to see whether it is marked as valid or not, and marks the entry in $T_i$ accordingly. If in the end the table $T_m$ contains a solution marked as valid, a set of pairwise compatible

[3]For every user there are only two user arrows that we need to consider (Lemma 7.1). If we consider one of them, the other one is the *alternative user arrow*.

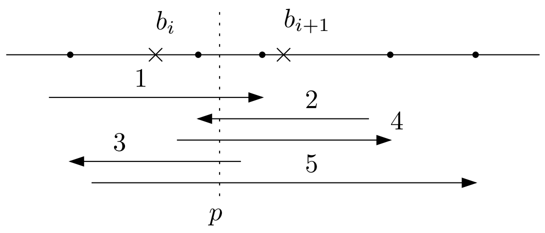


**Figure 7.2.5:** *Arrow types intersecting at a point $p$ between base stations $b_i$ and $b_{i+1}$.*

solutions from all tables exists and can be retraced easily.

The overall running time of the algorithm is $O(m \cdot n^{2k+1} \cdot \log n)$. There is a solution to $k$-1D-JBS if and only if the algorithm finds such a set of compatible neighbors.

**Lemma 7.9** *There exists a solution to $k$-1D-JBS if and only if $A_{k-\mathrm{JBS}}$ finds a set of pairwise compatible solutions.*

**Proof.** $(\Rightarrow)$ Every arrow intersects at least one base station. A global solution directly provides us with a set of compatible subsolutions $\Sigma_{\mathrm{opt}} = \{s_1^{\mathrm{opt}}, \ldots, s_m^{\mathrm{opt}}\}$. Since the global solution has cost at most $k$, so have the solutions of the subproblems. Hence, the created entries will appear in the tables of the algorithm and will be considered and marked as valid. Thus, there is at least one set of compatible solutions that is discovered by the algorithm.

$(\Leftarrow)$ We have to show that the global solution constructed from the partial ones has cost at most $k$. Suppose for a contradiction that there is a point $p$ where the clique size is bigger than $k$ and therefore bigger than the clique at $b_i$ (the left neighboring base station of $p$) and the clique at $b_{i+1}$ (the right neighboring base station of $p$). We divide the arrows intersecting point $p$ into 5 groups as in Figure 7.2.5. Arrows of type 1 (2) have their head between $b_i$ and $b_{i+1}$ and their tail to the left (right) of $b_i$ ($b_{i+1}$). Arrows of type 3 (4) have their tail between $b_i$ and $b_{i+1}$ and their head to the left (right) of $b_i$ ($b_{i+1}$). Finally, type $5$ arrows intersect both $b_i$ and $b_{i+1}$. For the clique at $p$ to be bigger than that at $b_i$ some arrows not considered at $b_i$ have to create conflicts. The only such arrows (considered at $b_{i+1}$ but not at $b_i$) are of type $4$. Observe that arrows of type $1$, $2$ and $5$ are considered both at the table for $b_i$ and at the table for $b_{i+1}$. If their presence increases the clique size at $p$, then no type $3$ arrow can be in the maximum clique at $p$ (observe that arrows of type $3$ and $4$ are compatible).

A type 3 arrows are the only arrows present at $p$ but not at $b_{i+1}$, the clique at $p$ cannot be bigger than the clique at $b_{i+1}$, a contradiction. □

To sum up, we have shown the following theorem.

**Theorem 7.10** *Problem $k$-1D-JBS can be solved in $O(m \cdot\ n^{2k+1}\ \log n)$ time.*

### 7.2.5 Approximation Algorithm

In this section we present an approximation algorithm for 1D-JBS that relies on the properties of arrow graphs from Theorem 7.4. Let $A$ denote the set of all user arrows of the given instance of 1D-JBS. From the perfectness of arrow graphs it follows that it is equivalent to ask for a feasible selection $A_{\text{sel}} \subseteq A$ minimizing the chromatic number of its arrow graph $G(A_{\text{sel}})$ (among all feasible selections) and to ask for a feasible selection $A_{\text{sel}}$ minimizing the maximum clique size of $G(A_{\text{sel}})$ (among all feasible selections). Exploiting this equivalence, we can express the 1D-JBS problem as an integer linear program as follows. We introduce two indicator variables $l_i$ and $r_i$ for every user $i$ that indicate whether she is served by the left or by the right base station, i.e. if the user's left or right user arrow is selected. Moreover, we ensure by the constraints that no cliques in $G(A_{\text{sel}})$ are large and that each user is served. The ILP formulation is as follows:

$$\min \quad k \tag{7.2.1}$$

$$\text{s.t.} \quad \sum_{l_i \in C} l_i + \sum_{r_i \in C} r_i \leq k, \quad \forall \text{ cliques } C \text{ in } G(A) \tag{7.2.2}$$

$$l_i + r_i = 1, \quad \forall i \in \{1, \ldots, |U|\} \tag{7.2.3}$$

$$l_i, r_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, |U|\} \tag{7.2.4}$$

$$k \in \mathbb{N} \tag{7.2.5}$$

The natural LP relaxation is obtained by allowing $l_i, r_i \in [0, 1]$ and $k \geq 0$. Given a solution to this relaxation, we can use a rounding technique to get an assignment of users to base stations that has cost at most twice the optimum, i.e., we obtain a 2-approximation algorithm. Let us denote by *opt* the optimum number of colors needed to serve all users. Then *opt* $\geq k$, because the optimum integer solution is a feasible fractional solution. Construct now a feasible solution from a solution to the relaxed problem by rounding $l_i := \lfloor l_i + 0.5 \rfloor$, $r_i := 1 - l_i$. Before the rounding the size of every (fractional) clique is at most $k$; afterwards

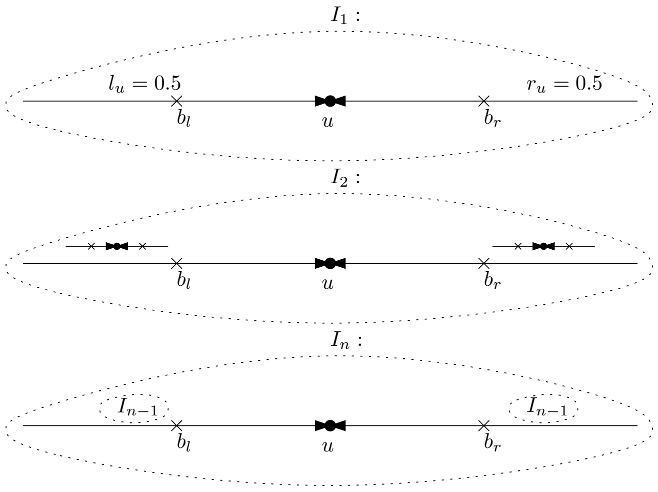


**Figure 7.2.6:** *Lower bound example for the 2-approximation ratio of the LP relaxation technique.*

the size can double in the worst case. Therefore, the cost of the rounded solution is at most $2k \leq 2opt$.

The factor of 2 is tight for our technique because the gap between a fractional and an integral solution can really get arbitrarily close to 2: In Figure 7.2.6 the cost of an optimal fractional solution is smaller than the cost of an optimal integral solution by a factor arbitrarily close to 2. In this example, the basic construction $I_1$ contains two base stations $b_l$ and $b_r$ and one user $u$ in-between. Both the solution of the ILP and the solution of the LP relaxation have cost 1. $I_2$ is constructed recursively by adding to $I_1$ two (scaled) copies of $I_1$ in the tail positions of the arrows. In this case the cost of the relaxed LP is 1.5 and the integral cost is 2. The construction $I_n$, after $n$ recursive steps, is shown at the bottom of Figure 7.2.6. This construction is achieved by using $I_1$ and putting two scaled $I_{n-1}$ settings in the tail of the arrows from $I_1$. The cost of the LP relaxation for $I_n$ is $\frac{n+1}{2}$, whereas the cost of the ILP is $n$.

One issue that needs to be discussed is how the relaxation can be solved in time polynomial in $n$ and $m$, as there can be an exponential number of constraints (7.2.2). (Figure 7.2.7 shows that this can really happen. The potentially exponential number of maximal cliques in arrow graphs distinguishes them from interval graphs, which have only a linear number of maximal cliques.) Fortunately, we can still solve such an LP in polynomial time with the ellipsoid method

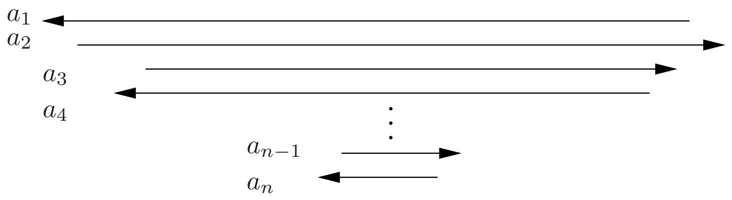


**Figure 7.2.7:** *Example of an arrow graph with an exponential number of maximum cliques. For every choice of arrows from a compatible pair $(a_{2i-1}, a_{2i})$ we get a clique of size $n/2$, which is maximum. The arrow graph can arise from a 1D-JBS instance with two base stations in the middle and $n/2$ users on either side.*

of Khachiyan [79] applied in a setting similar to [67]. This method only requires a separation oracle that provides us for any values of $l_i, r_i$ with a violated constraint, if one exists, see also Section 4.6. It is easy to check for a violation of constraints (7.2.3) and (7.2.4). For constraints (7.2.2), we need to check if for given values of $l_i, r_i$ the maximum weighted clique in $G(A)$ is smaller than $k$. By Theorem 7.4 this can be done in time $O(n \log n)$. Summarizing, we get the following theorem:

**Theorem 7.11** *There is a polynomial-time $2$-approximation algorithm for the 1D-JBS problem.*

### 7.2.6 Different Interference Models

Up to now we have analyzed the discrete interference model where the interference region has no effect beyond the targeted user. One step towards a more realistic model is to consider the interference region, produced by a base station sending a signal to a user, to span also beyond the targeted user. We call the 1D-JBS problem using this more realistic interference model the *modified 1D-JBS* problem. For the 1-dimensional case this can be modeled by using *interference segments* with the user somewhere between the endpoints of this segment (the small black circles on the segments in Figure 7.2.8) and the base station in the middle of the segment. The conflict graph of such interference segments is another special case of trapezoid graphs. For an example see Figure 7.2.8. The trapezoid representing the segment $[a, b]$ (serving user $u$) from Figure 7.2.8 is built using the parallel edges $[a', u']$ (the projection of the segment $[a, u]$ onto the upper supporting line of the trapezoid) and $[u'', b']$ (the projection of the segment $[u, b]$ onto the lower supporting line of the trapezoid).

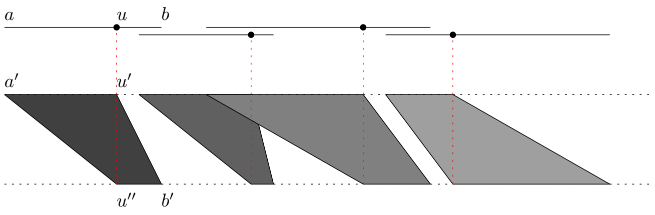


**Figure 7.2.8:** *Example for interference segments.*

We also get the trapezoid representation mentioned above if we consider a segment with a user between its endpoints as two arrows pointing to the user one from left and one from right. Then the triangle transformation for arrows (from Section 7.2.1) results in the trapezoid representation from Figure 7.2.8. Thus, for the *modified 1D-JBS* using Theorem 7.11 we have the following result:

**Corollary 7.12** *There is a polynomial-time* $2$*-approximation algorithm for the* modified 1D-JBS *problem.*

The proof is similar to the proof from Section 7.2.5, except that instead of arrow graphs we have another special case of trapezoid graphs.

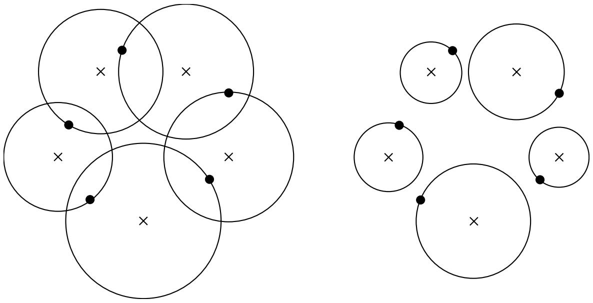


**Figure 7.3.1:** *A cycle of length 5 in the conflict graph of interference disks (left). It is not clear, however, whether an optimal solution to the selection problem will ever yield such a conflict graph; a different selection for this instance yields a conflict graph with five isolated nodes (right).*

## 7.3 2D-JBS

We now turn to the two-dimensional version 2D-JBS. We first show that the decision variant $k$-2D-JBS of 2D-JBS is NP-complete. Then we present a constant factor approximation for a constrained version of it and briefly discuss lower bounds for natural algorithms for the general 2D-JBS problem.

One could be led to believe that an extension of the approximation algorithm in Section 7.2.5 should lead to an approximation algorithm for 2D-JBS. However, the conflict graph of a set of interference disks is not necessarily perfect: It can have odd cycles as shown in Figure 7.3.1.

### 7.3.1 NP-Completeness of the $k$-2D-JBS Problem

In this section we briefly sketch our reduction from the general graph $k$-colorability problem [62] to 2D-JBS; the complete proof can be found in [120]. Our reduction follows the methodology presented in [66] for unit disk $k$-colorability.

Given any graph $G$, it is possible to construct in polynomial time a corresponding 2D-JBS instance that can be scheduled in $k$ rounds if and only if $G$ is $k$-colorable. We use an embedding of $G$ into the plane which allows us to replace the edges of $G$ with suitable base station chains with several users in a systematic way such that $k$-colorability is preserved. Our main result is the following:

**Theorem 7.13** *The $k$-2D-JBS problem in the plane is NP-complete for any fixed $k \geq 3$.*

In the $k$-2D-JBS instances used in our reduction, the selection of the base station serving each user is uniquely defined by the construction. Hence, our reduction proves that already the coloring step of the 2D-JBS problem is NP-complete.

**Corollary 7.14** *The coloring step of the $k$-2D-JBS problem is NP-complete for any fixed $k \geq 3$.*

### 7.3.2 Bounded Geometric Constraints

Here, we consider a constrained version of the 2D-JBS problem. In the real life application of mobile communication networks it is often the case that the maximum reach of a cell is limited by the maximum transmitting power. It is also common sense to consider that base stations cannot be placed arbitrarily but a certain minimum distance between them has to be maintained. These are the two geometric constraints that we use in this section. Namely, the base stations are at least a distance $\Delta$ from each other and have limited power to serve a user, i.e., every base station can serve only users that are at most $R_{\max}$ distance away from it. To make sure that under these constraints a feasible solution exists (i.e. all users can be served) we limit ourselves to instances where every user can be reached by at least one base station. We present a simple algorithm achieving an approximation ratio which only depends on the parameters $\Delta$ and $R_{\max}$.

Consider the following greedy approach $A_{2D-appx}$: In the current round the algorithm repeatedly picks an arbitrary user base-station pair $(u, b)$, where $u$ is an unserved user, such that the transmission from $b$ to $u$ can be added to this round without creating a conflict. If no such user base-station pair exists, the next round starts. The algorithm terminates when all users have been served.

The approximation ratio achieved by $A_{2D-appx}$ is given in the following analysis. Assume that the algorithm schedules the users in $k$ rounds. Let $u$ be a user served in round $k$, and let $b$ be the base station serving $u$. Since $u$ was not served in the previous rounds $1, 2, \ldots, k-1$, we know that in each of these rounds, at least one of the following is true:

- $b$ serves another user $u' \neq u$.
- $u$ is contained in an interference disk $d(b', u')$ for some user $u' \neq u$ that is served in that round.
- $b$ cannot transmit to $u$ because the disk $d(b, u)$ contains another user $u'$ that is served in that round.

In any of these cases, a user $u'$ is served, and the distance between $u$ and $u'$ is at most $2R_{\max}$ (since every interference disk has radius at most $R_{\max}$). Therefore, the disk with radius $2R_{\max}$ centered at $u$ contains at least $k$ users (including $u$). If $B'$ is the set of base stations that serve these $k$ users in the optimal solution, these base stations must be located in a disk with radius $3R_{\max}$ centered at $u$. Since any two base stations are separated by a distance of $\Delta$, we know that disks with radius $\Delta/2$ centered at base stations are interior-disjoint. Furthermore, the disks with radius $\Delta/2$ centered at the base stations in $B'$ are all contained in a disk with radius $3R_{\max} + \Delta/2$ centered at $u$. Therefore, the following inequality holds

$$|B'| \leq \frac{(3R_{\max} + \Delta/2)^2\pi}{(\Delta/2)^2\pi} = \frac{(6R_{\max} + \Delta)^2}{\Delta^2} .$$

Hence the optimal solution needs at least $k/|B'|$ rounds. This yields the following theorem.

**Theorem 7.15** *There exists an approximation algorithm with approximation ratio* $(\frac{6R_{\max}+\Delta}{\Delta})^2$ *for 2D-JBS in the setting where any two base stations are at least* $\Delta$ *away from each other and every base station can serve only users within distance at most* $R_{\max}$ *from it.*

### 7.3.3 General 2D-JBS

In the technical report[4] [52] we also discuss lower bounds on three natural greedy approaches for the general 2D-JBS problem: serve a maximum number of users in each round (*max-independent-set*), or repeatedly choose an interference disk of an unserved user with minimum radius (*smallest-disk-first*), or repeatedly choose an interference disk containing the fewest other unserved users (*fewest-users-in-disk*). In [52] we prove the following theorem.

**Theorem 7.16** *There are instances* $(U, B)$ *of 2D-JBS in general position (i.e., with no two users located on the same circle centered at a base station) for which the maximum-independent-set greedy algorithm, the smallest-disk-first greedy algorithm, and the fewest-users-in-disk greedy algorithm have approximation ratio* $\Omega(\log n)$*, where* $n = |U|$.

For instances of 2D-JBS that are not in general position, the smallest-disk-first greedy algorithm can have approximation ratio $n$, as shown in Figure 7.3.2.

[4]Lower bounds for the general 2D-JBS problem will also be a topic of Matúš Mihaľák's thesis.

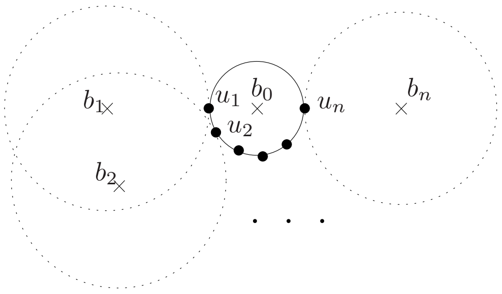


**Figure 7.3.2:** *Lower bound for smallest-disk-first algorithm.*

## 7.4 Open Problems

In this chapter we analyzed the 1D- and 2D-JBS problems that arise in the context of coordinated scheduling in packet data systems. These problems can be split into a selection and a coloring problem. In the one-dimensional case, we have shown that the coloring problem leads to the class of arrow graphs, for which we have discussed its relation to other graph classes and algorithms. For the selection problem we proposed an approach based on LP relaxation with rounding. For the 2D-problem, we have shown its NP-completeness.

The following problems remain still open:

- Is the 1D-JBS problem NP-complete or is there a polynomial time algorithm that solves it?
- Are there constant approximation algorithms for the unconstrained 2D-JBS problem?

# Chapter 8

# Summary of contributions

In this thesis we have studied algorithmic problems that are derived from practical applications as directly as possible. We were able to develop theoretical results with immediate applicability to the original problems. The thesis illustrates the interaction between theory and practice: We start from a practical problem, build a mathematical model for it that leads to new theoretical questions and challenges. The solution to these theory problems then has implications for the original problem or even directly leads to a satisfactory solution. In this context, it has proven crucial to have at one's disposal the whole available spectrum of algorithmic techniques. The reason for this is that depending on the type of the problem different techniques prove to be successful.

The sequential vector packing problem is a good example of a clean combinatorial problem that arises in practice and necessitates the design of new algorithms. We have developed a bicriteria $(\frac{1}{\varepsilon}, \frac{1}{1-\varepsilon})$- and a $(1, 2)$-approximation algorithm. For this algorithm we use the technique of LP-rounding and for its analysis structural properties about the optimal solutions. An NP-completeness proof justifies the use of approximation algorithms. Already this approximation algorithm is readily usable in the original industry setting. However, it requires an LP-solver. In order to stress the practical side even more, we analyze the performance of two natural greedy heuristics similar to those that were initially employed to tackle the problem. Our analysis shows that these heuristics can produce very bad solutions on some instances. We substantiate this theoretical analysis by experiments, by which we also demonstrate that a very simple randomized heuristic outperforms the greedy heuristics. The results for sequential vector packing highlight the interplay between algorithm analysis and real-world problems.

At the core of the optimization of a hub and spoke railway system lies the development of a mathematical model that captures enough of the real world problem to be useful and that is at the same time amenable to a computational approach that can produce solutions in reasonable time. We have presented a sequence of models and discussed their advantages and limitations. In particular, the use of LP-based optimization techniques not only produces solutions but also lower bounds and thus a quality guarantee that is difficult to provide by simple meta-heuristics for the problem at hand. The migration from a branch and cut model to a column generation model has allowed us to incorporate considerably more aspects of the problem without decreasing the size of the instances that we can solve in a given time. In particular, we were able to validate our approaches on real world data of Swiss federal railways SBB Cargo Ltd. We succeeded in producing solutions to these instances. The column generation model, that is part of an ongoing project at the time of writing this thesis, promises to be a useful tool in the calculation of schedules for companies such as SBB Cargo.

For the problem of OVSF-code assignment we have shown the DCA algorithm, which was proposed and referenced in several telecommunications publications to be incorrect both by a concrete counter-example and by an NP-hardness proof which also settled the complexity of the one-step offline CA problem addressed by the DCA algorithm. We also showed that the technique of dynamic programming can be applied to obtain a moderately efficient $n^{O(h)}$-algorithm. More importantly, we showed that it is much more natural to study the problem in an online setting. In particular, an optimal algorithm to the initial problem formulation, which solves an NP-complete problem in every step, is provably no better than a $\Theta(h)$-competitive scheme that we propose. We complemented this result by showing the $\Omega(h)$-competitiveness of other proposed schemes and sketching a resource augmented algorithm with constant competitive ratio. A crucial ingredient to the design of useful algorithms for the OVSF code assignment problem has been the change in perspective from the “classical” one step problem to the online problem.

Similar to the OVSF code assignment problem, the joint base station scheduling problem is inspired by a publication from telecommunications. It led us to the arrow graph class, which in turn is connected to trapezoid graphs. In these graph classes polynomial algorithms for chromatic number, weighted independent set, clique cover, and weighted clique exist as they are perfect. The perfectness of arrow-graphs helps in the design of an approximation algorithm for the one-dimensional version of the problem. Unfortunately, there is no direct correspondence to this result in two dimensions. Still, for the two dimensional case, we were able to prove NP-completeness and to give approximation algorithms in a restricted setting. It should be clear that this approximation algorithm is not suitable for the original application. On the other hand, our theoretical results give a different kind of insight into the original problem: The conceptual idea of a coordination of base stations on a very short time scale that should help to boost the performance of the network leads to hard combinatorial problems that must be solved very fast. Therefore, one might call into question the proposed procedure and search for specialized algorithms that can compute the desired scheduling in real-time only if it still seems desirable to employ the method from an applications point of view.

To sum up, the four problems encountered in this thesis show in different ways how practice and theory are connected and can benefit from each other.

# Appendix A

# Information on Experiments

**Table A.1:** *Characteristics of the computers on which we ran experiments*

| Machine | A | B | C |
|---|---|---|---|
| CPU type | Intel P4 | Intel P4, | AMD Athlon 64 X2/DC 4400+ |
| CPU clock | 3GHz | 3GHz | 1 GHz |
| memory | 2 GB | 3 GB | 4 GB |
| OS | Linux 2.4.22 | Linux 2.4.22 | Linux 2.6.13-15.8-smp |
| CPLEX | 9.0 | 9.0 | 10.0 |
| OPL | 3.7 | 3.7 | - |

## A.1 Setting for Computational Experiments

Table A.1 gives an overview over the computers that we used for the experiments.

## A.2 ILP Formulation

We give here the ILP formulation for the most basic model, Model 0. This formulation models both the routing and the scheduling aspect of the general train optimization problem (cf. Definition 5.1), but it ignores the capacity aspect. It is written in the OPL language, see [71]. It is explained in Section 5.4 and evaluated in Section 5.8.

```
////////////////
//   INPUT    //
////////////////

// number of trains
int+ nbTrains = ...;

// number of vertices
int+ nbVertices = ...;
int+ nbHubs = ...;

// big Ms used in Model
int MtwD = 30;
int MtwA = 36;
int Mdep = 36;
int Mtc  = 36;


int+   hubDistance = ...;
int+   averageSpeed = ...;
int+   hubTravelTime = hubDistance / averageSpeed;

// Ranges
range
   bool                0..1,
   idTrains            [1..nbTrains],
   idVertices          [0..nbVertices-1],
   // we experimented with relaxing some of the bool vars to [0..1] vars
   // if their integrality was implied by other vars
   float oone          [0.0..1.0],
   float departTimes   [0..MtwD],
   float arriveTimes   [0..MtwA],
   float betweenTimes  [0..MtwD];


// Sets
{int} Trains =    {i | i in idTrains};
{int} Vertices = {i | i in idVertices};
{int} Hubs = {i | i in [0..nbHubs-1]};

// Further parameters
// the minimum time that a car needs to be processed in the shunting yard

int+   shuntingTime = ...;
int+   hubShuntingTime = ...;

int+   maxTrainLength = ...;
bool   doGraphics = ...;
```

```
int+ costPerEngine = ...;
int+ costPerKilometer = ...;


struct arc {int i; int j;};
{arc}  Arcs     = ...;
int+ nbArcs = card(Arcs);
int+ length[Arcs] = ...;

int+ times[Arcs];
initialize
   forall(a in Arcs) times[a] = length[a]/averageSpeed;

display times;

struct shipment{
    int index;
    int start;
    int end;
    int volume;
    departTimes earliestDeparture;
    departTimes latestDeparture;
    arriveTimes earliestArrival;
    arriveTimes latestArrival;
};
{shipment}  Shipments  = ...;
display Shipments;

int nbShipments = 0;
initialize
   forall(s in Shipments) {
      s.index = nbShipments;
      nbShipments = nbShipments+1;
   };

// direct path lengths
int directpathscosts[0.. nbShipments-1]= ...;


////////////////
//   MODEL    //
////////////////
// train uses arc on its way to some hub
var bool travelsForth[Trains,Arcs];

// train uses arc on its way from some hub
var bool travelsBack[Trains,Arcs];

// train goes between two hubs
var bool travelsBetween[Trains, Hubs, Hubs];

// train starts at vertex
var bool starts[Trains,Vertices];

// train ends at vertex
var bool ends[Trains,Vertices];

// time at which a train arrives at a station on its way to some hub
var departTimes arrivesForth[Trains, Vertices];
```

```
// time at which a train arrives at a station on its way from some hub
var arriveTimes arrivesBack[Trains, Vertices];

// time at which train z starts a hub hub ride
var betweenTimes startsBetween[Trains];

// direct paths from Shipments
var bool direct[Trains, Shipments];

// train takes shipment and goes to hub
var bool takesForth[Trains, Shipments, Hubs];

// train takes shipment from hub
var bool takesBack[Trains, Shipments, Hubs];

// train takes shipment between hubs
var bool takesBetween[Trains, Shipments, Hubs, Hubs];

// second train depends on first for its front/back journey through h
var bool depFB[Trains,Trains, Hubs];

// second train depends on first for Hub Hub journey through h
var bool depFH[Trains,Trains, Hubs];

// second train depends on first for hub back journey through h
var bool depHB[Trains,Trains, Hubs];


// set branching priorities (we experimented with different settings here)
setting mipsearch{
  forall (z in Trains, a in Arcs) {
     setPriority(travelsForth[z,a],1);
     setPriority(travelsBack[z,a],1);

  };
};


minimize sum(z in Trains, h in Hubs, <u,v> in Arcs: v = h)
        costPerEngine * travelsForth[z,<u,v>]
   + sum(a in Arcs, z in Trains)
        (costPerKilometer* length[a] * (travelsBack[z,a] + travelsForth[z,a]))
   + sum(h in Hubs, hp in Hubs, z in Trains)
        (travelsBetween[z,h,hp] * hubDistance)
   + sum(z in Trains, s in Shipments)
        (directpathscosts[s.index] * direct[z,s])

subject to
{
   // a shipment can only be taken by passing trains
   forall(z in Trains, h in Hubs, v in Vertices, s in Shipments: v = s.start)
      takesForth[z,s,h] <= sum(<i,v> in Arcs) travelsForth[z,<i,v>]
                           + starts[z,v];

   forall(z in Trains, h in Hubs, v in Vertices, s in Shipments: v = s.end)
      takesBack[z,s,h] <= sum( <v, i> in Arcs) travelsBack[z,<v, i>]
                           + ends[z, v];

   forall(z in Trains, s in Shipments, h in Hubs, hp in Hubs)
      takesBetween[z,s,h,hp] <= travelsBetween[z,h,hp];
```

```
// each supply is taken back and forth
forall(s in Shipments)
   sum(z in Trains, h in Hubs) takesForth[z,s,h]
                 + sum(z in Trains) direct[z,s] >= 1;

forall(s in Shipments)
   sum(z in Trains, h in Hubs) takesBack[z,s,h]
                 + sum(z in Trains) direct[z,s]  >= 1;

// any train does at most one direct path or one trip to the hub
forall(z in Trains) {
   sum(s in Shipments) direct[z,s]
                 + sum(<i,v> in Arcs: v in Hubs) travelsForth[z,<i,v>]=1;
   sum(s in Shipments) direct[z,s]
                 + sum(<v,i> in Arcs: v in Hubs) travelsBack[z,<v,i>] =1;
};

// inflow outflow trains at nodes
 forall(z in Trains, v in Vertices: v not in Hubs)
   sum(<i,v> in Arcs) tra velsForth[z,<i,v>] + starts[z,v] =
      sum(<v,j> in Arcs) travelsForth[z,<v,j>]
         + sum(s in Shipments: s.start=v) direct[z,s];


 forall(z in Trains, v in Vertices: v not in Hubs)
   sum(<i,v> in Arcs) travelsBack[z,<i,v>]
     + sum(s in Shipments: s.end = v) direct[z,s] =
      sum(<v,j> in Arcs) travelsBack[z,<v,j>] + ends[z,v];


// inflow outflow trains at hubs
 forall(h in Hubs, z in Trains)
    sum(<i,h> in Arcs) travelsForth[z,<i,h>]
     + sum(hp in Hubs) travelsBetween[z,hp,h] =
           sum(<h,i> in Arcs) travelsBack[z,<h,i>]
             + sum(hp in Hubs) travelsBetween[z,h,hp];

// maximum Train length
forall(z in Trains) {
   ( sum(h in Hubs, s in Shipments) s.volume *  takesForth[z,s,h] )
      + sum(s in Shipments) s.volume * direct[z,s]  <= maxTrainLength;
   ( sum(h in Hubs, s in Shipments) s.volume *  takesBack[z,s,h] )
      + sum(s in Shipments) s.volume * direct[z,s]  <= maxTrainLength;
   sum(h in Hubs, hp in Hubs, s in Shipments)
     s.volume * takesBetween[z,s,h,hp]  <= maxTrainLength};


// inflow outflow shipments at hubs
 forall(h in Hubs, s in Shipments)
     sum(z in Trains) takesForth[z,s,h]
         + sum(t in Trains, hp in Hubs: hp<>h)
     takesBetween[t,s,hp,h] =
            sum(z in Trains) takesBack[z,s,h]
          + sum(t in Trains, hp in Hubs: hp<>h) takesBetween[t,s,h,hp];

// couple takesForth (takesBack) w/ travelsforth (travelsBack) at hubs
forall (h in Hubs, z in Trains, s in Shipments){
   takesForth[z,s,h] <= sum(<u,v> in Arcs: v=h) travelsForth[z,<u,v>];
   takesBack[z,s,h] <= sum(<u,v> in Arcs: u=h) travelsBack[z,<u,v>];
};


// Capacity at hubs
// is not modeled here!!


// time windows
forall(z in Trains, s in Shipments)
   arrivesForth[z,s.start] + (1- sum(h in Hubs) takesForth[z,s,h]) * MtwD
             >= s.earliestDeparture;

forall(z in Trains, s in Shipments)
   arrivesForth[z,s.start] - (1-sum(h in Hubs) takesForth[z,s,h]) * MtwD
             <= s.latestDeparture;

forall(z in Trains, s in Shipments)
   arrivesBack[z,s.end] + (1- sum(h in Hubs) takesBack[z,s,h]) * MtwA
             >= s.earliestArrival;

forall(z in Trains, s in Shipments)
   arrivesBack[z,s.end] - (1- sum(h in Hubs) takesBack[z,s,h]) * MtwA
             <= s.latestArrival;

// trains start and end at most once
forall(z in Trains)
   sum(v in Vertices) starts[z,v] <= 1;

forall(z in Trains)
   sum(v in Vertices) ends[z,v] <= 1;

// time consistency at nodes
      // pure travel time (not redundant!)
forall(z in Trains, <i,v> in Arcs)
   Mtc * (1 - travelsForth[z,<i,v>]) + arrivesForth[z,v]
             >= arrivesForth[z,i] + times[<i,v>];

     // + shunting time
forall(z in Trains, s in Shipments, <u,v> in Arcs: u=s.start)
   Mtc * (1 - travelsForth[z,<u,v>]) + arrivesForth[z,v] >=
       arrivesForth[z,u] + times[<u,v>]
         + sum(h in Hubs) (takesForth[z,s,h] * shuntingTime);

    // pure travel time (not redundant!)
forall(z in Trains, <i,v> in Arcs)
   Mtc * (1 - travelsBack[z,<i,v>]) + arrivesBack[z,v]
             >= arrivesBack[z,i] + times[<i,v>];

    // + shunting time
forall(z in Trains, s in Shipments, <u,v> in Arcs: u = s.end)
   Mtc * (1 - travelsBack[z,<u,v>]) + arrivesBack[z,v] >=
       arrivesBack[z,u] + times[<u,v>]
         + sum(h in Hubs) takesBack[z,s,h] * shuntingTime;


// time consistency at hub wrt trains
forall(h in Hubs, z in Trains, zp in Trains)
   Mdep * (1 - depFB[z,zp,h]) + arrivesBack[zp,h]
         >= arrivesForth[z,h] + hubShuntingTime;
   forall(h in Hubs, z in Trains, zp in Trains)
      Mdep * (1 - depFH[z,zp,h]) + startsBetween[zp]
            >= arrivesForth[z,h] + hubShuntingTime;

   forall(h in Hubs, z in Trains, zp in Trains)
      Mdep * (1 - depHB[z,zp,h]) + arrivesBack[zp,h]
            >= startsBetween[z] + hubShuntingTime + hubTravelTime;


   // dependence on engine
   forall(z in Trains, h in Hubs)
      depFB[z,z,h] >= sum(<i,h> in Arcs) travelsForth[z,<i,h>]
         + sum(<h,i> in Arcs) travelsBack[z,<h,i>] - 1;

   forall(z in Trains, h in Hubs)
      depFH[z,z,h] >= sum(<i,h> in Arcs) travelsForth[z,<i,h>]  +
          + sum(hp in Hubs) travelsBetween[z,h,hp] - 1;

   forall(z in Trains, h in Hubs, s in Shipments)
      depHB[z,z,h] >= sum(<h,i> in Arcs) travelsBack[z,<h,i>]
          + sum(hp in Hubs) travelsBetween[z,hp,h] - 1;

   // couple dep and takesforth / back
   forall(z in Trains, zp in Trains, s in Shipments, h in Hubs) {
      depFB[z,zp,h] >= takesForth[z,s,h] + takesBack[zp,s,h] - 1;
      depFH[z,zp,h] >= takesForth[z,s,h]
                           + sum(hp in Hubs) takesBetween[zp,s,h,hp] - 1;
      depHB[z,zp,h] >= sum(hp in Hubs) takesBetween[z,s,hp,h]
                           + takesBack[zp,s,h] - 1};


   // symmetry breaking constraints
   forall(z in Trains, zp in Trains: z < zp)
      sum(v in Vertices) starts[z,v] >= sum(v in Vertices) starts[zp,v];

   forall(v in Vertices, z in Trains, zp in Trains: z < zp)
      sum(vp in Vertices: vp <= v) starts[z,vp]
                    >= sum(vp in Vertices: vp <= v) starts[zp,vp];


   // basic initializations

   forall(h in Hubs, z in Trains)
      travelsBetween[z,h,h] = 0;

   // some valid inequalities

   forall(z in Trains)
      sum(h in Hubs, hp in Hubs) travelsBetween[z,h,hp] <= 1;
   forall(z in Trains, s in Shipments)
      sum(h in Hubs, hp in Hubs: h<>hp) takesBetween[z,s,h,hp] <= 1;

   forall(z in Trains, zp in Trains){
      sum(h in Hubs) depFB[z,zp,h] <= 1;
      sum(h in Hubs) depFH[z,zp,h] <= 1;
      sum(h in Hubs) depHB[z,zp,h] <=1;
   }
};

////////////////
```

```
//   OUTPUT   //
////////////////

display(z in Trains, s in Shipments: direct[z,s]>0 ) direct[z,s];
display(z in Trains, a in Arcs: travelsForth[z,a]>0) travelsForth[z,a];
display(z in Trains, a in Arcs: travelsBack[z,a]>0) travelsBack[z,a];
display(z in Trains, v in Vertices: arrivesForth[z,v]>0) arrivesForth[z,v];
display(z in Trains, v in Vertices: arrivesBack[z,v]>0) arrivesBack[z,v];
display(z in Trains: startsBetween[z]>0) startsBetween[z];
display(z in Trains, s in Shipments, h in Hubs: takesForth[z,s,h]>0)
            takesForth[z,s,h];
display(z in Trains, s in Shipments, h in Hubs: takesBack[z,s,h]> 0)
            takesBack[z,s,h];
display (z in Trains, s in Shipments, h in Hubs, hp in Hubs:
     takesBetween[z,s,h,hp]>0) takesBetween[z,s,h,hp];
display(z in Trains, zp in Trains, h in Hubs: depFB[z,zp,h]>0) depFB[z,zp,h];
display(z in Trains, zp in Trains, h in Hubs: depFH[z,zp,h]>0) depFH[z,zp,h];
display(z in Trains, zp in Trains, h in Hubs: depHB[z,zp,h]>0) depHB[z,zp,h];
display(z in Trains, h in Hubs, hp in Hubs: travelsBetween[z,h,hp]>0)
            travelsBetween[z,h,hp] ;
display(z in Trains, v in Vertices: starts[z,v]>0) starts[z,v];
display(z in Trains, v in Vertices: ends[z,v]>0) ends[z,v];
```

# Bibliography

[1] Karen Aardal and Stan van Hoesel. Polyhedral techniques in combinatorial optimization II: Computations. report UU-CS-1995-42, Utrecht University, 1995.

[2] Karen Aardal and Stan van Hoesel. Polyhedral techniques in combinatorial optimization I: Theory. *Statistica Neerlandica*, 50:3–26, 1996.

[3] Tobias Achterberg. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, 2004. `http://www.zib.de/Publications/abstracts/ZR-04-19/`.

[4] Fumiyuki Adachi, Mamoru Sawahashi, and Koichi Okawa. Tree structured generation of orthogonal spreading codes with different length for forward link of DS-CDMA mobile radio. *Electronic Letters*, 33(1):27–28, January 1997.

[5] Juri Adamek. *Foundation of Coding*. Wiley, 1991.

[6] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice Hall, 1993.

[7] Susanne Albers. *Competitive Online Algorithms*. BRICS Lecture Series, September 1996.

[8] Rujipun Assarut, Milda G. Husada, Ushio Yamamoto, and Yoshikuni Onozato. Data rate improvement with dynamic reassignment of spreading codes for DS-CDMA. *Computer Communications*, 25(17):1575–1583, 2002.

[9] Rujipun Assarut, Kenichi Kawanishi, Rajkumar Deshpande, Ushio Yamamoto, and Yoshikuni Onozato. Performance evaluation of orthogonal

variable-spreading-factor code assignment schemes in W-CDMA. In *ICC 2002 conference*, volume 5, pages 3050–3054, 2002.

[10] Philippe Augerat, José M. Belenguer, Enrique Benavent, Angel Corbéran, and Denis Naddef. Separating capacity constraints in the CVRP using tabu search. *European Journal of Operations Research*, 106:546–557, 1998.

[11] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation*. Springer, 1991.

[12] Masri Ayob, Peter Cowling, and Graham Kendall. Optimisation of surface mount placement machines. In *Proceedings of IEEE International Conference on Industrial Technology*, pages 486–491, 2002.

[13] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[14] Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977.

[15] Ulrich Blasum, Michael R. Bussieck, Winfried Hochstättler, Christoph Moll, Hans-Helmut Scheel, and Thomas Winter. Scheduling trams in the morning. *Mathematical Methods of Operations Research*, 49(1):137–148, 1999.

[16] Ulrich Blasum and Winfried Hochstättler. Application of the branch and cut method to the vehicle routing problem. Technical Report zpr2000-386, Zentrum für angewandte Informatik, Köln, 2000.

[17] Natashia Boland, John Detheridge, and Irina Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. electronically available, 2005.

[18] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[19] Julien Bramel and David Simchi-Levi. *The Vehicle Routing Problem*, chapter Set-Covering Algorithms for the Capacitated VRP. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[20] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: A survey*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[21] Alberto Caprara and Matteo Fischetti. *Annotated Bibliographies in Combinatorial Optimization*, chapter Branch-and-Cut Algorithms, pages 45–64. Discrete Mathematics and Optimization. Wiley, 1997.

[22] Hasan Çam. Nonblocking OVSF codes and enhancing network capacity for 3G wireless and beyond systems. *Intrenational Conference on Third Generation Wireless and Beyond*, pages 148–153, Mai 2002.

[23] Alberto Ceselli and Giovanni Righini. A branch and price algorithm for the capacitated p-median problem. *Networks*, 45(3):125–142, 2005.

[24] Wen-Tsuen Chen and Shih-Hsien Fang. An efficient channelization code assignment approach for W-CDMA. *IEEE Conference on Wireless LANs and Home Networks*, 2002.

[25] Wen-Tsuen Chen, Ya-Ping Wu, and Hung-Chang Hsiao. A novel code assignment scheme for W-CDMA systems. *Proc. of the 54th IEEE Vehicular Technology Society Conference*, 2:1182–1186, 2001.

[26] Ying-Chung Chen and Wen-Shyen E. Chen. Implementation of an efficient channelization code assignment algorithm in 3G WCDMA. 2003.

[27] Nicos Christofides, Aristide Mingozzi, and Paolo Toth. Exact algorithms for vehicle routing. *Mathematical Programming*, 20:255–282, 1981.

[28] Julia Chuzhoy and Joseph Naor. New hardness results for congestion minimization and machine scheduling. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing (STOC'04)*, pages 28–34, 2004.

[29] Vašek Chvátal. *Linear Programming*. Freeman, 1980.

[30] Mark Cielibak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer. Scheduling jobs on a minimum number of machines. In *Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*, pages 217–230. Kluwer, 2004.

[31] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[32] Edward G. Coffman Jr., Michael R. Garey, and David S. Johnson. *Algorithm Design for Computer System Design*, chapter Approximation Algorithms for Bin Packing: An updated Survey, pages 49–106. Springer, 1984.

[33] Edward G. Coffman Jr., Michael R. Garey, and David S. Johnson. *Approximation Algorithms*, chapter Approximation Algorithms For Bin Packing: A Survey, pages 46–93. PWS Publishing Company, 1997.

[34] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, pages 151–158, 1971.

[35] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Discrete Mathematics and Optimization. Wiley, 1998.

[36] Don Coppersmith and Prabhakar Raghavan. Multidimensional on-line bin packing: Algorithms and worst-case analysis. *Operations Research Letters*, 4:48–57, 1989.

[37] Jean-François Cordeau, Guy Desaulniers, Jacques Desrosiers, Marius M. Solomon, and François Soumis. *The Vehicle Routing Problem*, chapter VRP with Time Windows. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[39] Elias Dahlhaus, Peter Horák, Mirka Miller, and Joseph F. Ryan. The train marshalling problem. *Discrete Applied Mathematics*, 103(1-3):41–54, 2000.

[40] Elias Dahlhaus, Fredrik Manne, Mirka Miller, and Joseph F. Ryan. Algorithms for combinatorial problems related to train marshalling. In *Proceedings of AWOCA 2000, In Hunter Valley*, pages 7–16, July 2000.

[41] Emilie Danna and Claude Le Pape. *Column Generation*, chapter Branch-and-Price Heuristics: A case study on the vehicle routing problem with time windows, pages 99–129. Springer-Verlag, 2005.

[42] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic (2-2/(k+1))n algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

[43] Suman Das, Harish Viswanathan, and Gee Rittenhouse. Dynamic load balancing through coordinated scheduling in packet data systems. In *Proceedings of Infocom'03*, 2003.

[44] Guy Desaulniers, Jacques Desrosiers, Arielle Lasry, and Marius M. Solomon. *Computer-Aided Transit Scheduling*, chapter Crew Pairing for a Regional Carrier, pages 19–41. Lecture Notes in Economics and Methematical Systems. Springer-Verlag, 1999.

[45] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. *Essays and Surveys in Metaheuristics*, chapter Accelerating strategies in column generation methods for vehicle routing and crew scheduling problems, pages 309–324. Kluwer, 2001.

[46] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors. *Column Generation*. Springer-Verlag, 2005.

[47] Russel G. Downey and Michael R. Fellows. *Parametrized Complexity*. Monographs in Computer Science. Springer-Verlag, 1999.

[48] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.

[49] Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Marc Nunkesser, Gábor Szabó, and Peter Widmayer. An algorithmic view on OVSF code assignment. TIK-Report 173, Computer Engineering and Networks Laboratory (TIK), ETH Zürich, August 2003. Available electronically at ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report173.pdf.

[50] Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Marc Nunkesser, Gábor Szabó, and Peter Widmayer. An algorithmic view on OVSF code assignment. *in Proc. of the 21st Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 2996:270–281, 2004.

[51] Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Marc Nunkesser, Gábor Szabó, and Peter Widmayer. Joint Base Station Scheduling. *in Proc. of the 2nd International Workshop on Approximation and Online Algorithms*, LNCS 3351:225–238, 2004.

[52] Thomas Erlebach, Riko Jacob, Matúš Mihaľák, Marc Nunkesser, Gábor Szabó, and Peter Widmayer. Joint base station scheduling. Technical Report 461, ETH Zürich Institute of Theoretical Computer Science, 2004.

[53] Thomas Erlebach, Riko Jacob, Matúš Mihal̆ák, Marc Nunkesser, Gábor Szabó, and Peter Widmayer. An algorithmic view on OVSF code assignment. *Algorithmica*, 2006. to appear.

[54] Thomas Erlebach and Frits C.R. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *Algorithmica*, 46:27–53, 2001.

[55] Romano Fantacci and Saverio Nannicini. Multiple access protocol for integration of variable bit rate multimedia traffic in UMTS/IMT-2000 based on wideband CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1441–1454, August 2000.

[56] Uriel Feige, David Peleg, and Guy Kortsarz. The dense $k$-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[57] Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 44(3):216–229, 2004.

[58] Stefan Felsner, R. Müller, and L. Wernisch. Trapezoid graphs and generalizations, geometry and algorithms. *Discrete Applied Mathematics*, 74:13–32, 1997.

[59] Amos Fiat and Gerhard J. Woeginger. *Online Algorithms*. Lecture Notes in Computer Science. Springer-Verlag, 1998.

[60] Carl E. Fossa Jr. *Dynamic Code Sharing Algorithms for IP Quality of Service in Wideband CDMA 3G Wireless Networks*. PhD thesis, Virginia Polytechnic Institute and State University, April 2002.

[61] Carl E. Fossa Jr. and Nathaniel J. Davis IV. Dynamic code assignment improves channel utilization for bursty traffic in 3G wireless networks. *IEEE International Communications Conference*, 2002.

[62] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[63] Michael Gatto, Riko Jacob, and Marc Nunkesser. Optimization of a railway hub-and-spoke system: Routing and shunting. In Ioanis Chatzigiannakis and Sotiris Nikoletseas, editors, *Poster Proceceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*, pages 15–26. CTI-Press, 2005.

[64] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[65] Ralph E. Gomory. *Combinatorial Analysis*, chapter Solving linear programming problems in integers, pages 211–215. Proceedings of Symposia in Applied Mathematics X. American Mathematical Society, 1960.

[66] Albert Gräf, Martin Stumpf, and Gerhard Weißenfels. On coloring unit disk graphs. *Algorithmica*, 20(3):277–293, March 1998.

[67] Martin Grötschel, Lászlo Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[68] Martin Grötschel, Lászlo Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, Berlin, 1988.

[69] Alexander Hall, Steffen Hippler, and Martin Skutella. Multicommodity flows over time: Efficient algorithms and complexity. In *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming (ICALP'03)*, LNCS 2719, pages 397–409. Springer-Verlag, June 2003.

[70] Randolph Hall. On the road to recovery. *OR/MS Today*, June 2004. available at `www.lionhrtpub.com/orms/orms-6-04/frsurvey.html`.

[71] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, 1999.

[72] Wolfgang Hiller. *Rangierbahnhöfe*. Transpress VEB Verlag für Verkehrswesen, 1983.

[73] Dorit S. Hochbaum, editor. *Approximation Algorithms*. PWS Publishing Company, 1997.

[74] Harri Holma and Antti Toskala. *WCDMA for UMTS*. Wiley, 2001.

[75] Brian Kallehauge, Jesper Larsen, Oli B. G. Madsen, and Marius M. Solomon. *Column Generation*, chapter Vehicle Routing Problem with Time Windows, pages 67–98. Springer-Verlag, 2005.

[76] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 214–221, 1995.

[77] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[78] Anthony C. Kam, Thit Minn, and Kai-Yeung Siu. Supporting rate guarantee and fair access for bursty data traffic in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 19(11):2121–2130, November 2001.

[79] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[80] Jon M. Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[81] Bernhard Korte and Jens Vygen. *Combinatorial Optimization*. Springer-Verlag, 3rd edition, 2006.

[82] Laszlo Ladányi, Ted R. Ralphs, and Leslie E. Trotter. *Computational Combinatorial Optimization*, chapter Branch, Cut, and Price: Sequential and Parallel. Springer-Verlag, 2001.

[83] Jaana Laiho, Achim Wacker, and Tomáš Novosad. *Radio Network Planning and Optimisation for UMTS*. Wiley, New York, 2002.

[84] Gilbert Laporte. *Annotated Bibliographies in Combinatorial Optimization*, chapter Vehicle Routing, pages 223–240. Discrete Mathematics and Optimization. Wiley, 1997.

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

[86] Gilbert Laporte, Yves Nobert, and Martin Desrochers. Optimal routing under capacity and distance restrictions. *Operations Research*, 33:1050–1073, 1985.

[87] Eugene E. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys. *The Travelling Salesman Problem*. Wiley, 1985.

[88] Marco E. Lübbecke and Uwe T. Zimmermann. Shunting minimal rail car allocation. *Computational Optimization and Applications*, 31(3):295–308, 2005.

[89] Ramon M. Lentink. *Algorithmic Decision Support for Shunt Planning*. PhD thesis, Erasmus Research Institute of Management (ERIM), 2006.

[90] Fillia S. Makedon, Christos H. Papadimitriou, and Ivan H. Sudborough. Topological bandwidth. *SIAM Journal on Algebraic and Discrete Methods*, 6(3):418–444, July 1985.

[91] Richard Kipp Martin. *Large Scale Linear and Integer Optimization*. Kluwer, 1999.

[92] Kurt Mehlhorn and Stefan Näher. *LEDA a Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[93] Thit Minn and Kai-Yeung Siu. Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. *IEEE Journal on selected areas in communications*, 18(8):1429–1440, 2000.

[94] Burkhard Monien and Ivan H. Sudborough. Min cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, 58(1-3):209–229, 1988.

[95] Matthias Müller-Hannemann and Karsten Weihe. Moving policies in cyclic assembly–line scheduling. *Theoretical Computer Science*, 351:425–436, 2006.

[96] Denis Naddef and Giovanni Rinaldi. *The Vehicle Routing Problem*, chapter Branch-and-Cut Algorithms for the Capacitated VRP. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[97] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Discrete Mathematics and Optimization. Wiley, 1988.

[98] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[99] Albert Nijenhuis and Herbert Wilf. *Combinatorial Algorithms*. Academic Press, 2nd edition, 1978.

[100] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[101] Cynthia Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 140–149, 1997.

[102] David Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45:758–767, 1997.

[103] Gerhart Potthoff. *Verkehrsströmungslehre, Betriebstechnik des Rangierens*, volume 2. Transpress VEB Verlag für Verkehrswesen, 1977.

[104] Ted R. Ralphs. Branch and cut. `www.branchandcut.org`.

[105] Ted R. Ralphs. Branch and cut references. `http://branchandcut.org/reference.htm#implementations`.

[106] Ted R. Ralphs. Symphony 5.0. `www.branchandcut.org/SYMPHONY`.

[107] Ted R. Ralphs, Leonid Kopman, William R. Pulleyblank, and Leslie E. Trotter. On the capacitated vehicle routing problem. *Mathematical Programming*, 94(2–3):343–359, 2003.

[108] Venkatesh Raman, Saket Saurabh, and C. R. Subramanian. Faster fixed parameter tractable algorithms for undirected feedback vertex set. *ISAAC*, pages 241–248, 2002.

[109] Angelos N. Rouskas and Dimitrios N. Skoutas. OVSF codes assignment and reassignment at the forward link of W-CDMA 3G systems. *PIMRC 2002*, 2002.

[110] Louis M. Rousseau, Michel Gendreau, and Dominique Feillet. Interior point stabilization for column generation. Technical report, Centre de recherche sur les transports, 2003.

[111] Frank Ruskey. Combinatorial generation. 2005.

[112] SBB. Cargo express. `www.sbbcargo.com/en/index/ang_produkte/ang_produkte_express.htm`.

[113] Armin Scholl. *Balancing and Sequencing of Assembly Lines*. Physica-Verlag, Heidelberg, 2nd edition, 1999.

[114] Uwe Schöning. A probabilistic algorithm for $k$-SAT based on limited local search and restart. *Algorithmica*, 32:615–623, 2002.

[115] Alexander Schrijver. *Theory of Linear and Integer Programming*. Discrete Mathematics and Optimization. Wiley, 1986.

[116] David B. Shmoys. *Approximation Algorithms*, chapter Cut Problems and Their Application to Divide-and-Conquer, pages 192–235. PWS Publishing Company, 1997.

[117] François Soumis. *Annotated Bibliographies in Combinatorial Optimization*, chapter Decomposition and Column Generation, pages 115–126. Discrete Mathematics and Optimization. Wiley, 1997.

[118] Frits C.R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2:215–227, 1999.

[119] Jeremy P. Spinrad. *Efficient Graph Representations*, volume 19 of *Field Institute Monographs*. AMS, 2003.

[120] Gábor Szabó. *Optimization Problems in Mobile Communication*. No. 16207, ETH Zürich, 2005.

[121] Marco Tomamichel. Algorithmische Aspekte von OVSF Code Assignment mit Schwerpunkt auf Offline Code Assignmnent. Semester thesis, Computer Engineering and Networks Laboratory (TIK), ETH Zürich, 2004.

[122] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*, chapter An overview of vehicle routing problems. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[123] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*, chapter Branch-and-Bound Algorithms for the Capacitated VRP. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[124] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2002.

[125] Wouter M. C. van Wezel and Jan Riezebos. Algorithmic support for human rail shunting planners. In *Proceedings of the 18th International Conference on Production Research*, July 2005.

[126] François Vanderbeck. *Decomposition and Column Generation for Integer Programs*. PhD thesis, Université Catholique de Louvain, September 1994.

[127] Robert J. Vanderbei. *Linear Programming, Foundations and Extensions*. Kluwer, 2nd edition, 2001.

[128] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[129] T. S. Wee and M. J. Magazine. Assembly line balancing as generalized bin-packing. *Operations Research Letters*, 1:56–58, 1982.

[130] Ingo Wegener. *Complexity Theory. Exploring the Limits of Efficient Algorithms*. Springer-Verlag, 2005.

[131] Emo Welzl. Boolean satisfiability—combinatorics and algorithms. unpublished lecture notes, 2005.

[132] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001.

[133] Laurence A. Wolsey. *Integer Programming*. Discrete Mathematics and Optimization. Wiley, 1998.

[134] Jian Yang and Joseph Y.-T. Leung. The ordered open-end bin-packing problem. *Operations Research*, 51(5):759–770, 2003.

# Curriculum Vitae

## Marc Nunkesser

born on May 29, 1976 in Dortmund, Germany

| | |
|---|---|
| 2002 – 2006 | **PhD Student at ETH Zürich** |
| 1998– 2002 | **Diploma in Computer Science** |
| | University of Dortmund, Germany |
| 1997 – 1998 | **Study of Computer Science** |
| | Institut National des Sciences Appliquées de Lyon, France |
| 1995 – 1997 | **Intermediate Diploma** |
| | University of Dortmund, Germany |
| 1986 – 1995 | **Abitur at Goethe Gymnasium** |
| | Dortmund, Germany |