Generic Branch-Cut-and-Price

Diplomarbeit
bei PD Dr. M. Lübbecke

vorgelegt von Gerald Gamrath [1]
Fachbereich Mathematik der
Technischen Universität Berlin

Berlin, 16. März 2010

[1]Konrad-Zuse-Zentrum für Informationstechnik Berlin, gamrath@zib.de

Contents

Acknowledgments

First of all, I want to thank my parents and my brother for their support and encouragement during my academic studies.

Thanks to Marco E. Lübbecke and Marc E. Pfetsch for leading my studies into the direction that is now covered in this thesis. Furthermore, I wish to thank Marco E. Lübbecke for supervising this thesis, for always being available for questions and answering all of them, and for providing me with many helpful comments about earlier drafts of this thesis. Additionally, I want to thank Prof. Dr. Martin Grötschel for awakening my interest for combinatorial optimization and providing the wonderful working athmosphere at Zuse Institute Berlin.

I wish to thank Timo Berthold, Stefan Heinz, Jens Schulz, Michael Winkler, and Kati Wolter for reading earlier versions of this thesis, their professional advice and many helpful suggestions. Thanks to Tobias Achterberg for many hints concerning implementational issues and to Bernd Olthoff for answering my linguistical questions.

Last but not least, I want to thank my lovely girlfriend Inken Olthoff. She read most of this thesis and provided me with many helpful comments concerning content and clarity of presentation. Thanks a lot for your support and for being there for me whenever I needed it!

Chapter 1

Introduction

Many real-world optimization problems can be formulated as mixed integer programs (MIPs). Although solving MIPs is computationally hard [86], state-of-the-art MIP solvers—commercial and non-commercial ones—are able to solve many of these problems in a reasonable amount of time (see e.g., Koch [55]). A widely used technique to solve MIPs is the branch-and-cut paradigm which is employed by most MIP solvers.

In this thesis, we regard a different, but related method to solve MIPs, namely the branch-and-price method and its extension, the branch-cut-and-price method. Their success relies on exploiting problem structures in a MIP via a decomposition. The problem is split into a coordinating problem and one or more typically well structured subproblems that can often be solved rather efficiently. For many huge and extremely difficult, but well structured combinatorial optimization problems, this approach leads to a remarkably better performance than a branch-and-cut algorithm.

While there exist very effective generic implementations of branch-and-cut, almost every application of branch-(cut-)and-price is *ad hoc*, i.e., problem specific. Therefore, using a branch-(cut-)and-price algorithm usually comes along with a much higher implementational effort. In recent years, there has been a development towards the implementation of a generic branch-(cut-)and-price solver. Such a solver should ideally detect the structure of a problem, perform the decomposition—if promising—and solve the problem without further user interaction. An actual implementation of such a fully automated solving process is still a long way off. However, there are codes in development, e.g., `DIP` [78] and `BaPCod` [93], that just require the user to define the structure of the problem, before an automated branch-(cut-)and-price solving process is started.

Typically, such a generic implementation does not achieve the performance of a problem specific one. However, it ideally incorporates sophisticated acceleration strategies and other expert knowledge that would be missing in a basic problem specific implementation and that can partially compensate the disadvantage due to the generic approach. A generic implementation provides the possiblity to solve a problem with a branch-(cut-)and-

price algorithm without any implementational effort and enables researchers to easily test new ideas.

This thesis deals with the generic branch-cut-and-price solver GCG that extends the existing non-commercial state-of-the-art MIP solver and branch-and-price *framework* SCIP [3] to a branch-and-price *solver*. GCG was developed by the author of the thesis and meets the aforementioned demands, i.e., for a given structure, it performs a decomposition and solves the resulting reformulation with a branch-cut-and-price algorithm. Actually, it still takes into account the original problem and solves both problems simultaneously, profiting from the additional information.

We present the theoretical background, implementational details, and computational results concerning the solver GCG. Computations are carried out for four classes of problems that are known to fit well into the branch-and-price approach. We investigate whether even a generic approach to branch-cut-and-price is still more effective than a state-of-the-art branch-and-cut MIP solver for these problems.

Outline of the thesis

In the remainder of this chapter we present some basic definitions, give a short summary of the history of branch-and-price and review current developments concerning this topic.

The foundation of the generic branch-cut-and-price approach presented in this thesis is the Dantzig-Wolfe decomposition for MIPs which we discuss in Chapter 2.

After that, in Chapter 3 we present the branch-cut-and-price framework SCIP which is the basis of our implementation. Furthermore, we describe the general structure of GCG and some general information about the computational experiments that we conducted.

The next three chapters focus on the most important parts of a branch-cut-and-price solver. The solving process of the employed relaxation by column generation is treated in Chapter 4. Chapter 5 describes how this is combined with a branch-and-bound approach in order to compute an optimal solution. Furthermore, in Chapter 6, we describe how to include cutting plane generation to obtain a branch-cut-and-price algorithm. In each of these chapters, we first present the theoretical background, followed by implementational details and some computational results.

Chapter 7 deals with the overall performance of the branch-cut-and-price solver GCG and the impact of the features mentioned in the previous chapters. Since SCIP with default plugins is a state-of-the-art branch-and-cut based MIP solver, we also draw a comparison between the results obtained by GCG and SCIP in order to assess the effectiveness of the generic branch-cut-and-price approach.

Finally, we summarize the contents of this thesis in Chapter 8 and present concluding remarks as well as directions for further research.

In the appendix, we present a German summary, survey the symbols used in this thesis, define the classes of problems used for our computational experiments, and present detailed computational results.

1.1 Definitions

In this section, we define the most important terms that we use in this thesis. For a more detailed introduction into combinatorial optimization, linear and mixed integer programming we refer to [18, 40, 85]. The notation used in this thesis is summarized in Appendix B.

Problem definitions and some polyhedral theory

For a given set of real-valued variables, a *linear program* is an optimization problem that either minimizes or maximizes a linear objective function, subject to some linear equations or inequalities. Using various transformations, we can transform each linear program into the form that is presented in the following definition.

Definition 1.1 (Linear Program)
Let $n, m \in \mathbb{N}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. An optimization problem of the form

$$\begin{aligned} \min \quad & c^T x \\ s.t. \quad & Ax \geq b \\ & x \in \mathbb{R}^n_+ \end{aligned}$$

is called a linear program (LP).

Note that throughout this thesis, we denote by $\mathbb{Z}_+$, $\mathbb{Q}_+$, and $\mathbb{R}_+$ the non-negative integer, rational, and real numbers, respectively.

The set of solutions to an LP forms a polyhedron:

Definition 1.2 (Polyhedron [69])
Let $n \in \mathbb{N}$.

- *A* polyhedron *$P \subseteq \mathbb{R}^n$ is the set of points that satisfy a finite number $m \in \mathbb{N}$ of linear inequalities; that is, $P = \{x \in \mathbb{R}^n \mid Ax \geq b\}$, where (A, b) is an $m \times (n+1)$ matrix.*

- *A polyhedron is said to be* rational, *if there exists $m' \in \mathbb{N}$ and an $m' \times (n+1)$ matrix (A', b') with rational coefficients such that $P = \{x \in \mathbb{R}^n \mid A'x \leq b'\}$.*

- *A polyhedron is called* bounded *if there exists $\omega \in \mathbb{R}_+$ such that $P \subseteq \{x \in \mathbb{R}^n \mid -\omega \leq x_j \leq \omega \text{ for } j = 1, \ldots, n\}$. A bounded polyhedron is called* polytope.

In this thesis, we restrict ourselves to rational polyhedra and assume rational-valued matrices and vectors. This is actually no limitation in practice since computers are restricted to rational numbers, anyway.

When adding integrality restrictions to a part of the variables of an LP, we get a *mixed integer program*:

Definition 1.3 (Mixed Integer Program)
Let $n, m \in \mathbb{N}$, $c \in \mathbb{Q}^n$, $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, and $I \subseteq [n] := \{1, \dots, n\}$. An optimization problem of the form

$$\begin{aligned} min \quad & c^T x \\ s.t. \quad & Ax \geq b \\ & x \in \mathbb{Q}^n_+ \\ & x_i \in \mathbb{Z} \qquad \forall i \in I \end{aligned} \tag{1.1}$$

is called mixed integer program *(MIP).*

Like in the case of linear programs, there are several variations of mixed integer programs like maximization problems and problems containing equality constraints, but all of these can be transformed to the previously stated form, see e.g., [39]. Since each MIP in maximization form can be transferred into a minimization problem by multiplying the objective function coefficients by minus one, we restricted ourselves to minimization problems in the following. Throughout this thesis, a solution is called integral, if and only if it satisfies the integrality restrictions, even if continuous variables may have fractional values.

We distinguish the following special cases of MIPs:

Definition 1.4
A MIP of form (1.1) is called

- *an* integer program (IP) *if $I = [n]$*
- *and a* binary program (BP) *if it is an IP and $x_i \in \{0, 1\}$ $\forall i \in I$.*

The set $X = \{x \in \mathbb{Q}^n_+ \mid Ax \geq b, x_i \in \mathbb{Z}\ \forall i \in I\}$ of solutions to a MIP is a subset of the polyhedron $P = \{x \in \mathbb{Q}^n_+ \mid Ax \geq b\}$. The latter is the set of solutions to the so-called *LP relaxation* of the MIP, i.e., the LP that we obtain by discarding the integrality restrictions of the MIP. Due to the integrality restrictions, the set X is not a polyhedron. However, since we assumed rational data, its convex hull $conv(X)$ is a polyhedron:

Theorem 1.5 ([65])
If P is a rational polyhedron and $X = P \cap \{x \in \mathbb{Q}^n_+ \mid x_i \in \mathbb{Z}\ \forall i \in I\} \neq \emptyset$, then $conv(X)$ is a rational polyhedron whose extreme points are a subset of X and whose extreme rays are the extreme rays of P.

This does not neccessarily hold for polyhedra not restricted to rational data and since Theorem 1.5 is an important result which we will use in Chapter 2, it is one of the reasons for assuming rational data througout this thesis.

Solving methods

There exist solving methods which can solve LPs in polynomial time, e.g., the ellipsoid method [52, 53] and interior point methods [51, 81]. In practice, the simplex method [21] is often used, although it may have an exponential running time. In the context of MIP solving, however, it performs empirically better than the alternatives mentioned previously. In this thesis, we will therefore restrict ourselves to the simplex method for solving LPs.

Adding integrality constraints increases the complexity of the problem: MIP-solving is $\mathcal{NP}$-hard [86].

Two methods for solving MIPs are *LP based branch-and-bound* and the *general cutting plane method*. Both rely heavily on the *LP-relaxation* of the problem, i.e., the linear program obtained when disregarding the integrality restrictions of the MIP.

The branch-and-bound algorithm, a form of *divide-and-conquer*, divides the problem into subproblems until these can easily be solved to optimality. For each subproblem, the LP-relaxation is solved, providing a *lower bound* (also called *dual bound*) on the best feasible solution of the current subproblem. On the other hand, the best known primal solution to the global problem is referred to as the *incumbent*, its objective value is called *upper bound* or *primal bound*. If the lower bound of a subproblem is greater or equal to the upper bound, the current subproblem can be ignored since it cannot contain a solution with objective value better than that of the incumbent. Otherwise, the current problem is divided into multiple—typically two—subproblems. These two steps are called *bounding* and *branching*, respectively, which gives rise to the name of the algorithm. They are repeated until all subproblems are either solved to optimality or discarded due to their lower bound.

The branch-and-bound procedure can easily be illustrated as a tree, where each node represents one of the (sub-)problems. In particular, the root node of the tree represents the initial problem. Whenever a problem is divided into subproblems, these problems are represented by child nodes of the current problem's node in the tree. When a subproblem can be ignored due to the bounding process, the corresponding node is *pruned*.

On the other hand, in the general cutting plane method, the LP relaxation of the problem is also solved first. If the computed optimal LP solution is fractional, a *valid inequality* is added to the LP, which is satisfied by all feasible solutions of the MIP, but violated by the current fractional LP solution. Thus we can say that this inequality "cuts off" the fractional LP solution, it is therefore called *cutting plane*. Since geometrically, the cut can be seen as a hyperplane that separates the vector corresponding to the LP solution from all feasible solutions of the MIP, this process is also referred to as *separation*. This is repeated, until the optimal LP solution is integral and therefore an optimal solution of the MIP.

Most state-of-the-art MIP solvers use a combination of these two methods, called *branch-and-cut*. The LP relaxation of the subproblems is strengthened

by a reasonable number of cutting planes thereby trying to preserve its numerical stability. Often, only the LP relaxation of the root-node is strengthened by cutting planes, afterwards, the problem is solved by a branch-and-bound method.

Furthermore, *branch-and-price* [10] is a variant of branch-and-bound, where only a subset of the variables is given explicitly, most of them are handled implicitly. The key idea is that most of the variables will never be part of an optimal LP solution. They would just slow down the solution process and would consume too much memory. During the optimization process, at each node, the LP relaxation is solved with a *column generation* approach: Whenever one of the implicitly given variables might improve the current LP solution, it is added explicitly to the problem. We go into detail about column generation and branching in this context in Chapters 4 and 5, respectively.

Finally, *branch-cut-and-price* is a combination of branch-and-price and branch-and-cut. The variables are given implicitly and added to the problem only when needed. Additionally, cutting planes are added to the problem during the solving process in order to strenghen the LP relaxation. Variables and cutting planes are created alternately. Cutting plane generation in this context will be treated in Chapter 6.

1.2 A Brief History of Branch-and-Price

When talking about the history of branch-and-price, we have to start with column generation which is the foundation of branch-and-price.

Column generation is a method to solve linear problems with a huge number of variables. Instead of solving the complete linear program, we solve a restricted problem containing only a subset of the variables; remaining variables are treated implicitly. Therefor, a high number of smaller, typically well structured subproblems—called *pricing problems*—are solved that determine some of the implicitly given variables which are then explictly added to the restricted problem in order to improve its solution. Due to their structure, the pricing problems can typically be solved rather efficiently. We give a detailed description of column generation in Section 4.1.

Column generation has its roots in the 1960s when George B. Dantzig and Philip Wolfe proposed their decomposition principle for linear programs [22, 23]. The original problem is reformulated as a linear problem with (typically) an exponential number of variables. These variables represent the extreme points and rays of the polyhedra corresponding to the pricing problems which are linear subproblems.

Shortly after that, Paul C. Gilmore and Ralph E. Gomory presented a formulation of an integer program—the cutting stock problem—containing a huge number of variables that were implicitly treated [36, 37]. The pricing problem was a knapsack problem. However, only linear programming meth-

ods were used to solve the problem, so integrality of the variables could not be enforced: "That integers should result of the example is, of course, fortuitous" [36]. Nevertheless, similar reformulations were proposed for several other combinatorial optimization problems [66].

Combining the column generation approach with a branch-and-bound algorithm in order to solve these problems to integrality poses several challenges [8, 47] which we will discuss in Chapters 4 and 5. Hence, several years had to go by before column generation was successfully combined in practice with branch-and-bound to a method called *branch-and-price* or *IP column generation*.

In the 1990s, this concept was applied for example to the bin packing problem [90], the vertex coloring problem [63], and the generalized assignment problem [83]; problem independent surveys are given in [10, 97]. Besides, the column generation approach was based on a decomposition for integer programs [28, 91] that is similar to the Dantzig-Wolfe decomposition. This decomposition principle is presented in Chapter 2 of this thesis.

In the last decade, several general acceleration strategies for the branch-and-price process were proposed [26, 57] and branch-and-price was successfully applied to many problems [27]. Furthermore, branch-and-price was combined with the general cutting plane method to *branch-cut-and-price* [33, 87]. We will go into detail about this in Chapter 6.

Nowadays, branch-and-price is a well-established method to solve huge and extremely difficult combinatorial optimization problems. Its success relies on exploiting structures contained in the problem via a decomposition and being able to solve the occuring pricing problems rather efficiently with problem specific algorithms.

The most widely used method to solve MIPs, however, is the branch-and-cut algorithm employed by most state-of-the-art MIP solvers. Although the user can extend the existing branch-and-cut solvers by adding problem specific plugins, these solvers—commercial and non-commercial ones—feature very effective generic algorithms that can be used to solve many general MIPs without further problem knowledge.

The situation differs strongly for branch-and-price algorithms. There exist several branch-cut-and-price frameworks like `ABACUS` [48], `BCP` [80], `MINTO` [68], and `SCIP` [3]—which is the one we used and extended in this thesis. However, one cannot simply read in a problem and just solve it with a branch-and-price algorithm since the most important properties—like the pricing routine and a branching scheme—have to be implemented and provided by the user. Therefore, even when using the aforementioned possibilities, several problem specific parts of the solver have to be implemented before a problem can be solved.

It would be much more satisfactory to have generic implementations also for branch-and-price. These implementations shoul provide the essential functionalities to solve a general—or just a well structured—MIP with a branch-and-price approach. This could be used to easily test new ideas. Fur-

thermore, if it turns out that the branch-and-price approach performs well for specific problems, some of the generic functionalities could be replaced by problem specific ones to speed up the solving process.

In recent years, there has been some progress into this direction. François Vanderbeck has been developing important features [92, 94, 96, 95] for its own implementation called `BaPCod` which is "a prototype code that solves Mixed Integer Programs (MIP) by application of a Dantzig-Wolfe reformulation technique" [93].

Also the COIN-OR initiative [20] hosts a generic decomposition code, called `DIP` [76, 77] (formerly known as `DECOMP`), which is a "framework for implementing decomposition-based bounding algorithms for use in solving large-scale discrete optimization problems" [78].

The constraint programming `G12` project develops "user-controlled mappings from a high-level model to different solving methods" [75], one of which is branch-and-price.

Among these projects, only `DIP` is currently (March 2010) open to the public, but just as a trunk development version, there has not been a release, yet.

For all these implementations, the user has to specify the structure of a problem after reading it in and an automated Dantzig-Wolfe decomposition is performed that reformulates the problem. The reformulation is then solved with a branch-and-price algorithm; solving the pricing problems and branching are handled in a generic way.

In relation to this thesis, we developed our own implementation of a branch-cut-and-price solver called `GCG`. It extends the branch-cut-and-price *framework* `SCIP` [3], which already features one of the fastest non-commercial MIP solvers, to a branch-cut-and-price *solver*.

Chapter 2

Dantzig-Wolfe Decomposition for MIPs

The Dantzig-Wolfe decomposition for linear programs [22, 23] can be transferred to MIPs in two different ways: the *convexification approach* [28, 91] and the *discretization approach* [91, 47]. The former is the more general one, while the latter leads to better computational results when the problem has the appropriate structure so that it can be applied. The symbols related to the decomposition that are introduced in this chapter can also be looked up in Appendix B.

In this chapter, we present both concepts and and a comparison of them. Our presentation is based on [29, 57] for the convexification approach and [92, 96] for the discretization approach.

The first steps of the decomposition are the same in both approaches. Suppose we are given a MIP of the following form, which we will call the *original formulation*:

Model 2.1 (Original Program)

$$
\begin{aligned}
z^\star_{OP} = min \quad & \sum_{k\in[K]} {c_k}^T x^k & & \\
s.t. \quad & \sum_{k\in[K]} A^k x^k \geq b & & (2.1)\\
& D^k x^k \geq d^k & \forall k \in [K] \quad & (2.2)\\
& x^k \geq 0 & \forall k \in [K] \quad & (2.3)\\
& x^k_i \in \mathbb{Z} & \forall k \in [K], i \in [n^\star_k]. \quad & (2.4)
\end{aligned}
$$

The problem is modeled using $K \in \mathbb{N}$ vectors of variables $x^k \in \mathbb{Q}^{n_k}, k \in [K]$ and corresponding objective function vectors $c_k \in \mathbb{Q}^{n_k}$. We have two types of restrictions, *linking constraints* and *structural constraints*.

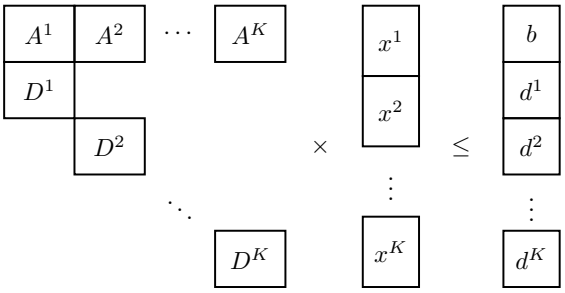


Figure 2.1: Structure of the constraints in the original problem

The *linking constraints* (2.1) are given by coefficient matrices $A^k \in \mathbb{Q}^{m_A \times n_k}$ and a right-hand side $b \in \mathbb{Q}^{m_A}$ and are "global" constraints that contain variables of different vectors x^k.

The *structural constraints* (2.2) can be divided into K blocks, where each block $k \in [K]$ imposes some restrictions on the variable vector x^k. These constraints are given by matrices $D^k \in \mathbb{Q}^{m_k \times n_k}$ and right-hand sides $d^k \in \mathbb{Q}^{m_k}$.

In addition to that, all variables are non-negative and for each variable vector x^k, the first $n_k^\star$ variables are of integral type.

The structure of the contraint maxtrix of the original formulation is illustrated in Figure 2.1. This structure will be called *bordered block diagonal* in the following.

Typically, when using the Dantzig-Wolfe reformulation, we have multiple blocks, i.e., $k > 1$. However, if we have just one block but the structural constraints define a combinatorial optimization problem that can be solved much more efficiently than the global problem, the Dantzig-Wolfe reformulation can be used to exploit this structure, too. In this case, the linking constraints do not link the values of the variables of different blocks but are complicating constraints that destroy the structure of the subproblem when treating all constraints together.

We will now introduce a set X_k for each block $k \in [K]$ that contains the vectors satisfying the *structural constraints* as well as the non-negativity and integrality constraints associated with this block:

$$X_k := \{x^k \in \mathbb{Z}_+^{n_k^\star} \times \mathbb{Q}_+^{n_k - n_k^\star} \mid D^k x^k \geq d^k\}. \tag{2.5}$$

Restricting the values of x^k to this set X_k for all $k \in [K]$, we treat constraints (2.2), (2.3) and (2.4) implicitly and can therefore omit them in the *compact formulation*:

Model 2.2 (Compact Program)

$$z^\star_{CP} = min \quad \sum_{k\in[K]} {c_k}^T x^k$$

$$s.t. \quad \sum_{k\in[K]} A^k x^k \geq b \tag{2.6}$$

$$x^k \in X_k \qquad \forall k \in [K]. \tag{2.7}$$

At this point, the further steps differ for the convexification approach and the discretization approach. We start with a detailed description of the convexification approach. After that, we present the discretization approach and give a comparison between both approaches.

2.1 The Convexification Approach

In the convexification approach, we will now represent each vector $x^k \in X_k$ by a convex combination of extreme points plus a conical combination of extreme rays, like it is also done in the Dantzig-Wolfe decomposition for linear programs (see [22, 23]).

In contrast to the classical Dantzig-Wolfe decomposition, the set X_k is not a polyhedron since we have integrality conditions on some of the variables. It is, however, the set of solutions to a MIP defined by rational data, so its convex hull is a polyhedron (see Section 1.1). From now on, we thus investigate the convex hull of X_k in order to get a polyhedron that we can describe by a finite number of extreme points and extreme rays:

Theorem 2.3 (Minkowski and Weyl Theorems)
A set $X \subseteq \mathbb{Q}^n$ is a polyhedron if and only if there exist finite sets $P = \{p_1, p_2, \ldots, p_m\} \subseteq \mathbb{Q}^n$ and $R = \{r_1, r_2, \ldots, r_\ell\} \subseteq \mathbb{Q}^n$ such that $X = conv(P) + cone(R)$.

So, for each set X_k there exist finite sets $P_k \subseteq \mathbb{Z}_+^{n_k^\star} \times \mathbb{Q}_+^{n_k - n_k^\star}$ of extreme points of $conv(X_k)$ and $R_k \subseteq \mathbb{Z}_+^{n_k^\star} \times \mathbb{Q}_+^{n_k - n_k^\star}$ of extreme rays of $conv(X_k)$ so that each $x^k \in X_k \subseteq conv(X_k)$ can be represented as a convex combination of the extreme points plus a non-negative combination of the extreme rays, i.e., for each $x^k \in X_k$ there exists $\lambda \in \mathbb{Q}_+^{|P|+|R|}$ with

$$x^k = \sum_{p\in P} \lambda_p \cdot p + \sum_{r\in R} \lambda_r \cdot r; \quad \sum_{p\in P} \lambda_p = 1. \tag{2.8}$$

With this result, we can substitute x^k in the compact formulation (2.2) according to (2.8) and get the following *extended fomulation*.

For ease of presentation, we define

$$c_q^k := {c_k}^T q \text{ and } a_q^k := A^k q \text{ for } q \in P_k \cup R_k, \text{ for all } k \in [K]. \tag{2.9}$$

Model 2.4 (Extended Formulation for Convexification)

$$
\begin{aligned}
z^{\star}_{EPC} = min \quad & \sum_{k\in[K]}\sum_{p\in P_k} c_p\lambda^k_p + \sum_{k\in[K]}\sum_{r\in R_k} c_r\lambda^k_r && && \\
s.t. \quad & \sum_{k\in[K]}\sum_{p\in P_k} a_p\lambda^k_p + \sum_{k\in[K]}\sum_{r\in R_k} a_r\lambda^k_r \geq b && && (2.10)\\
& \sum_{p\in P_k} \lambda^k_p = 1 && \forall k \in [K] && (2.11)\\
& \lambda^k \geq 0 && \forall k \in [K] && (2.12)\\
& \sum_{p\in P_k} p\lambda^k_p + \sum_{r\in R_k} r\lambda^k_r = x^k && \forall k \in [K] && (2.13)\\
& x^k_i \in \mathbb{Z}. && \forall k \in [K], && \\
& && i \in [n^{\star}_k]. && (2.14)
\end{aligned}
$$

This problem is a reformulation of the compact problem and thus equivalent to the original problem. In addition to the K variable vectors x^k that represent the solution vectors of the compact problem (Model 2.2), we get for each block $k \in [K]$ one variable λ^k_p for each extreme point $p \in P_k$ as well as one variable λ^k_r for each extreme ray $r \in R_k$.

For each $k \in [K]$, the coupling constraints (2.13) link the variable x^k to the values given by the selected combination of extreme points and extreme rays of $conv(X_k)$, which is a convex one for the former and a conical one for the latter because of constraints (2.11) and (2.12). Using Theorem 2.3 (Minkowsky and Weyl), it follows $x^k \in conv(X_k)$ and, due to the integrality constraints (2.14), even $x^k \in X_k$. Therefore, for each $k \in [K]$, x^k satisfies the corresponding constraint of type (2.7) in the compact problem.

The objective function and constraints (2.10) are equivalent to their counterparts in the compact formulation: x^k was substituted according to (2.8) and definition (2.9) was used.

When we want to solve the LP-relaxation of Model 2.4, we relax the integrality of the variables x^k and remove constraints (2.14). As the vectors x^k are only used to enforce integrality, after removing the integrality constraints (2.14), we can also remove the coupling constraints (2.13) as well as the variable vectors x^k, $k \in [K]$, from the relaxation and get the following linear program as a relaxation of Model 2.4.

Model 2.5 (Master Problem for Convexification)

$$z^{\star}_{MPC} = min \quad \sum_{k\in[K]}\sum_{p\in P_k} c_p\lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} c_r\lambda_r^k$$

$$s.t. \quad \sum_{k\in[K]}\sum_{p\in P_k} a_p\lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} a_r\lambda_r^k \geq b \tag{2.15}$$

$$\sum_{p\in P_k} \lambda_p^k = 1 \qquad \forall k \in [K] \tag{2.16}$$

$$\lambda^k \geq 0 \qquad \forall k \in [K] \tag{2.17}$$

This *master problem (MP)* is a relaxation of the extended formulation and therefore, its optimal solution value is also a lower bound for the optimal solution value of the original problem. Each optimal solution to the master problem (Model 2.5) can be transformed into a (possibly fractional) solution candidate of the original problem. As we will in Section 2.3, the master problem typically gives rise to a better lower bound than the LP-relaxation of the original problem, but it also has a big handicap: It has, in general, exponential many variables and these variables are given implicitly. Computing even just one of the variables is in general $\mathcal{NP}$-hard as they correspond to solutions of an arbitrary MIP. Hence, computing all extreme points and extreme rays in advance is far to costly and even if we would do so, an LP containing all the variables explicitly would consume a vast amount of memory and would be computationally hard to solve. In Chapter 4, we will describe how to solve this master problem in a more efficient way by using column generation.

2.2 The Discretization Approach

In many cases when a problem can be decomposed in the described way, all the blocks or a part of them are identical, i.e., the polyhedra $conv(X_k)$ are identical and all contained points have the same objective function value. In the bin packing problem (see Section C.1), for example, all bins have the same size, so all the blocks are identical.

In the convexification approach however, in order to force the integrality of the points chosen in each polyhedron, we have to distinguish between the extreme points related to different blocks even if the polyhedra of these blocks are identical. Therefore, we need to create multiple variables for the same extreme point, each of which is related to a different block.

In case of pure integer programs, we can use the discretization approach to avoid this.

Starting with the original program (Model 2.1), we transform it to the compact program (Model 2.2) like described previously. Instead of using the Minkowski and Weyl theorems to get a representation of all points $x \in X_k$

and introducing variables for the extreme points and extreme rays, we simply create integral variables λ_p^k and λ_r^k for a number of points $p \in X_k$ and rays $r \in \mathbb{Z}^n$. Again, we only allow a convex combination of points what is equal to choosing exactly one of them this time.

The following therorem, a slightly modified version of Theorem 6.1 in [69], shows that it is possible to get such a representation with finitely many points and rays for each set X_k defined like in (2.5).

Theorem 2.6
If $\mathcal{P} = \{x \in \mathbb{Q}_+^n \mid Ax \leq b\} \neq \emptyset$ and $S = \mathcal{P} \cap \mathbb{Z}^n$, where (A, b) is a rational $m \times (n+1)$ matrix, then there exist a finite set of points $\{q^\ell\}_{\ell \in L}$ of S and a finite set of rays $\{r^j\}_{j \in J}$ of $\mathcal{P}$ such that

$$S = \left\{ x \in \mathbb{Q}_+^n \mid x = \sum_{\ell \in L} \alpha_\ell q^\ell + \sum_{j \in J} \beta_j r^j, \sum_{\ell in L} \alpha_\ell = 1, \alpha \in \mathbb{Z}_+^{|L|}, \beta \in \mathbb{Z}_+^{|J|} \right\}.$$

For the proof of this theorem, we refer to [69]. An integer matrix is required there, but we can easily scale the rational matrix (A, b) to integer values without changing the problem.

Corollary 2.7
For each set X_k defined like in (2.5), there exist finite sets $P_k \subseteq X_k$ and $R_k \subseteq \mathbb{Z}^{n_k}$ such that the following is equivalent:

1. *$x \in X_k$*

2. *There exists $\lambda \in \mathbb{Z}_+^{|P_k|+|R_k|}$ with $\sum_{p \in P_k} \lambda_p = 1$ such that*

$$x = \sum_{p \in P_k} \lambda_p p + \sum_{r \in R_k} \lambda_r r. \tag{2.18}$$

This looks similar to the Minkowski and Weyl theorems 2.3, except that the factors λ are integral. This is compensated by a higher number of points in the set P_k while the number of rays stays the same.

Using this representation of all points in X_k, we can substitute x^k in the compact formulation (model 2.2) and get an *extended formulation* for the discretization approach using the abbreviations (2.9).

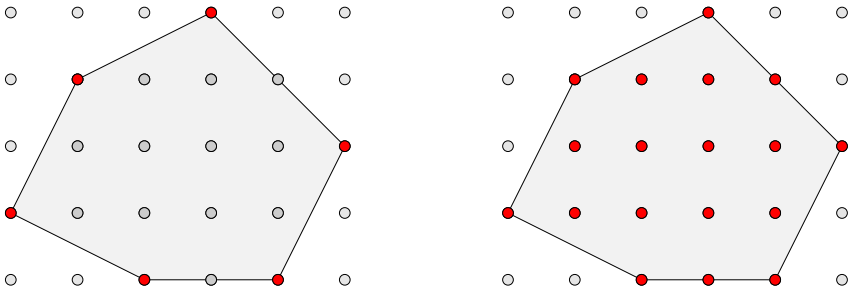


Figure 2.2: The points represented by variables in the convexification (left) and the discretization approach (right) for a bounded polyhedron $conv(X_k)$

Model 2.8 (Extended Formulation for Discretization)

$$z^{\star}_{EPD} = min \quad \sum_{k\in[K]}\sum_{p\in P_k} c_p^k \lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} c_r^k \lambda_r^k$$

$$s.t. \quad \sum_{k\in[K]}\sum_{p\in P_k} a_p^k \lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} a_r^k \lambda_r^k \geq b \tag{2.19}$$

$$\sum_{p\in P_k} \lambda_p^k = 1 \qquad \forall k \in [K] \tag{2.20}$$

$$\lambda^k \in \mathbb{Z}_+^{|P_k|+|R_k|} \qquad \forall k \in [K] \tag{2.21}$$

Model 2.8 is again a reformulation of the compact problem and thus also of the original problem. As mentioned previously, x^k is replaced according to (2.18) and (2.9) in objective function and linking constraints. Due to Theorem 2.6, constraints (2.20) and (2.21) are equivalent to constraints (2.7) in the compact formulation.

Hence, we can impose integrality constraints directly on the variables related to the points and rays in order to enforce integrality of the corresponding solution in the original program and do not need the vectors x^k anymore.

Apart from the missing variable vectors x^k and the integrality of the λ variables, this looks similar to the extended formulation of the convexification approach. The only further difference is given implicitly in the definition of the sets P_k and R_k.

The relation between the sets of variables in the convexification and the discretization approach are pictured in Figure 2.2 and Figure 2.3. The polyhedron $conv(X_k)$ is colored grey while the points and rays contained in the sets P_k and R_k, respectively, are colored red. The chosen points in the discretization approach for an unbounded polyhedron are the smallest set of the form given in Corrolary 2.7, one could also choose a bigger set.

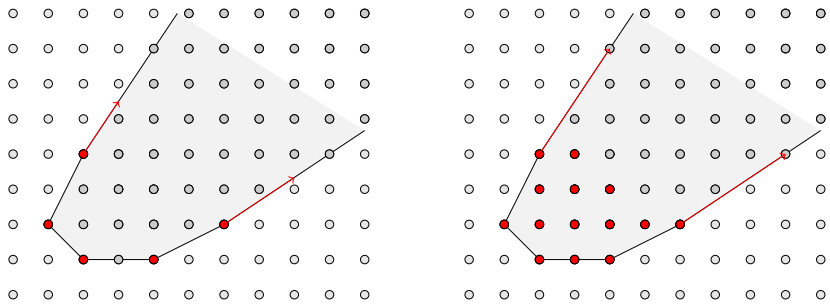


Figure 2.3: The points and rays represented by variables in the convexification (left) and the discretization approach (right) for an unbounded polyhedron $conv(X_k)$

Furthermore, when relaxing the integrality restrictions, we get a master problem, that is also equivalent to the master problem in the convexification approach.

Model 2.9 (Master Problem for Discretization)

$$z^{\star}_{MPD} = min \quad \sum_{k\in[K]}\sum_{p\in P_k} c_p^k\lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} c_r^k\lambda_r^k$$

$$s.t. \quad \sum_{k\in[K]}\sum_{p\in P_k} a_p^k\lambda_p^k + \sum_{k\in[K]}\sum_{r\in R_k} a_r^k\lambda_r^k \geq b \tag{2.22}$$

$$\sum_{p\in P_k} \lambda_p^k = 1 \qquad \forall k \in [K] \tag{2.23}$$

$$\lambda^k \geq 0 \qquad \forall k \in [K] \tag{2.24}$$

The constraints (2.23) and (2.24) assure that a convex combination of points and a conical combination of rays is chosen. The rays R_k have the same directions as in the convexification approach, they are only scaled to integral values. The set of points P_k contains all the extreme points that are contained in the convexification approach, but even some more points. These additional points, however, are all interior points, so they do not change the polyhedron described by $conv(P_k) + cone(R_k)$. Thus, this polyhedron is the same for the two different definitions of the sets P_k and R_k in the convexification and the discretization approach: it is the set $conv(X_k)$, the convex hull of the set X_k.

Therefore, in both master formulations, we optimize over the same polyhedron, namely the intersection of the polyhedron defined by the linking constraints and the polyhedron $conv(X_k)$ and both problems have the same objective function, so these problems are equivalent.

The integrality restrictions on the λ-variables in the extended formulation 2.8 allow standard primal heuristics to find feasible solutions and also

cutting planes could be derived from this formulation. The biggest advantage, however, is that we do not need the relation between blocks and variables in order to enforce integrality, so we can treat identical blocks jointly and need to create the same point only once for all these blocks.

Suppose $[K]$ can be partitioned into L sets $K_\ell, \ell \in [L]$ of identical blocks, called *identity classes* in the following. For ease of presentation, we assume that the first L blocks are pairwise different and for $1 \leq \ell \leq L$, block ℓ represents the set K_ℓ, i.e.,

$$K_\ell = \{k \in [K] \mid c_k = c_\ell, A^k = A^\ell, D^k = D^\ell, d^k = d^\ell\}.$$

Since all blocks $k \in K_\ell$ are identical, they have the same set of solutions $X_k = X_\ell$ and thus also the same points $P_k = P_\ell$ and rays $R_k = R_\ell$ in the representation.

Moreover, in the extended formulation 2.8, there is no distinction between two variables of different, identical blocks representing the same point (or ray), they have the same objective function coefficient and the same coefficients in the linking constraints (2.19). Therefore, we can aggregate the variables representing the same point or the same ray and get new variables

$$\tilde{\lambda}^\ell_p = \sum_{k \in K_\ell} \lambda^k_p \tag{2.25}$$

and

$$\tilde{\lambda}^\ell_r = \sum_{k \in K_\ell} \lambda^k_r. \tag{2.26}$$

This leads to the following *aggregated extended formulation*, again using the abbreviations (2.9).

Model 2.10 (Aggregated Extended Formulation for Discretization)

$$\begin{aligned}
z^\star_{EPDa} = min \quad & \sum_{\ell \in [L]} \sum_{p \in P_\ell} c^l_p \tilde{\lambda}^\ell_p + \sum_{\ell \in [L]} \sum_{r \in R_\ell} c^l_r \tilde{\lambda}^\ell_r & \\
s.t. \quad & \sum_{\ell \in [L]} \sum_{p \in P_\ell} a^l_p \tilde{\lambda}^\ell_p + \sum_{\ell \in [L]} \sum_{r \in R_\ell} a^l_r \tilde{\lambda}^\ell_r \geq b & (2.27)\\
& \sum_{p \in P_\ell} \tilde{\lambda}^\ell_p = |K_\ell| \qquad \forall \ell \in [L] & (2.28)\\
& \tilde{\lambda}^\ell \in \mathbb{Z}_+^{|P_\ell|+|R_\ell|} \qquad \forall \ell \in [L] & (2.29)
\end{aligned}$$

In the objective function and in the linking constraints (2.19), we introduced the variables $\tilde{\lambda}^\ell_p$ and $\tilde{\lambda}^\ell_r$ according to their definitions (2.25) and (2.26) and we now get one summand for each class $\ell \in [L]$ of identical blocks instead of one for each block $k \in [K]$.

The convexity constraints (2.20) of identical blocks are added up, we substitute $\tilde{\lambda}_p^\ell = \sum_{k \in K_\ell} \lambda_p^k$, and the right-hand side changes to the cardinality of the class of identical blocks.

Finally, the integrality constraints (2.21) of identical blocks are combined to constraints (2.29) forcing the aggregated variables $\tilde{\lambda}^\ell$ to be integral.

Altogether, this leads to the following lemma:

Lemma 2.11 (Equivalence of the Extended Formulations)
The aggregated extended problems (Model 2.10) is equivalent to the extended problem for the discretization apporach (Model 2.8): Each solution to Model 2.8 corresponds to a solution to Model 2.10 with the same objective function value and vice versa.

Proof 2.12
Given a solution to Model 2.8, by applying the aggregation prescriptions (2.25) and (2.26) we get values of the aggregated variables that form a feasible solution for Model 2.10 with the same objective value.

On the other hand, if we have a solution $\tilde{\lambda}$ to Model 2.10, for each class of identical blocks, all variables have non-negative integer values and the values of variables that belong to points sum up to $|K_\ell|$. We can distribute these values among the set K_ℓ of identical blocks of this class to get a feasible solution of Model 2.8 by doing the following for each class $\ell \in [L]$ of identical blocks. Let $\tilde{\lambda}_{p_1}^\ell, \dots, \tilde{\lambda}_{p_s}^\ell$ be the variables corresponding to points that have strictly positve value in the given solution and $K_\ell = \{k_1, \dots, k_{|K_\ell|}\}$. We set

$$\begin{aligned} \lambda_r^{k_1} &= \tilde{\lambda}_r^\ell && \forall r \in R_\ell \\ \lambda_r^k &= 0 && \forall k \in K_\ell \setminus \{k_1\}, r \in R_\ell. \end{aligned}$$

For $i = 1, \dots, s$ and $j = \sum_{t=1}^{i-1} \tilde{\lambda}_{p_t}^\ell + 1, \dots, \sum_{t=1}^{i} \tilde{\lambda}_{p_t}^\ell$, we set:

$$\begin{aligned} \lambda_{p_i}^{k_j} &= 1 \\ \lambda_p^{k_j} &= 0 && \forall p \in P_\ell \setminus \{p_i\}. \end{aligned}$$

That is, we distribute the solution values among the set K_ℓ of identical blocks of this class, such that in each block, exactly one variable corresponding to a point gets value 1, all other variables receive value 0.

Hence, the convexity constraints of Model 2.8 are satisfied, as well as constraints (2.19), since the value of each variable in the given solution was distributed among variables that have the same coefficients in (2.19) as the given variable in (2.27). Finally, the values were also distributed among variables with the same objective function coefficient, thus, the objective function value stays the same. □

The disaggregation of a solution to the aggregated extended problem presented in Proof 2.12 is not unique. By permuting the set of blocks corresponding to an identity class, we get a different solution in the non-aggregated extended problem and in the original problem. This is due to the aforementioned symmetry: In the original and in the non-aggregated extended problem, we get, for each feasible solution, equivalent solutions by permuting identical blocks. All these solutions correspond to a single solution in the aggregated extended problem, since we do not distinguish the identical blocks in this model.

Corollary 2.13
The following problems are equivalent:

- *the original problem (Model 2.1)*
- *the compact problem (Model 2.2)*
- *the extended problem for the convexification approach (Model 2.4)*
- *the extended problem for the discretization approach (Model 2.8)*
- *the aggregated extended problem for the discretization approach (Model 2.10)*

When relaxing the integrality of the $\tilde{\lambda}^\ell$ variables in order to get the LP relaxation, we get a *master problem* quite similar to the master problem in the convexification approach (Model 2.5).

Model 2.14 (Aggregated Master Problem for Discretization)

$$
\begin{aligned}
z^\star_{MPDa} = min \quad & \sum_{\ell\in[L]}\sum_{p\in P_\ell} c_p^l \tilde{\lambda}_p^\ell + \sum_{\ell\in[L]}\sum_{r\in R_\ell} c_r^l \tilde{\lambda}_r^\ell \\
s.t. \quad & \sum_{\ell\in[L]}\sum_{p\in P_\ell} a_p^l \tilde{\lambda}_p^\ell + \sum_{\ell\in[L]}\sum_{r\in R_\ell} a_r^l \tilde{\lambda}_r^\ell \geq b \\
& \sum_{p\in P_\ell} \tilde{\lambda}_p^\ell = |K_\ell| \qquad \forall \ell \in [L] \\
& \tilde{\lambda}^\ell \geq 0 \qquad \forall \ell \in [L]
\end{aligned}
$$

Like the extended formulations, the master problems of the two different approaches are equivalent, as the following lemma states.

Lemma 2.15 (Equivalence of the Master Problems)
The aggregated master problem for the discretization approach (Model 2.14) is equivalent to the master problem for the convexification approach (Model 2.5): Each solution to one of the problems corresponds to a solution of the other problem with the same objective function value and vice versa.

Proof 2.16
The convexification master problem (Model 2.5) is equivalent to the non-aggregated master problem of the discretization approach (Model 2.9), so it is sufficient to show that the aggregated master problem for the discretization approach (Model 2.14) is equivalent to Model 2.9.

Given a solution to Model 2.9, by applying the aggregation prescriptions (2.25) and (2.26), we get values of the aggregated variables that form a feasible solution for Model 2.14 with the same objective value.

On the other hand, if we have a solution $\tilde{\lambda}$ to Model 2.14, for each class of identical blocks, all variables have non-negative values and the values of variables that belong to points sum up to $|K_\ell|$. We construct a solution λ to Model 2.9 in the following way: For each class of identical blocks $\ell \in [L]$ we set

$$\lambda_q^k = \frac{\tilde{\lambda}_q^\ell}{|K_\ell|} \; \forall k \in K_\ell, q \in P_\ell \cup R_\ell.$$

The values are distributed among variables with the same coefficients in objective function and linking constraints, so the objective function value stays the same and constraints (2.22) are still satisfied. The non-negativity constraints (2.24) are still satisfied, too, and by dividing the values by $|K_\ell|$, the sum of values of variables corresponding to points is 1 for each block, so constraints (2.23) are also satisfied. Hence, this solution is feasible for Model 2.9 and has the same objective value as the given one. □

Thus, the aggregated master problem for discretization (Model 2.14) has the same optimal objective value as the convexification master problem (Model 2.5) and the discretization master problem (Model 2.9), i.e., $z^\star_{MPC} = z^\star_{MPD} = z^\star_{MPDa}$. In the following, when speaking of one of these master problems, we will simply denote its optimal objective value by $z^\star_{MP}$.

Remark 2.17
When transferring a solution of the aggregated master problem (Model 2.14) into a solution to the original problem, we also need to distribute the values of variables corresponding to a class of identical blocks to the single blocks of this class. This could be done in the way as described in Proof 2.16, but in practice, it is much more efficient to preserve integrality of the variables. As far as possible, in each block, exactly one point should be chosen, only for the last blocks, when all remaining values are fractional, a convex combination of multiple points has to be chosen.

This way, an integral solution of the aggregated master problem results in an integral solution of the original problem after the transformation. For a more sophisticated scheme to transfer a solution from the aggregated extended problem to the original problem which makes use of a lexicographical order we refer to [95].

Like the convexification master problem, the both master problems for the discretization approach have in general an exponential number of variables.

We will give a detailed description of how to solve them with a column generation approach in Chapter 4.

Let us regard again the variables corresponding to integral points. Like mentioned previously, we do not need them to solve the master problem and as we will see later, we will only create variables corresponding to extreme points in the column generation procedure. Nevertheless, in order to find optimal or even feasible solutions to one of the extended formulations for the discretization approach (Model 2.8 or Model 2.10), interior points are essential since they may be contained in an optimal solution or the linking constraints may even forbid solutions containing only extreme points.

It is important to notice, that for a pure BP, all feasible integral points are extreme points, so in this case, we do not need to bother about creating interior points in the column generation procedure.

For an IP, we have two different possibilities: On the one hand, if the IP is bounded, it can be converted into a BP by replacing each integral variable by some binary variables. On the other hand, we can use branching rules that modify the subproblems, the set of solutions $X_k, k \in [K]$ and thus also the polyhedra $conv(X_k)$, so that each initially interior point will become an extreme point at some level in the branching tree. In Chapter 5, we will discuss different branching rules with that property.

The discretization approach can also be applied to MIPs but this is more complicated. We will only give a short overview at this point, for which we assume that the MIP is bounded. For a deeper discussion, we refer to [96]. Since the MIP is bounded, the projection of $conv(X_k)$ to the integral variables leads to a set P_k of integral points and for each point of this set, we get a polytope in the continuous variables. Like in the discretization approach for IPs, we choose exactly one point of the set P_k and for the continuous variables, we choose a convex combination of extreme points of the polytope corresponding to that integral point. The number of variables is still finite since we have a finite number of integral points and for each of these points a finite number of extreme points of the corresponding polytope in the continuous variables.

2.3 Quality of the Relaxation

If we would solve the original problem (Model 2.1) with a standard branch-and-bound algorithm, we would get a lower bound $z^\star_{LP}$ on the optimal objective value $z^\star_{OP}$ by solving its LP-relaxation.

The LP relaxation of the extended formulation is the master problem, Model 2.5 for the convexification, Model 2.9 or 2.14 for the discretization approach. All these three models give rise to the same lower bound $z^\star_{MP}$ which is in general a better bound than the bound $z^\star_{LP}$ obtained by solving the LP relaxation of the original problem:

Theorem 2.18
For the optimal objective value $z^\star_{OP}$ to the original problem and the solutions $z^\star_{LP}$ and $z^\star_{MP}$ of its LP relaxation and the master problem, respectively, the following holds:

$$z^\star_{LP} \leq z^\star_{MP} \leq z^\star_{OP}. \tag{2.30}$$

Proof 2.19
In the LP relaxation, we minimize over the intersection of the two polyhedra

$$P_A = \left\{ (x^{1T}, \ldots, x^{KT})^T \in \mathbb{Q}^n_+ \mid \sum_{k\in[K]} A^k x^k \geq b \right\}$$

and

$$P^{LP}_D = \left\{ (x^{1T}, \ldots, x^{KT})^T \in \mathbb{Q}^n_+ \mid D^k x^k \geq d^k \; \forall k \in [K] \right\}.$$

The set of feasible solutions to the master problem is the intersection of P_A with the polyhedron

$$P^{MP}_D = conv(X_1) \times \cdots \times conv(X_K)$$

that further restricts the polyhedron P^{LP}_D since it is the inclusion minimal polyhedron containing all points that satisfy both $D^k x^k \geq d^k$ as well as the integrality restictions for each block.

Therefore, $P^{MP}_D \subseteq P^{LP}_D$ and so $P_A \cap P^{MP}_D \subseteq P_A \cap P^{LP}_D$. It follows

$$z^\star_{LP} = \min_{x\in P_A \cap P^{LP}_D} \left\{ (c_1^T, \ldots, c_K^T)x \right\} \leq \min_{x\in P_A \cap P^M_D} \left\{ (c_1^T, \ldots, c_K^T)x \right\} = z^\star_{MP}$$

and since the master problem is a relaxation of the extended formulation which is equivalent to the original problem, it follows (2.30). □

It can be shown (see [35]) that the master problem is the dual formulation of the Lagrangean dual obtained by dualizing the linking constraints (2.1), so the master problem provides the same lower bound as the Lagrangean relaxation.

The bound obtained by the master problem is typically strictly better than the LP bound, using it as a relaxation helps in closing part of the integrality gap. However, when the pricing subproblem possesses the *integrality property*, i.e., each basic solution to the pricing problem is integral even if it is solved as an LP, P^{MP}_D equals P^{LP}_D, so the LP bound equals the bound obtained by the master problem [57]. Examples are the shortest path problem or a minimum cost flow problem with integral data.

We end this chapter with a small example that demonstrates the different polyhedra and the better lower bound of the master problem.

Example 2.20
Let the following original problem be given:

Model 2.21

$$\begin{aligned} min \quad & -x - 4y & \\ s.t. \quad & 5x + 4y \leq 20 & (2.31)\\ & x + 6y \leq 21 & (2.32)\\ & 3x + y \leq 10 & (2.33)\\ & x, y \geq 0 & \\ & x, y \in \mathbb{Z}. & \end{aligned}$$

We treat constraint (2.31) *as a* linking constraint*, i.e., it will be part of the master problem, and regard constraints* (2.32) *and* (2.33) *as* structural constraints *of the single block; they are used to define the set* X_1*.*

Figure 2.4 illustrates the problem; the black line represents the linking constraint (2.31)*, so the polyhedron* P_A *contains all non-negative points that are to the left of this line.*

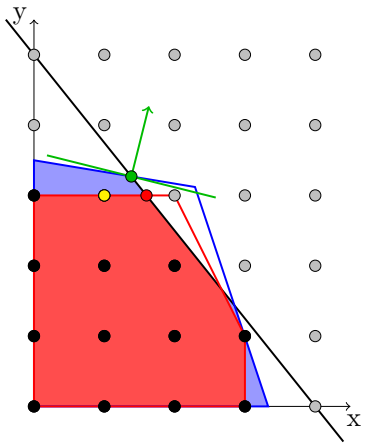


Figure 2.4: The polyhedra considered by the LP relaxation and the master problem

The blue lines enclose the polyhedron P_D^{LP} *described by the structural constraints* (2.32) *and* (2.33) *and the non-negativity constraints. The convex hull of integral points in* P_D^{LP} *is surrounded by the red lines and equals the polyhedron* P_D^{MP}*.*

When solving the LP relaxation of the problem, we optimize over the intersection of P_A *and* P_D^{LP}*, which is the colored area (blue and red). Solving the master problem corresponds to optimizing over the intersection of* P_A *and* P_D^{MP}*, this polyhedron is painted red and is a subset of the former polyhedron.*

For the given objective function, the integer point that is colored yellow is the optimum of Model 2.21. The optimal solution to the LP relaxation is illustrated by the green point while the master problem's optimal solution is represented by the red point. As one can easily see, the optimal objective value of the LP relaxation is better than the optimal objective value of the master problem, so the master problem gives rise to the tighter lower bound.

Chapter 3

Extending SCIP to a Generic Branch-Cut-and-Price Solver

In this chapter, we describe a way to integrate the Dantzig-Wolfe decomposition into a MIP solver. First, we briefly explain the algorithmic concept of the branch-cut-and-price *framework* SCIP, which is the basis of our implementation.

After that, we present the generic branch-cut-and-price *solver* GCG and describe its general structure and its solution process. Details about the solving process of the master problem, specific branching rules and specialized cutting plane separators are given in Chapters 4, 5, and 6, respectively.

At the end of this chapter, we state general information about the computational studies that we performed in order to evaluate the performance of our implementation.

3.1 SCIP—a MIP Solver

SCIP [2] is a framework created to solve *Constraint Integer Programs*, shortly called *CIPs* . Constraint Integer Programming is an integration of Constraint Programming (see for example [9]) and Mixed Integer Programming. For an exact definition and discussion of CIPs we refer to [1, 4]. SCIP was developed by Achterberg et al. [3] and is implemented in C.

SCIP is conceived as a framework that provides the infrastructure to implement branch-and-bound based search algorithms. The majority of the algorithms that are needed to control the search, e.g., branching rules, must be included as external plugins. The plugins are user defined callback objects, which interact with the framework through a very detailed interface provided by SCIP. This is, they define a set of methods that are registered in the framework and called by SCIP during the solving process.

In the following, we give a short overview of the most important plugins. A detailed discussion of all types of plugins supported by SCIP and the underlying concepts can be found in [1, Chapter 3]. Furthermore, Figure 3.1

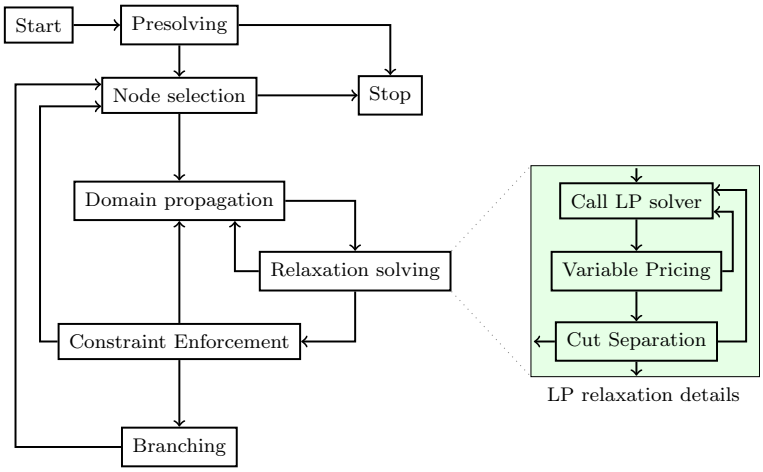


Figure 3.1: Solving process of SCIP with detailed LP relaxation

illustrates how the different plugins cooperate in the main solving loop of SCIP. For clarity reasons, we missed out primal heuristics in the figure. They can be called after each step of the solving loop.

The central objects of SCIP are the *constraint handlers*. Each of them represents one class of constraints, e.g., linear constraints, and provides methods to handle constraints of its type. Their primary task is to check solutions for feasibility with respect to their constraint class and to enforce that solutions satisfy all constraints of their class. Furthermore, they can provide additional constraint specific methods for presolving, separation, domain propagation, and branching.

For the branch-and-bound process, *branching rules* and *node selectors* are needed. The former split the current problem into subproblems and create new nodes in the branch-and-bound tree, while the latter select the next node to be processed when the solving process of a node is finished.

Domain propagation, also called *node preprocessing*, is performed at every node of the branch-and-bound tree. Its goal is to tighten the local domains of variables for the current subproblem.

LP based branch-and-cut algorithms try to solve the current subproblem via its LP relaxations. If its solution satisfies the integrality restrictions, the subproblem is solved, otherwise, the optimal LP objective value serves as a dual bound for the subproblem. This is also the default strategy in SCIP. However, other relaxations of the problem can be used, too. In SCIP, this functionality is provided by *relaxation handler* plugins, which can compute dual bounds and primal solution candidates. They can be used in addition to the LP relaxation or even replace it.

Furthermore, *primal heuristics* try to find feasible solutions and *cutting plane separators* can produce valid inequalities that cut off the current LP solution or a given arbitrary solution.

Finally, *variable pricers* can add variables to the problem during the solving process. Variables can be treated implicitly and added to the problem only when needed. This concept is called *column generation*—see Chapter 4 for a detailed discussion—and allows to implement branch-and-price algorithms in SCIP.

The current distribution of SCIP already contains a bundle of plugins that can be used for MIP solving. The most important ones are described in [1, Chapter 5 – 10]. SCIP with default plugins is a state-of-the-art MIP solver which is competitive (see Mittelmann's "Benchmarks for Optimization Software" [67]) with other free solvers like CBC [31], GLPK [38], and Symphony [79] and also with commercial solvers like Cplex [43] and Gurobi [42]. Therefore, in order to compare the performance of our branch-cut-and-price solver GCG to a branch-and-cut MIP solver, we will use SCIP with default plugins as a MIP solver of this kind.

3.2 GCG—a Generic Branch-Cut-and-Price Solver

GCG (an acronym for "Generic Column Generation") is an add-on for SCIP that takes into account the structure of a MIP and solves it with a branch-cut-and-price approach after performing a Dantzig-Wolfe decomposion. It was developed by the author of this thesis. GCG supports both the convexification approach (see Section 2.1) as well as the discretization approach (Section 2.2). The description of the solving process and the structure of GCG in the remainder of this section is valid for both these approaches. Differences that occur for the pricing, branching, and separation process are discussed in the following three chapters. The most important parameters of GCG described in the following are summarized in Appendix B.

Even though the specifications of SCIP did not always allow a straightforward implementation and sometimes lead to some overhead, by embedding GCG into SCIP, we benefit from the efficient branch-and-bound framework with all its sophisticated functionalities.

The general structure of GCG is the following: Besides the main SCIP instance, that reads in the original problem, we use a second SCIP instance that represents the extended problem. In the following, we will call these instances the *original* and the *extended (SCIP) instance*, respectively. In addition to these two SCIP instances, all pricing problems are represented by their own SCIP instance, too.

The original SCIP instance is the primary one which coordinates the solving process, the extended SCIP instance is controlled by a relaxation handler that is included into the original SCIP instance. This relaxation handler is called *DW relaxator* in the following. The relaxator provides the two parameters *usedisc* and *aggrblocks*, that specify, whether the discretization approach should be used for the decomposition and whether identical blocks should be aggregated, respectively.

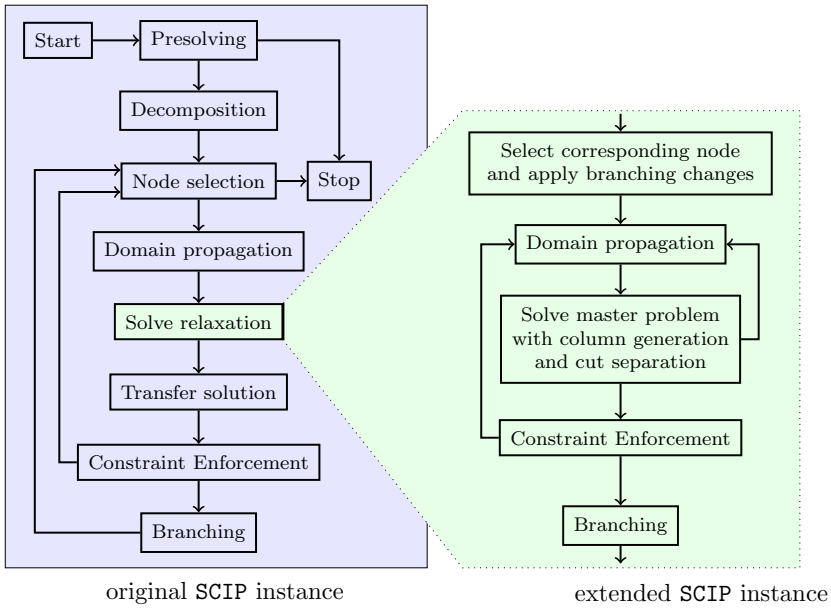


Figure 3.2: Solving process of GCG

The structure of GCG can be interpreted in the following way: We treat the original problem as the essential one, by actually solving it with a "standard" branch-and-bound method, like it is done by most state-of-the-art MIP solvers. The only difference is, that we do not use the LP relaxation to compute lower bounds and corresponding primal solution candidates, but another relaxation, namely the master problem. Hence, the original SCIP instance coordinates the solving process while the extended instance is only used to represent and solve the relaxation.

In the following, we survey the solving process of GCG, the most important parts are described in detail in the following chapters. The main solving loop of GCG is illustrated in Figure 3.2.

Dantzig-Wolfe decomposion In the current version of GCG, information about the structure of the problem has to be provided in an additional file that is read in after reading the MIP. In particular, the number of blocks in the constraint matrix and the variables corresponding to each block can be specified in this file. In addition to that, constraints can be explicitly labelled to be linking constraints (see (2.1) in Model 2.1), which forces these constraints to be transferred to the extended problem (see Model 2.4, 2.8 and 2.10). In the following, we describe the decomposition and the setup of the SCIP instances, an ilustration is given in Figure 3.3.

After the original instance has finished its presolving process, the relaxator performes the Dantzig-Wolfe decomposion and initializes the extended

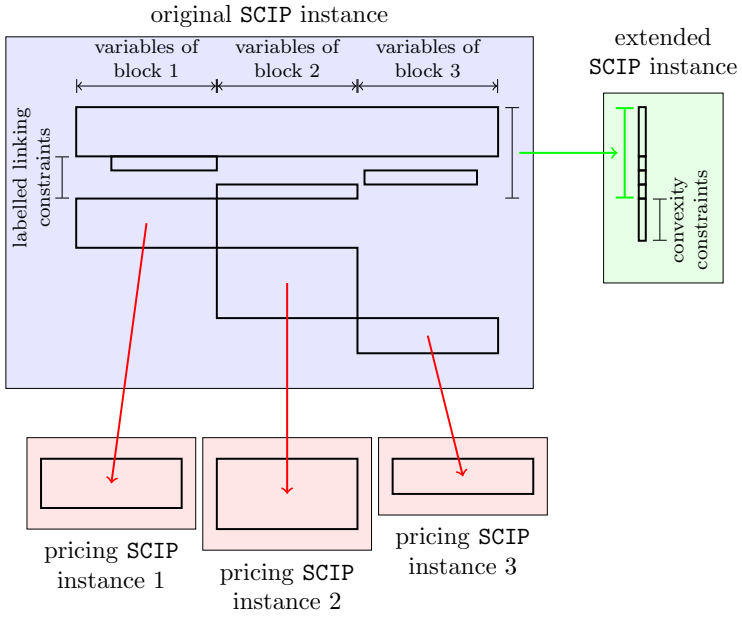


Figure 3.3: Example for the decomposition process of the original problem's constraint matrix. We miss out the right-hand sides of the linear constraint for the sake of clarity. Note that the extended SCIP instance does not contain any variables right from the beginning so the corresponding constraints do neither.

SCIP instance as well as the SCIP instances representing the pricing problems. Initially, the extended instance does contain neither constraints nor variables. The variables that are labelled to be part of a block are copied and added to the corresponding pricing problem: For each constraint of the original instance, it is determined whether it only contains variables of one block. If this is the case, the constraint is regarded as a *structural constraint* of this block and added to the corresponding pricing problem. Otherwise, it is viewed as a *linking constraint*, transformed according to the Dantzig-Wolfe reformulation and added to the extended instance. If a constraint is explicitly labeled to be a linking constraints, it will be transferred to the extended instance even if all contained variables belong to the same block. Furthermore, the convexity constraints (see e.g., (2.11) in Model 2.4) are created in the extended problem.

We do not add variables to the extended instance in this process. This is only done in the solving process of the master problem. Hence, we do not only use a restricted master problem, but we also restrict the extended problem to the same set of variables. Therefore, the constraints in the extended problem do not explicitly contain any variables at the beginning, but they implicitly know the coefficients of all implicitly given variables.

When using the discretization approach and aggregation of blocks, identical blocks are identified during the reformulation process and aggregated. We provide a rather basic check for identity, i.e., in the problem definition, constraints and variables have to be defined in the same sequence for identical blocks. Much more sophisticated methods could be used for this process, but that was not the aim of this thesis.

Coordination of the branch-and-bound trees During the solving process, the extended instance builds the same branch-and-bound tree as the original instance. Each node of the original instance corresponds to a node of the extended instance. In the following, we describe, how this is established. An illustration is given in Figure 3.4.

The connection between a node in the original instance and the corresponding node in the extended instance is established by additional constraint handlers in the original and the extended instance. In a slight abuse of the proper sense of constraint handlers, they do not handle problem restrictions but establish the coordination of both branch-and-bound trees. In the following, we will call the constraint handler in the original instance the *origbranch* constraint handler and the one in the extended instance the *masterbranch* constraint handler. The corresponding constraints are called *origbranch* and *masterbranch* constraints, respectively. These constraints are added to the nodes of the original and extended instance, respectively, to synchronize the solving process of both instances.

Each origbranch constraint knows the node in the original instance it belongs to and the origbranch constraints associated with the father and the children of this node in the branch-and-bound tree. Furthermore, it contains a pointer to the masterbranch constraint of the corresponding node in the extended instance. Each masterbranch constraint knows about the node in the extended instance to which it belongs, the masterbranch constraints corresponding to its father and its children and the origbranch constraint of the corresponding node in the original instance.

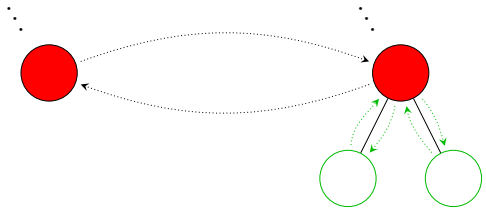
We need this overhead, since we cannot create branch-and-bound nodes of two SCIP instances at the same time. Nodes of one instance can only be created by a branching rule plugin included in this instance.

When branching in the extended instance (see step 2 in Figure 3.4), we just create two children without imposing any further branching restrictions. To each of these children, a masterbranch constraint is added and pointers to these constraints are stored in the masterbranch constraint of the current node. The masterbranch constraints of the child nodes only know the node they belong to as well as the masterbranch constraint corresponding to their father node. The connection to the origbranch constraint of the corresponding node in the original instance is established later.

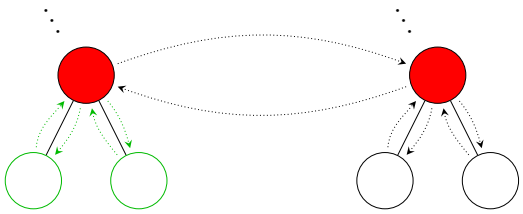
In a subsequent step, branching is performed at the corresponding node in the original instance (see step 3 in Figure 3.4). The branching rule of the original instance creates two child nodes, too. To these child nodes,

1. Connection between the two current nodes established, solve the master problem

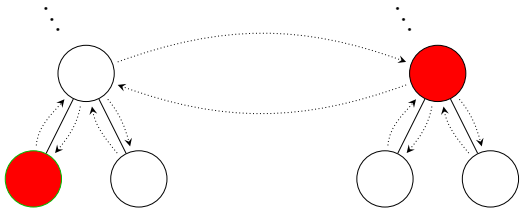
2. Perform branching in the extended SCIP instance



3. Perform branching in the original SCIP instance (if needed)



4. Select next node in the original SCIP instance



5. Select the corresponding node in the extended SCIP instance, establish the connection between the nodes

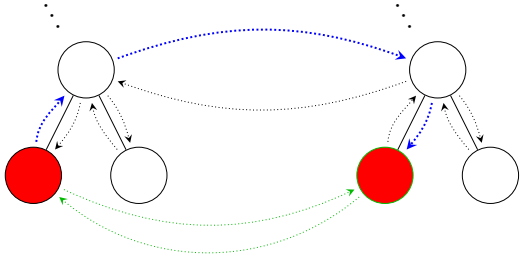


Figure 3.4: The coordination of the branch-and-bound trees. The branch-and-bound tree of the original SCIP instance is pictured on the left side, the one of the extended instance on the right side. We identify the origbranch and masterbranch constraints with the branch-and-bound nodes and represent the pointers stored at the constraints by the dotted lines.

origbranch constraints are added. Like the masterbranch constraints, they initially only know about the corresponding node in the original instance and the origbranch constraint of the father node. Furthermore, the branching rule imposes restrictions on the (original) subproblems corresponding to the child nodes.

The corresponding subproblems in the extended instance were created without any restrictions. In order to transfer the restrictions imposed on the original subproblems to the corresponding nodes in the extended instance, the original branching rule can store information about these restrictions at the origbranch constraints added to the child nodes. Furthermore, it can provide callback functions that are called when a node in the extended instance is activated. These callback functions can be used to finally enforce the branching restrictions in the subproblems of the corresponding nodes in the extended instance. More details about these callback functions and how the branching rules incorporated in GCG use them can be found in Chapter 5.

Node selection We currently use the default node selection rule of SCIP in the original instance, which realizes a best first search. When solving a node in the branch-and-bound tree of the original SCIP instance, the lower bound of the node is not computed by solving its LP relaxation, but the DW relaxator is called for this purpose. It instructs the extended SCIP instance to continue its solving process. A special node selector in the extended instance chooses as the next node to be processed the node corresponding to the current node in the original instance. Its subproblem is always a valid reformulation of the subproblem corresponding to the current node in the original instance.

As described previously, the connection between the nodes was not yet established. It is established by the node selector when it activates the node in the extended instance that corresponds to the current node in the original instance. The origbranch constraint corresponding to the current node in the original instance knows the origbranch constraint of its father node. If there does not exist a father node, then the current node is the root node and the node selector activates the root node in the extended instance, too. Otherwise, the father node was activated before and the corresponding origbranch constraint already knows the corresponding masterbranch constraint. If the current node in the original problem is a left child, then the node selector activates in the extended problem the left child of the node corresponding to the father node of the current node in the original problem (see step 5 in Figure 3.4). Otherwise, it chooses the right child of the node in the extended problem corresponding to the father node. Furthermore, the origbranch and the masterbranch constraints corresponding to the selected nodes in the original and extended instance, respectively, store the pointer to each other.

When the node in the extended instance is activated, one of the previously mentioned callback methods, that is defined by the branching rule in the original instance, is called. Since the branching rule in the original

instance did not know about the corresponding node in the extended instance when it performed branching, this callback method is used to apply the branching restrictions to the selected node in the extended instance. For this purpose, it uses the information about the branching restriction that was stored in the origbranch constraint of the corresponding node and transfers it to the subproblem corresponding to the newly selected node in the extended instance.

Relaxation solving Then, the extended SCIP instance solves the selected node. Therefor, the LP relaxation of the problem corresponding to the node—the master problem—is solved by column generation. For this purpose, we added a variable pricer plugin to the extended instance. An introduction into column generation and a detailed description of the solving process of the master problem are given in Chapter 4. The computed dual bound can be strengthened further by cutting planes (see Chapter 6). During and after this process, primal heuristics are applied that try to find feasible solutions to the extended problem.

After the master problem is solved, the node is pruned if its lower bound exceeds the primal bound. If the solution to the master problem is feasible in the extended problem, we have solved the problem corresponding to the node to optimality. Otherwise, the branching rule of the extended instance is called which simply creates two children in the branch-and-bound tree and the corresponding masterbranch constraints as described previously.

The solving process of the extended problem is then put on hold and the DW relaxator transfers the new information: The dual bound of the node in the extended instance is transferred to the current node of the original instance. Furthermore, the current solution to the master problem and—if existing—new feasible solutions are transformed into the original variable space and added as the current relaxator's solution and new feasible solutions, respectively.

Branching The node in the original instance is pruned if and only if the corresponding node in the extended instance was pruned, too. Both nodes have the same dual bound and since each solution of the extended problem corresponds to a solution of the original problem and vice versa, both instances have the same primal bound, too.

Basically, the relaxator's solution substitutes the solution obtained by solving the LP in the branch-and-bound solving process: It satisfies all but the integrality restrictions of the original problem. The problem corresponding to the current node has been solved to optimality if the integrality restrictions are satisfied, too. Furthermore, this solution guides the branching decisions and it can be used by primal heuristics to derive feasible solutions in the original variable space.

If the node was not pruned, i.e., if the relaxator's solution is fractional and the dual bound does not exceed the primal bound, we perform branching

in the original instance and create two children as described previously. The branching decisions will be transferred to the corresponding nodes in the extended problem later. Different possibilities to perform branching in this context are presented in Chapter 5.

After branching, the original SCIP instance selects a node that is solved next and the process is iterated.

An alternative interpretation of the branch-cut-and-price process

Like stated previously, we treat the original instance as the essential one and actually solve it with a "standard" branch-and-bound method. The LP relaxation is substituted by the DW relaxator that uses the extended SCIP instance to solve the master problem.

Anyhow, solving a MIP using the Dantzig-Wolfe decomposion and branch-cut-and-price can also be interpreted in another way: The problem that is actually solved is the extended problem, we only need the original problem to ensure integrality of the solution in the original variable space and to guide branching decisions. Thus, another possible structure for the implementation would have been to make the extended instance the coordinating one and to store the original problem only for the previously mentioned purposes.

For the discretization approach, integrality can even be directly enforced in terms of the variables of the extended formulation and branching is often performed in terms of these variables as well (see Chapter 5). Hence, we could actually forget about the original problem, solve the extended problem to optimality by branch-cut-and-price and transfer the optimal solution of the extended problem to the original problem afterwards.

Motivation for the employed structure

We had to weight these possibilities against each other and we chose the former one for our implementation since it fits better into the SCIP framework and makes better use of the functionalities already provided by SCIP. The original problem can be read in by the original instance using the default file reader plugins provided by SCIP, so there is a variety of formats that can be read in. If we do not read an additional file defining the structure of the problem, the problem is solved by SCIP with a branch-and-cut algorithm.

On the other hand, if we read such an additional file, the DW relaxator is activated, it creates the second SCIP instance, performes the Dantzig-Wolfe decomposion and substitutes the LP relaxation in the branch-and-bound process. Furthermore, by choosing this structure, we solve both problems simultaneously and can transfer information from one problem to the other during the solving process. Hence, techniques that speed up the solving process, e.g., presolving and domain propagation, can be used in both instances.

We provide possibilities to use either the convexification or the discretization approach, so we had to make sure, that the structure of the implementation fits for both approaches. For the convexification approach, the first

interpretation is the more natural structure, anyway, since integrality is defined in terms of the original variables. For the discretization approach, we can define branching rules that branch in terms of the variables in the extended problem and do not change the original problem explicitly, turning the implementation into an algorithm that essentially realizes the second interpretation.

However, we do not forget about the original problem and under certain conditions, we can conclude that a problem is solved to optimality due to the additional information provided by the original problem: If we only used the extended problem and the solution to the master problem was fractional, we would perform branching unless the node can be pruned. Anyhow, a fractional solution of the master problem may correspond to an integral solution of the original problem, like in the convexification approach. In this case, we have solved the current subproblem since this original integral solution can be retransferred into an optimal solution of the current extended problem.

3.3 Computational Environment

For the computational experiments presented in the following chapters concerning the pricing process (Section 4.4), different branching schemes (Section 5.7), and the separation of cutting planes (Section 6.4), and for final results and comparisons to `SCIP` (Chapter 7), we used four classes of problems: The *bin packing problem (BPP)*, the *vertex coloring problem (VCP)*, the *capacitated p-median problem (CPMP)*, and a *resource allocation problem (RAP)*.

Problem definitions and the different test sets for each of the problem classes are presented in Appendix C. For each test set, we also defined a *small test set* containing less instances. In order to reduce the computational effort, we used these test sets in all computations concerning the performance effect of certain features of `GCG`. As some of these small test sets were defined with respect to the performance of `GCG` for the single instances, we did not use them for the comparison of `GCG` and `SCIP`. In this context, we used the complete test sets; the results are presented in Section 7.4. We present summaries of the results in the according chapters and interpret the results. Detailed results of the computations can be looked up in Appendix D.

For all computations presented in this thesis, we used `GCG` version `0.7.0` and `SCIP` version `1.2.0.5`, which were compiled with `gcc 4.4.0` with the `-O3` optimizer option under `openSUSE 11.2 (64 Bit)`. `Cplex` 12.1 [43] was always used as underlying LP solver for solving the LP relaxations.

The computational tests described in Section 4.4 and 6.4 were carried out on a 3 GHz Intel Xeon with 4 MB cache and 8 GB RAM. In the remaining chapters, the computations for the BPP and the VCP were performed on a 3.60 GHz Intel Pentium D with 2 MB cache and 4 GB RAM, those for the CPMP on a 2.66 GHz Intel Core 2 Quad with 4 MB cache and 4 GB RAM.

The computations for the RAP test sets were performed on a 2.83 GHz Intel Core 2 Quad with 6 MB cache and 16 GB RAM.

When summarizing the results and not listing individual results for each single instance, we report average values for each test set. However, there are different possibilities to define the average values.

Since the values for the instances in our test sets differ highly in their magnitude, we use the *shifted geometric mean*. For non-negative numbers $a_1, \ldots, a_k \in \mathbb{R}_+$, for instance the number of nodes or the final gap of the individual instances of a test set, and a shift $s \in \mathbb{R}_+$, it is defined as

$$\gamma_s(a_1, \ldots, a_k) = \left(\prod_{i=1}^{k} \max\{a_i + s, 1\} \right)^{\frac{1}{k}} - s.$$

It focuses on ratios instead of totals, so it does not overestimate the huge numbers, as it is done by the arithmetic mean. By shifting the values, we prevent the smaller numbers from having a bigger influence which is often encountered for the geometric mean. For a more detailed description and comparison of these three possibilities to compute averages, we refer to [1].

In the summary tables in the following chapters and also in Tables D.1 to D.30 in Appendix D, we just list shifted geometric means for each test set, only the number of timeouts is always given in absolute numbers. All time measurements are given in seconds. We often present percental changes of some settings compared to a "default" setting. In this case, we highlight changes of at least five percent by printing them blue if they represent an improvement and red in case of a deterioration.

We use a shift of $s = 5$ for the LP solving time, $s = 10$ for pricing time, solving time of the master problem and total time as well as for the number of nodes. A shift $s = 100$ is used for the number of pricing problems that are solved, the number of pricing rounds, the number of variables created in the master problem, and the final gap in percent. We used the small shifts for time and nodes since many of the regarded problems are solved in a small amount of time and after a small number of nodes. The other numbers are typically higher so we also used a greater shift. Since all shifts are larger than 1, the maximum in the definition of the shifted geometric mean is always achieved for the first value, so the shifted geometric mean is actually given as

$$\gamma_s(a_1, \ldots, a_k) = \left(\prod_{i=1}^{k} (a_i + s) \right)^{\frac{1}{k}} - s.$$

We define the final gap of an instance by $\frac{|pb-db|}{|db|}$ with pb and db being the global primal and dual bound, respectively, at the moment when the solving process was stopped. The dual bound db is always positive for the instances we regarded, so we do not divide by zero. However, if we did not find a

solution for the instance within the time limit, the gap for this instance is infinity, denoted by a "–" in the tables. In this case, we charged a gap of 100% for the computation of the average. For general MIPs, this value is often exceeded and would therefore not be a good choice, for the problem classes that we examined in this thesis, however, final gaps larger than that did not occur.

In the summary tables, we typically list the shifted geometric mean for each test set. In order to get an average number over all test sets, we compute the shifted geometric mean of the mean values for the single test sets. Since the test sets have different sizes (the small test sets range from 12 to 28 instances), the single instances from the smaller test sets have a greater influence on the overall average. This is intended as we want to give the same weight to each test set. Furthermore, since we defined multiple test sets for some of the problem classes treated in this thesis, these classes of problems are overrepresented in the computation of the average values. However, the different test sets for one class typically differ from each other in terms of problem size so we decided to treat each size the same way as all other test sets. If there are big differences between the results for different classes of problems, we name these differences, anyhow, and do not just consider the average value.

Chapter 4

Solving the Master Problem

As described previously, we solve the original and the extended problem simultaneously. In order to compute dual bounds, we use the LP relaxation of the extended formulation, which we called the master problem (Model 2.5, Model 2.9, or Model 2.14 for the different approaches). As the extended formulation has an exponential number of variables, the master problem does so, too. In this chapter, we describe how we solve the master problem despite this.

We present the concept of *column generation* (sometimes also called *delayed column generation*) in Section 4.1, which is a method to solve LPs with a huge number of variables. Instead of treating all variables explicitly, only a small subset of variables is added to the LP, the remaining variables are considered implicitly and added to the LP only when needed.

In Section 4.2, we illustrate how this concept can be used to solve the master problem obtained by the Dantzig-Wolfe decomposition. After that, in Section 4.3, we describe the integration into the generic branch-cut-and-price framework `GCG` and discuss implementational details. Finally, in Section 4.4, we present computational results for the pricing process based on the problems classes presented in Appendix C.

4.1 Basics in Column Generation

Column generation is a method for solving linear programs with typically a huge number of variables but a reasonable number of constraints.

It is mostly used for solving the LP relaxation of certain (mixed-)integer programs where the variables are not given explicitly but by a specific structure (see [27]). In the set partitionig formulation of the bin packing problem (see Section C.1), for instance, we define a binary variable for each feasible packing of a bin, that is a set of items that can be assigned to a bin without exceeding its capacity. There is a huge number of these packings, so we do not want to enumerate all of them but treat them implicitly.

In this section, we focus on the solution process of such an LP, which is

an important basis for the concepts presented in this thesis.

We start with an LP that explicitly contains all variables. This problem is called the *master problem (MP)*.

Model 4.1 (Master Problem)

$$
\begin{aligned}
z_{MP} = min \quad & c^T x & \\
s.t. \quad & Ax \geq a & (4.1)\\
& Bx = b & (4.2)\\
& x \in \mathbb{Q}^n_+ &
\end{aligned}
$$

with $c \in \mathbb{Q}^n$, $A \in \mathbb{Q}^{m \times n}$, $B \in \mathbb{Q}^{p \times n}$, $a \in \mathbb{Q}^m$, *and* $b \in \mathbb{Q}^p$.

In Chapter 2, we also defined master problems, namely the LP relaxations of the extended fomulations. We did so, since we will solve these problems—which are also LPs with an exponential number of variables—using column generation (see Section 4.2). In this process, they will be the counterpart of Model 4.1. In this section, however, we regard the general master problem defined in Model 4.1.

Storing the whole LP explicitly in the memory of a computer is typically impossible due to the enormous number of variables, so we restrict the problem to a subset of the variables and add variables to this subset only when needed. The resulting problem is called the *restricted master problem (RMP)*.

Model 4.2 (Restricted Master Problem)

$$
\begin{aligned}
z_{RMP} = min \quad & c_N x_N & \\
s.t. \quad & A_{\cdot,N} x_N \geq a & (4.3)\\
& B_{\cdot,N} x_N = b & (4.4)\\
& x_N \in \mathbb{Q}^{|N|}_+ &
\end{aligned}
$$

with $N \subseteq \{1, \ldots, n\}$

In order to define subsets of vectors and matrices, we normally need an ordered set. As we do not need a special ordering in this section, we assume the set N to be ordered increasingly and use it in a slight abuse of notation in order to define subvectors and submatrices.

Each solution to the RMP is also a solution to the master problem, if we set all variables not contained in the master problem to 0. In the following, we assume that this is done whenever we speak of solutions of the RMP that are regarded with respect to the master problem. An optimal solution to the RMP, however, may be suboptimal in the master problem, since considering

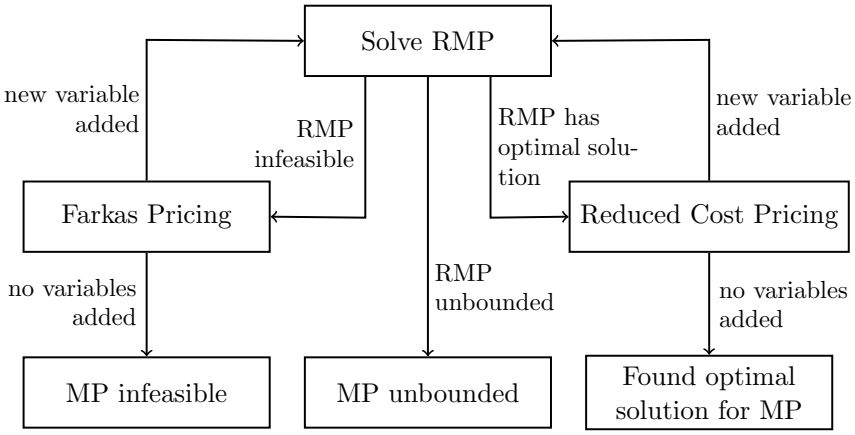


Figure 4.1: The solving process of the master problem

the whole set of variables could lead to a better solution. Nevertheless, we start the solving process of the master by solving the RMP to optimality.

If the RMP is primally unbounded, then this also holds for the master problem (see Theorem 4.7 at the end of this section). We can stop the solving process since we have proven the master problem to be unbounded.

If the RMP is primal infeasible, the master problem might also be primal infeasible. In most cases, however, we only have to add some variables to the set N in order to restore feasibility. In particular, when solving the RMP the first time, we can start without variables, i.e., $N = \emptyset$, so the RMP is infeasible if $a \not\leq 0$ or $b \neq 0$. The process that looks for variables that can help to restore feasibility is called *Farkas pricing* [41, 16] in the following.

In case we get an optimal primal solution to the RMP, we do not know whether it is also optimal for the master problem. Hence, we have to look for variables in the master problem, that could improve that solution and add these variables to the set N of variables of the RMP. We refer to this process as *reduced cost pricing* since we look for variables with negative reduced costs.

After variables have been added, the RMP is resolved and the process is iterated. This is repeated, until no more variable can improve the current solution or can help to fix an infeasibility. If the RMP is still infeasible, then the master problem is infeasible, too. If the RMP has an optimal solution, then this solution is also optimal for the master problem. We will prove these statements at the end of this section.

The solving process of the master problem is illustrated in Figure 4.1.

Once the RMP is feasible, it stays feasible since adding variables conserves feasibility. Hence, Farkas pricing is used until the RMP gets feasible, after that, we use nothing but reduced cost pricing.

In the next two subsections, we specify how the two pricing types work exactly, starting with reduced cost pricing and concluding with Farkas pric-

ing.

Before that, we take a look at the dual of the master problem 4.1 which gives us some more insight into the pricing problems.

Model 4.3 (Dual Master Problem)
For the master problem 4.1, the dual problem looks as follows:

$$\begin{aligned} z_{DP} = max \quad & a^T\pi + b^T\sigma \\ s.t. \quad & A^T\pi + B^T\sigma \leq c \\ & \pi \in \mathbb{Q}_+^m \\ & \sigma \in \mathbb{Q}^p. \end{aligned} \tag{4.5}$$

The vector π represents the dual variables related to the inequality constraints (4.1), σ is associated with the equality constraints (4.2).

As we miss out variables in the RMP, the dual of the restricted master problem does not contain all constraints but only those related to the variables $x_j, j \in N$.

4.1.1 Reduced Cost Pricing

Reduced cost pricing is performed whenever the RMP is solved to optimality, so there exist an optimal primal solution $x^\star$ and the corresponding optimal dual solution $(\pi^\star, \sigma^\star)$.

The reduced cost of a non-negative primal variable x_i with objective function coefficient c_i and coefficients $A_{\cdot,i}$ and $B_{\cdot,i}$ w. r. t. constraints (4.3) and (4.4), respectively, is given as $c_i - A_{\cdot,i}^T\pi - B_{\cdot,i}^T\sigma$ for the current dual solution (π, σ). The corresponding dual constraint is $A_{\cdot,i}^T\pi + B_{\cdot,i}^T\sigma \leq c_i$, so the dual constraint is violated if and only if the corresponding primal variable has negative reduced cost.

Therefore, looking for a variable that could improve the RMP's optimal solution equals searching a constraint of type (4.5) in the dual master problem that is violated by the optimal dual solution $(\pi^\star, \sigma^\star)$ of the restricted master problem.

The *pricing problem*

$$c^\star = \min\left\{c_i - A_{\cdot,i}^T\pi^\star - B_{\cdot,i}^T\sigma^\star \mid i \in [n]\right\} \tag{4.6}$$

computes the smallest reduced cost of all variables in the master problem w. r. t. this dual solution. If $c^\star < 0$, this optimum is achieved for a variable $x_{i^\star}$ and we add this variable to the set N of variables of the RMP. If $c^\star \geq 0$, we have the proof, that the current solution to the RMP is also optimal for the master problem.

In general, the pricing problem could be solved by iterating over all variables and computing their reduced costs, it is, however, usually much more efficient to solve an optimization problem that exploits a certain structure of the variables. For example, for the bin packing problem, where each variable

corresponds to a feasible packing of the items, the pricing problem can be solved by solving a knapsack problem.

4.1.2 Farkas Pricing

If the RMP is infeasible, the *Farkas-Lemma* gives us a possibility to prove this infeasiblity.

Lemma 4.4 (Farkas)
For Model 4.2, exactly one of the following holds:

1. $\exists x_N \in \mathbb{Q}_+^{|N|}$ *with* $A_{\cdot,N} x_N \geq a$, $B_{\cdot,N} x_N = b$
2. $\exists u \in \mathbb{Q}_+^m, v \in \mathbb{Q}^p$ *with* $u^T A_{\cdot,N} + v^T B_{\cdot,N} \leq 0^T$ *and* $u^T a + v^T b > 0$.

For the proof of Lemma 4.4, we refer to [85, 39].

Given $u \in \mathbb{Q}_+^m, v \in \mathbb{Q}^p$ that prove the infeasiblity of the RMP, the inequality

$$(u^T A_{\cdot,N} + v^T B_{\cdot,N}) x_N \geq u^T a + v^T b$$

is a valid inequality in the restricted master problem 4.2, since it is a combination of valid inequalities.

This inequality cannot be satisfied: As x_N is non-negative, $(u^T A_{\cdot,N} + v^T B_{\cdot,N}) x_N$ is non-positive while the right-hand side $u^T a + v^T b$ is strictly positive.

In the dual problem, the vector $(u^T, v^T)^T$ represents a ray; adding this vector to a feasible dual solution will never violate a constraint but improve the objective value of the solution. We call the vectors u and v the *(dual) Farkas multipliers* in the following.

The goal of the Farkas pricing is to now find a variable of the master problem that corrupts this proof and add it to the RMP, or to detect that there is no such variable and the master is infeasible, too. The proof is corrupted if a variable $x_i, i \in [n]$ is found with $u^T A_{\cdot,i} + v^T B_{\cdot,i} > 0^T$. From the dual point of view, this variable induces the dual constraint $A_{\cdot,i}^T \pi + B_{\cdot,i}^T \sigma \leq c_i$ that is always violated when adding a sufficiently large multiple of the vector $(u^T, v^T)^T$ to a dual feasible solution. Therefore, the vector $(u^T, v^T)^T$ is not a feasible ray for the dual of RMP enlarged by x_i, anymore.

The *pricing problem* in this case looks as follows:

$$c^\star = \min \left\{ -A_{\cdot,i}^T u - B_{\cdot,i}^T v \mid i \in [n] \right\}.$$

Hence, the pricing problem for Farkas pricing is similar to the pricing problem for reduced cost pricing. We simply use a zero objective function and the farkas multipliers u and v instead of the dual solution values π and σ.

Finally, let us note that in most implementations of column generation procedures, Farkas pricing is not performed in the explicit form stated previously. Instead, the RMP is often initialized with some variables corresponding

to a previously computed heuristic solution or with artificial variables that are penalized by a *big M* cost, like in the the simplex first phase. However, due to the generic approach presented in this thesis, we restrict ourselves to Farkas pricing for restoring feasibility of the RMP.

4.1.3 Finiteness and Correctness

We conclude this section with two theorems proving the finiteness and the correctness of the column generation procedure as described above.

Theorem 4.5 (Finiteness of the column generation process)
The column generation process terminates after a finite number of iterations.

Proof 4.6
In each iteration, we either are finished and stop or add at least one variable of the master problem to the restricted master problem.

Since the number of variables in the master problem is finite, it is sufficient to show that each variable is added to the RMP at most once.

When performing reduced cost pricing, we add a variable $x_{i^\star}$ to the RMP only if the optimum $c^\star$ of (4.6) is achieved for $i^\star$ and $c^\star < 0$. Since $c^\star$ equals the reduced cost of $x_{i^\star}$, $x_{i^\star}$ has negative reduced cost. The current solution to the RMP, however, is optimal and thus, there exists no variable in the RMP with negative reduced cost. Therefore, $x_{i^\star}$ was not part of the RMP before.

In the case of Farkas pricing, a variable is added only if it corrupts the infeasibility proof given by u and v. For the dual of the current RMP, $(u^T, v^T)^T$ forms a ray; the variable that is added corresponds to a new constraint in the dual of the RMP that limits this ray. If the variable that is added was part of the RMP before, the corresponding dual constraint would have been part of the RMP's dual before, too. This is not possible since $(u^T, v^T)^T$ is a ray in the dual of the current RMP. □

Theorem 4.7 (Equivalence of master problem and RMP)
After the column generation process has finished, the following holds:

- *The RMP is unbounded if and only if the master problem is unbounded.*
- *The RMP is infeasible if and only if the master problem is infeasible.*
- *The RMP has an optimal solution if and only if the master problem has an optimal solution and each optimal solution to the RMP is also optimal for the master problem.*

Proof 4.8
The column generation procedure can be interpreted as a generalization of the simplex method. Instead of explicitly enumerating all variables to find one with negative reduced cost, the pricing step of the simplex method is implicitly performed by solving the pricing problem. However, we give a formal proof for the equivalence in the following.

- *If the RMP is unbounded, there exist a ray* $r \in \mathbb{Q}_+^{|N|}$ *with* $c_N r < 0$, $A_{\cdot,N} r \geq 0$, *and* $B_{\cdot,N} r = 0$ *that proves the unboundedness. Let* $\tilde{r} \in \mathbb{Q}_+^n$ *with* $\tilde{r}_i = r_j$ *if* $i = N_j$ *and* $\tilde{r}_i = 0$, *if* $i \notin N$. *Then* $\tilde{r}$ *is a ray in the master problem proving its unboundedness:* $c\tilde{r} = c_N r < 0$, $A\tilde{r} = A_{\cdot,N} r \geq 0$, *and* $B\tilde{r} = B_{\cdot,N} r = 0$. *Hence, the master problem is also unbounded.*

- *If the RMP is infeasible, then there exist a vector* $(u^T, v^T)^T$ *that proves the infeasibility (see Section 4.1.2). This vector represents a ray in the dual of the RMP with strictly positive objective function value. The dual of the RMP contains the same variables as the dual of the master problem—the RMP contains all constraints—but only a subset of the constraints, since the RMP does not contain all variables contained in the master problem.*

  The column generation process is finished, so we did not add a variable in the last pricing round. Since the RMP is infeasible, we performed Farkas pricing and searched for a variable x_i *in the master satisfying* $u^T A_{\cdot,i} + v^T B_{\cdot,i} > 0^T$. *We did not add a variable to the RMP, thus, such variable does not exist in the master problem. So in the dual of the master problem, adding* $(u^T, v^T)^T$ *to a feasible solution preserves feasibility as it does not increase the left-hand side of any of the constraints.*

  Thus, $(u^T, v^T)^T$ *is also a feasible ray in the dual of the master problem with strictly positive objective function value. Therefore, the master problem is infeasible, too.*

- *Let the RMP have an optimal solution. As the RMP is feasible and the column generation process is finished, we performed reduced cost pricing in the last pricing round but did not add a variable.*

  For the optimal solution of the RMP, we get a corresponding solution $\tilde{x}$ *in the master problem by setting the solution values of all variables not contained in the RMP to* 0. *This solution is also a basic solution, so we can perform a simplex iteration in order to improve it. In the simplex method, we search for a variable in the master that has negative reduced cost w. r. t.* $\tilde{x}$. *This is exactly what was done in the last pricing round and since no variable was added to the RMP, all variables in the master have non-negative reduced cost. Due to the optimality criterion of the simplex method,* $\tilde{x}$ *is an optimal solution of the master problem.*

4.2 Solving the Dantzig-Wolfe Master Problem

When applying the Dantzig-Wolfe decomposition to a problem like described in Chapter 2, we get an extended problem with typically an exponential number of variables. Now, we describe how the master problem—the LP relaxation of this problem—is solved with a column generation approach.

For the two different approaches to the Dantzig-Wolfe decomposition, we get slightly different master problems, Model 2.5 for the convexification approach and Model 2.9 and Model 2.14 for the discretization approach.

In this section, we picture how the master problem of the convexification approach (Model 2.5) is solved. This can easily be transferred to the non-aggregated and to the aggregated master problem for the discretization approach (Model 2.9 and Model 2.14), for the latter, one has to replace the set $[K]$ of all blocks by the set $[L]$ of partitions. Hence, whenever we mention the master problem in this section, we refer to Model 2.5.

The master problem has less constraints than the original problem, but in general exponentially many, implicitly given variables. Even if we would generate all variables, reading in such a big LP would consume much memory and solving it would be computationally hard.

In order to avoid this and since most of the variables will not be needed in an optimal solution of the master problem, we use the concept of column generation, which was introduced in the last section, to solve the master problem to optimality. We focus on the reduced cost pricing in the following, Farkas pricing is handled in the same way, replacing the dual solution values by the Farkas multipliers and omitting the objective function coefficients.

As described in the last section, we regard a *restricted master problem (RMP)* in which we do not consider all extreme points and extreme rays, but only subsets $\bar{P}_k \subseteq P_k$ and $\bar{R}_k \subseteq R_k$ for $k \in [K]$. Variables are added to these subsets only when needed to improve the current solution or to repair an infeasibility.

Before we state the pricing problem, we first take a look at the dual of the master problem (Model 2.5).

Model 4.9 (Dual Master Problem)

$$
\begin{aligned}
z^{\star}_{DMP} = max \quad & \sum_{i=1}^{m_A} b_i \pi_i + \sum_{k \in [K]} \rho_k & & \\
s.t. \quad & {a_p^k}^T \pi + \rho_k \leq c_p^k & \forall p \in P_k, k \in [K] \quad & (4.7) \\
& {a_r^k}^T \pi \leq c_r^k & \forall r \in R_k, k \in [K] \quad & (4.8) \\
& \pi \geq 0 & & \\
& \rho \quad free & &
\end{aligned}
$$

The vector $\pi \in \mathbb{K}_+^{m_A}$ represents the dual variables related to the linking constraints (2.15) while $\rho \in \mathbb{K}^K$ refers to the convexity constraints (2.16).

As we miss out variables in the restricted master problem, the dual of the restricted master does not contain all constraints of type (4.7) and (4.8) but only those related to the extreme points $p \in \bar{P}_k, k \in [K]$ and extreme rays $r \in \bar{R}_k, k \in [K]$.

Looking for a variable that could improve the optimal solution of the RMP equals searching a constraint of type (4.7) or (4.8) in the master problem that is violated by the current dual solution of the restricted master problem.

For a given block $k \in [K]$, the reduced cost of the variable corresponding to an extreme point $p \in P_k$ are given as

$$\bar{c}_p^k = c_p^k - (\pi^T a_p^k + \rho_k) \overset{(2.9)}{=} {c_k}^T p - (\pi^T A^k p + \rho_k)$$

while an extreme ray $r \in R_k$ has corresponding reduced cost

$$\bar{c}_r^k = c_r^k - \pi^T a_r^k \overset{(2.9)}{=} {c_k}^T r - \pi^T A^k r.$$

We get a variable with negative reduced cost belonging to block k by solving the MIP

$$\bar{c}_k^\star = \min \left\{ \left({c_k}^T - \pi^T A^k \right) x - \rho_k \mid x \in X_k \right\}. \tag{4.9}$$

If $\bar{c}_k^\star \geq 0$, there exists no column belonging to block k with negative reduced cost, so no extreme point or extreme ray of $conv(X_k)$ can improve the RMP's current solution.

If $\bar{c}_k^\star < 0$ and finite, the optimum is achieved at an extreme point p of $conv(X_k)$ which has negative reduced cost w. r. t. the current solution of the master problem. Thus, we can add a column to the RMP with objective function coefficient ${c_k}^T p$ and coefficients $A^k p$ and 1 in constraints (2.15) and (2.16), respectively.

If $\bar{c}_k^\star = -\infty$, then there exists an extreme ray r of $conv(X_k)$ with $\left({c_k}^T - \pi^T A^k \right) r < 0$ and we can add a column with objective function coefficient ${c_k}^T r$ and coefficients $A^k r$ and 0 in constraints (2.15) and (2.16), respectively.

In order to solve the master problem to optimality, we start with solving the RMP to optimality. Afterwards, we solve the pricing problems (4.9) for all blocks $k \in [K]$. If we find a new variable with negative reduced cost, we add it to the RMP and resolve it. We repeat this until all pricing MIPs have non-negative objective values so that there exists no variable in the master problem with negative reduced cost w. r. t. the current solution of the RMP. The latter is therefore also optimal for the master problem.

There are multiple degrees of freedom concerning the pricing of new variables. We can stop the pricing routine after adding one variable, search the variable with the most negative reduced costs, take all optimal solutions with negative reduced costs or much more. We will discuss these options in the next section.

Naturally, the optimal objective value $z_{RMP}^\star$ of the RMP in each iteration of the column generation process is an upper bound on the optimal objective value $z_{MP}^\star$ of the master problem since it converges from above towards the optimal master solution value. It is, however, in general not an upper bound on the optimal objective value $z_{OP}^\star$ of the original problem, since the master

problem does not respect the integrality restrictions in the original program. Only if the current solution tranfers into a solution of the original program that satisfies the integrality restrictions, the current solution to the RMP corresponds to a feasible solution to the master problem and $z^\star_{RMP}$ is thus an upper bound on $z^\star_{OP}$.

Additionally, even if the master problem is not yet solved to optimality, the optimal objective value $z^\star_{RMP}$ of the RMP can be used to compute a lower bound in case we know the optimal solution values $\bar{c}^\star_k$ to the pricing problems (4.9) for all $k \in [K]$.

Theorem 4.10 ([97])
Let $z^\star_{RMP}$ be the optimal objective value of the current RMP and $\bar{c}^\star_k$, $k \in [K]$ be the solution values to the pricing MIPs (4.9) w. r. t. the the RMP's current solution. Then

$$LB_{RMP} = z^\star_{RMP} + \sum_{k \in [K]} \bar{c}^\star_k \leq z^\star_{OP}.$$

is a valid lower bound on the optimal objective value of the master problem.

4.3 Implementation Details

The solving process of the master problem is a crucial part of the branch-cut-and-price solver in terms of performance: it typically consumes most of the running time. Thus, it is very important to perform it as efficient as possible.

As described in the last sections, after solving the RMP, we solve pricing problems of the form (4.9). They give rise to further variables that can be added to the RMP to improve its solution—or fix an infeasibility—or the proof, that the solvability of the RMP equals that of the master problem and an optimal solution to the RMP—if existent—is also optimal for the master problem.

The pricing process is structured into two components in our implementation: the *variable pricer* and a set of *pricing solvers*. The former coordinates the pricing process, while the latter are called by the variable pricer to solve a specific pricing problem. `SCIP` itself supports the use of variable pricers (see Section 3.1 and [1, Chapter 3]), so we implemented a plugin of this type and included it into the `SCIP` instance corresponding to the extended problem. This type of plugin has two essential callbacks that are called during the pricing process, one for Farkas pricing, the other for reduced cost pricing. The former is called by `SCIP` whenever the RMP is infeasible, the latter is called if the RMP is feasible. This is repeated until no more variables were added in the last pricing round.

The concept of pricing solvers was added to the project by the author of this thesis. They are used by the pricer in a black box fashion: Whenever a specific pricing problem should be solved, it is given to the set of solvers. The pricing problem is solved by one of the solvers and a set of solutions is

returned. We chose this concept, which is similar to the way the LP solver is handled in SCIP, in order to provide a possiblity to add further solvers without the need to change the inner structure of the branch-cut-and-price solver.

Although the usage of a problem specific pricing solver can improve the performance of the pricing process by far, we will in the following focus on the influence of the pricer itself since we discuss a generic approach to branch-cut-and-price. Therefore, the default pricing solver incorporated in the current version of GCG is used, which solves the pricing problem as a MIP using SCIP. This is the most general pricing solver: each pricing problem is formulated as a MIP, so the solver can solve all kinds of pricing problems. Solving specific subproblems within a branch-and-bound algorithm by modeling them as a MIP (a so-called "subMip") and solving them with a MIP solver has previously been applied successfully in primal heuristics and separators, for a survey see [30].

The second pricing solver included in the current version of GCG is a knapsack solver. It serves as an example for a special purpose pricing solver. It will be used in Chapter 7 to demonstrate the potential savings in computational time when using such a solver.

Since the goal of this thesis was to develop a generic branch-cut-and-price solver, we will concentrate on the MIP pricing solver, which was also used for most of the computations presented in this thesis.

In the remainder of this section, we will discuss implementational details and parameters that can be adjusted to tune the pricing process. An overview of the parameter provided by GCG, the symbols used for them in this thesis and the effect of these parameters can be found in Appendix B. We will start with the Farkas pricing in Section 4.3.1 and deal with the reduced cost pricing in Section 4.3.2. In the current version of GCG, we assume the pricing problems to be bounded, so we only have to create variables corresponding to extreme points in the pricing routine. Although this is to be extended in the future, we only cover the case of creating extreme points in the following as it also eases the presentation. Also for clarity reasons, we base the discussion on the master problem of the convexification approach (Model 2.5). For the discretization approach, pricing is performed in the same way, only when aggregating blocks, the number of blocks that are actually treated changes from K to L. In Section 4.4, we finally present computational results concerning the described methods.

4.3.1 Farkas Pricing

When the current RMP is infeasible, the SCIP instance representing the extended problem automatically calls the variable pricer that we implemented. The infeasibility proof by Farkas multipliers is provided by the LP solver. The task of the Farkas pricing is to find a variable that corresponds to a dual constraint that cuts off that ray.

Farkas pricing problem

Given dual Farkas multipliers $u \in \mathbb{Q}_+^{m_A}$ corresponding to the linking constraints and $v \in \mathbb{Q}^K$ corresponding to the convexity constraints, find a block $k \in [K]$ and a point $p \in P_k$ of this block with

$$u^T A^k p + v_k > 0 \tag{4.10}$$

or conclude that no such point exists.

If we have only one block ($K = 1$), the implementation is straightforward: We solve the pricing problem

$$\bar{c}_k^\star = \max \left\{ u^T A^k x^k + v_k \mid x^k \in X_k \right\}. \tag{4.11}$$

The solution corresponds to a point $p \in P_k$. If $\bar{c}_k^\star > 0$, the corresponding extreme point is added to the set $\bar{P}_k$ of points regarded in the RMP. If $\bar{c}_k^\star \leq 0$, the infeasibility of the master problem is proven. The pricing round is finished and `SCIP` calls the LP solver to reoptimize the RMP if and only if variables were added in this pricing round. Afterwards, the variable pricer is called again, if needed.

For the case of K blocks (and if these blocks were not aggregated for the discretization approach), it gets more involved. For each block $k \in [K]$, we have a pricing problem of the form (4.11) and we have to solve these pricing problems consecutively. During this process, we can stop solving the pricing problems once we find a solution with positive objective function value. However, we can also continue the pricing process and try to find multiple variables. This implies more effort since we solve more pricing problems, but it possibly gives rise to more variables to add to the RMP which hopefully reduces the number of Farkas pricing rounds needed to restore feasibility of the RMP. However, one variable suffices to cut off the dual ray $(u^T, v^T)^T$ and thus, it is not clear whether adding more variables speeds up the solving process. In particular, if the ray may be cut off only by variables of a specific block, solving all remaining pricing problem without the hope to find other variables is unnecessary.

Hence, we decided to stop the pricing procedure after finding one variable that cuts off the dual ray and add just this variable to the RMP in the default settings. Nevertheless, we introduced a parameter $M_f \in [1, \infty]$ that defines the maximum number of variables that are added in one Farkas pricing round. In addition to that, using the parameter $R_f \in [0, 1]$, a relative limit on the number of pricing problems that are solved in a pricing round can be imposed. It is a relative limit, so setting $R_f = 0.5$ corresponds to solving half of the pricing problems. If new variables were added after solving this part of the pricing problems, the current pricing round is stopped, otherwise, it is continued until a problem gives rise to variables that cut off the dual ray. Note, that for all solutions of these problems with positive objective function

value, that were found during the solving process of the pricing problem, we add a variable to the RMP, in particular also for suboptimal solutions. By setting $R_f = 0$, the pricing process is stopped after a pricing problem with positive optimal objective function value is found and a variable is added for each solution with positive objective function value, that was found during the solving process.

In case we do not solve all pricing problems in a Farkas pricing round due to these parameters, the order in which the pricing problems are examined is very important. The goal is to solve those pricing problems at first that are more likely to have a positive optimum. This way, we hope that the pricing procedure can be stopped after solving a small number of pricing problems.

A possible order of the pricing problems can be defined in terms of the dual Farkas multipliers v associated with the convexity constraints of the blocks. A higher value corresponds to a higher probability that the optimal objective function value of the problem is positive, especially if we have no estimate for the value $u^T A^k x^k$ of optimal solutions x^k of the different blocks $k \in [K]$. Hence, we solve the pricing problems in decreasing order of the dual Farkas multipliers v_k.

Input: Farkas multipliers $u \in \mathbb{Q}^{m_A}$ and $v \in \mathbb{Q}^K$ w. r. t. linking and convexity constraints, respectively

1 Sort the pricing problems $k \in K$ by v_k in descending order $k_1, \ldots, k_K$
2 **for** $i \leftarrow 1$ **to** K **do**
3 Solve the pricing problem

$$\bar{c}^\star_{k_i} \leftarrow \max \left\{ u^T A^{k_i} x + v_{k_i} \mid x \in X_{k_i} \right\}.$$

4 **if** $\bar{c}^\star_{k_i} > 0$ **then**
5 Add the computed optimal solution $x^\star$ to the set $\bar{P}_{k_i}$
6 **stop!**

Algorithm 1: Default Farkas Pricing Algorithm

For the problems that we regard in this thesis, this approach outperformes solving the problems in their natural order, i.e., by increasing block number k, by far. We have to admit that most of the regarded problems have rather similar blocks and a set partitioning structure in the master problem. Therefore, the sets X_k of solutions to the pricing problems of the blocks are similar and these variables lead to similar values $u^T A^k x^k$ so that the optimal objective function value of the pricing problems depends in large part on the dual Farkas multipliers. However, many of the problems that are solved with a Dantzig-Wolfe decomposition approach have that structure, so solving the pricing problems in the specified order should perform well for most of these problems. Nevertheless, for rather different blocks and arbitrary structure in

the master problem, defining another order could be more efficient.

The default Farkas pricing method is pictured in Algorithm 1. In Section 4.4.1, we give a comparison to some other settings.

4.3.2 Reduced Cost Pricing

In the reduced cost pricing routine, we look, from the theoretical point of view, for a variable of the master problem with the most negative reduced cost. This variable is added to the RMP and the process is iterated. For the master problem obtained by the Dantzig-Wolfe decomposition, this translates into solving the pricing problem (4.9) for each block $k \in [K]$. After that, if and only if the best optimal objective value is negative, a variable is added to the RMP that represents the extreme point given by the computed optimal solution.

By solving all pricing problems, however, we potentially get a whole bunch of variables that can improve the RMP's current solution, namely for each pricing problem with negative optimal objective value at least the variable corresponding to the optimal solution. If we found further (suboptimal) solutions with negative objective value during the solving process of the pricing problems, we could add these variables as well. Note, that in Section 4.1, we added only one variable to the RMP per pricing round, however, finiteness and correctness of the pricing process are obviously conserved when adding more than one variable. In this case, the size of the RMP grows faster and we run the risk of creating variables that will never be part of the basis which leads to a slowdown of the simplex algorithm. We have to trade this off against the benefit that variables are created without further effort, which would possibly have been created in one of the successive pricing rounds, anyway.

Additionally, when adding only the variable with the most negative reduced cost, this variable will definitely be chosen to enter the basis, which equals to the so-called Dantzig pivot rule of the simplex method. Computational experiments (see [32]), however, have shown that this pivot rule is mostly inferior to more sophisticated rules, like steepest-edge pivot rules. We will not go into details about pivot rules of the simplex method in this thesis and refer to [85, 32] and the references given there, but we want to give an indication, that adding more than just one variable to the RMP per round can help to reduce the number of simplex iterations during the solving process of the master problem. Since the LP solver is treated as a black box, we do not know which pivot rule is used by it, thus we cannot modify the pricing process to search the variable with the highest priority according to this pivot rule and add just this variable. However, when adding more than one variable, the LP solver can choose the "best" of the new variables to enter the basis according to the pivot rule employed by it. Additionally, after performing one simplex iteration, there is the possibility that one of the other newly created variables still has negative reduced cost and can further

improve the objective function value. This way, we can reduce the number of times pricing has to be performed and hopefully also the total time spent for the solving process of the master problem.

Another approach to reduce the effort of the pricing routine is to stop the pricing process after a given number of variables with negative reduced costs were found in this pricing round, add these variables to the RMP and iterate the process. That way, we do not solve the pricing problem as stated in Section 4.1, since we do not calculate the most negative reduced cost. We find, however, variables with negative reduced cost that can enter the basis when the RMP is re-optimized. Finiteness and correctness of the procedure is assured as long as we do not abort the pricing process before finding a new variable. Thus, at least in the last pricing round, we have to solve all pricing problems to optimality in order to guarantee that no variable with negative reduced cost exists in the master problem that can improve the solution of the RMP.

We included these approaches into our branch-cut-and-price solver `GCG` and provide some parameters to tune them. In the following, we will name the most important parameters and their influence.

First of all, let us notice that without imposing limits by parameters, we solve all pricing problems and add all variables with negative reduced cost to the RMP that were found in this pricing round. In particular, we also add suboptimal solutions of the pricing problems as long as they have negative reduced cost.

Since these can be quite a lot of variables, a maximum number $M_r \in [1, \infty]$ of variables that are added per reduced cost pricing round can be specified. In the default setting, it is set to 100.

In addition to that, we can choose whether the pricing routine is stopped after this number of variables with negative reduced cost was found or, alternatively, all pricing problems are solved and after that, the M_r variables with the most negative redced costs are added to the RMP. By choosing the second possiblity, the parameter is called *onlybest*, together with a limit $M_r = 1$, we perform pricing as stated in Section 4.1, i.e., we only add one variable with the most negative reduced cost to the RMP.

Furthermore, we can limit the number of variables created for one block in a pricing round by a value $M_p \in [1, \infty]$. Setting this value to 1 corresponds to taking into account only variables corresponding to an optimal solution of a pricing problems while a greater value allows to create variables for suboptimal solutions, too. In the default setting, we set $M_p = \infty$. Apparently, the parameters M_p and M_r overlap in some cases: By setting $M_p \geq M_r$ or $M_p \leq \frac{M_r}{K}$, one of these parameters becomes redundant.

Like for the farkas pricing, we introduced a parameter $R_r \in [0, 1]$. It is a relative limit on the number of pricing problems that are solved in each pricing round. If the limit is reached and new variables were added in the current pricing round, the round is stopped, otherwise, it is continued until improving variables have been found.

In case we do not solve all of the pricing poblems in each pricing round due to the parameter settings, it is also important, in which order the pricing problems are examined. If we start with solving those pricing problems that are expected to have the more negative solutions, we avoid solving instances that are likely to have a non-negative optimum. The order can be specified by the parameter *sorting*. The pricing problems can be solved in their natural order, i.e., by increasing k ($sorting = 0$) or by decreasing dual solution value of the convexity constraint corresponding to that block ($sorting = 1$), like it is also done for the Farkas pricing. This makes use of the fact that for each block, the negation of the corresponding dual solution value goes into the reduced costs of the variables corresponding to that block. Hence, the higher the value, the more likely it is that variables of the block have negative reduced cost. It turned out that like for the Farkas pricing, this ordering of the pricing problems is superior, so it is used per default.

Another approach to speed up the pricing process is the use of heuristics. Instead of solving the pricing problems to optimality, we use a heuristic that is expected to find "good" solutions in a reasonable amount of time. For this purpose, each pricing solver can provide a routine that solves the given pricing problem in a heuristic fashion. If a heuristic finds variables with negative reduced costs, we add them to the RMP, according to the previously defined settings. After that, the RMP is resolved and the process is iterated. If the heuristic does not find any variable with negative reduced cost, however, we have to solve the pricing problems to optimality in order to find an improving variable or to prove that no such variable exists. The usage of heuristics can be turned on with the parameter *useheur*, per default, it is disabled.

For the MIP pricing solver, solving the problem heuristically is done by solving the pricing problem with `SCIP`, too. In order to limit the effort spent for the heuristic call of the pricing solver, we limit the number of solving nodes and the number of stalling nodes. The first one is a hard limit on the number of nodes that are at most processed, the latter defines the number of nodes, that may be processed without finding an improving solution. Additionally, we set a limit on the objective value of solutions, thus only solutions with negative reduced costs are accepted and we cancel the solving process once a given gap has been reached.

The reduced cost pricing process is presented in Algorithm 2.

4.3.3 Making Use of Bounds

In Section 4.2, we defined a lower bound LB_{RMP} on the optimal objective value of the master problem that can be computed using the current optimal solution $z^\star_{RMP}$ of the RMP and the optimal solutions $\bar{c}^\star_k, k \in [K]$ of all pricing problems (see Theorem 4.10). In the following, we describe how we use this bound to speed up the solving process.

Input: An optimal dual solution of the RMP $\pi \in \mathbb{Q}^{m_A}$, $\rho \in \mathbb{Q}^K$ w. r. t. linking and convexity constraints, respectively

1 nvars $\leftarrow 0$ `// number of variables added`
2 heur $\leftarrow$ *useheur* `// solve pricing problems heuristically?`
3 V $\leftarrow \emptyset$ `// set of variables found so far`
4 Sort the pricing problems $k \in [K]$ according to the selected sorting method and obtain an order $k_1, \ldots, k_K$
5 **for** $i \leftarrow 1$ **to** K **do**
6 nvarsprob $\leftarrow 0$ `// nr. of variables found in this problem`
7 Solve the pricing problem

$$\bar{c}_k^\star \leftarrow \min \left\{ \left({c_{k_i}}^T - \pi^T A^{k_i} \right) x - \rho_{k_i} \mid x \in X_{k_i} \right\} \tag{4.12}$$

 heuristically, if heur = TRUE, otherwise, solve it to optimality.
8 **if** $\bar{c}_{k_i}^\star < 0$ **then**
9 **for** *each solution* $x^\star$ *of* (4.12) *that is found (sorted increasingly w. r. t. objective function value)* **do**
10 **if** $\left({c_{k_i}}^T - \pi^T A^{k_i} \right) x^\star - \rho_{k_i} < 0$ **then**
11 **if** *onlybest* **then**
12 V $\leftarrow$ V $\cup \{(x^\star, i)\}$
13 nvarsprob $\leftarrow$ nvarsprob $+ 1$
14 **else**
15 $\bar{P}_{k_i} \leftarrow \bar{P}_{k_i} \cup \{x^\star\}$
16 nvars $\leftarrow$ nvars $+ 1$
17 nvarsprob $\leftarrow$ nvarsprob $+ 1$
18 **if** *nvarsprob* $= M_p$ **or** nvars $= M_r$ **then**
19 **break!**
20 **if** nvars $= M_r$ **or** *(*$i \geq R_r \cdot K$ **and** nvars > 0*)* **then**
21 **break!**
22 **if** *onlybest* **then**
23 Sort the set V w. r. t. the reduced costs of the saved variables
24 **for** $(x^\star, i) \in$ V *in increasing order w. r. t. reduced costs* **do**
25 $\bar{P}_{k_i} \leftarrow \bar{P}_{k_i} \cup \{x^\star\}$
26 nvars $\leftarrow$ nvars $+ 1$
27 **if** nvars $= M_r$ **then**
28 **break!**
29 **if** nvars $= 0$ **and** *heur* $=$ *TRUE* **then**
30 heur $\leftarrow$ FALSE
31 **Goto** Line 5

Algorithm 2: Reduced Cost Pricing Algorithm

Early branching First, it is possible to define a maximum number of reduced cost pricing rounds $maxrounds \in [0, \infty]$ to be performed at each node of the branch-and-price tree.

When creating a node, `SCIP` initially sets its local dual bound to the dual bound of the father node, for the root node, it is set to minus infinity. This local bound is automatically updated once the LP relaxation at the node is solved to optimality. If $maxrounds < \infty$ and this number is exceeded for a branch-and-price node, we do not know whether further variables with negative reduced cost exist in the master problem. Thus, we did not solve the master problem to optimality and the RMP's optimal objective function value is not a valid lower bound for the optimal solution value of the original program.

However, the reduced cost pricing callback of variable pricers in `SCIP` provides the possibility to specify a lower bound on the optimal objective value of the local master problem. `SCIP` then automatically updates the dual bound of the node to this value if it is better, i.e., higher, than its current dual bound. Hence, in each pricing round in which the RMP and all pricing problems are solved to optimality, the pricer computes the lower bound LB_{RMP} and returns this value so that `SCIP` updates the dual bound of the node, if possible.

The behavior resulting from setting $maxrounds < \infty$ is denoted by *early branching* in the following, since the solving of the current node is interrupted and branching is performed "earlier".

We decided to give no possibility to limit the number of Farkas pricing rounds, since then we would not have a feasible solution to the current master problem and would not even know whether such a solution exists. On the one hand, if the current master problem is infeasible, proving this by completing the Farkas pricing process allows us to prune the current node in the branch-and-bound tree. On the other hand, if it is not infeasible, we will have to restore the feasibility in its children, anyway. All the dual rays providing an infeasibility proof at the current node can still proof the infeasibility in the subsequent nodes, so they have to be forbidden by adding new variables to the RMP, anyhow. Besides, without knowing a feasible solution of the master problem, we do not have the guidance for the branching process that is usually provided by the relaxator's current solution.

Early branching typically results in a slower increase of the dual bound. In return, the branch-and-price tree is explored faster and nodes that are situated deeper in the branch-and-price tree are investigated earlier. Since these nodes correspond to problems that are more likely to have an integral solution (e.g., since some variables are fixed to integral values), it may result in feasible solutions being found earlier in the solving process.

Due to the weaker dual bound, this is primarily useful when we do not want to solve the problem to optimality, but want to find a solution with a given quality, e.g., a solution that has at most 10% additional cost compared to the optimal solution. Besides, this concept is often used for primal

heuristics. For example, pricing is performed only at the root node. After that, one hopes, that the current set of variables contains a feasible solution to the extended problem and tries to find a solution quickly by branching in a depth-first-search way (see [10]).

Per default, we set $maxrounds = \infty$ so the number of pricing rounds per node is not limited.

Early termination Nevertheless, we use a weakened form of early branching, also called *early termination* of the pricing process (see [97]) even with the default settings. In case we know that the objective function value of each feasible solution is integral—e.g., if the original problem's objective function contains only integer variables with integral coefficients—we stop the pricing process if further improvements of the node's dual bound will never have any advantages with regard to the bounding process. In the following, we describe how this is detected.

Let $z^\star_{RMP}$ be the optimal objective value of the RMP and LB be the current dual bound of the active node, i.e., the maximum of the parent node's dual bound and all feasible lower bounds LB_{RMP} computed in former pricing rounds at the current node. By solving the master problem to optimality, we obtain a lower bound $z^\star_{MP}$ on the optimal objective function value of the current original problem. This one lies between the two former values, i.e., $LB \leq z^\star_{MP} \leq z^\star_{RMP}$ (see Section 4.2).

If $\lceil LB \rceil \geq z^\star_{RMP}$ then it follows $\lceil LB \rceil = \lceil z^\star_{MP} \rceil$. This means, that the best feasible solution for the current problem has objective value at least $\lceil LB \rceil$, so we can set the lower bound of the node to $\lceil LB \rceil$. Solving the master problem to optimality can lead to no better dual bound, thus we can stop the solving process of the master problem. Actually, the dual bound of the node does not have to be updated to $\lceil LB \rceil$ as `SCIP` knows about the integrality of the objective function value of each solution. Therefore, it prunes the node as soon as a solution with value $\lceil LB \rceil$ or better is found, even if the dual bound LB is maintained.

This result can easily be generalized to problems, for which $z \in \mathbb{Q}$ exists such that each feasible solution has value $k \cdot z$ with $k \in \mathbb{Z}$: if $\lceil \frac{LB}{z} \rceil \geq \frac{z^\star_{RMP}}{z}$, the solving process of the master at the current node can be stopped.

By using early termination, we reduce the so-called *tailing-off effect* [57]. That means, that the column generation process finds a near optimal solution in a reasonable amount of time, but obtaining optimality and proving it typically needs many pricing rounds. Furthermore, the pricing problems often get much harder to solve when the dual variables converge to the optimal dual solution of the master problem. The long tail of computations needed to prove the optimality is avoided by stopping the column generation earlier.

However, computing the exact lower bound by solving the master problem to optimality can improve the effectiveness of methods that make use of so-called *pseudocosts*. Pseudocosts denote the average increase in the dual bound after tightening the bound of a certain variable (also see Section 5.6.1).

They are used for example by branching rules [5], node selection strategies [1], and some primal heuristics [11].

Nevertheless, our computational studies (see Chapter 7) show that the effort saved in the pricing routine compensates this disadvantage and the overall computation time is decreased when using early termination. However, it can be deactivated with the *abortpricing* parameter.

Objective limit for LP solving Finally, let us note that the RMP is not always solved to optimality before the pricing plugin is called. Like most state-of-the-art MIP solvers, `SCIP` imposes a limit on the objective function value of the LP. When solving the LP with the dual simplex algorithm, e.g., after adding branching restrictions, the solving process of the LP is stopped as soon as the objective value of the current LP solution is greater or equal to the primal bound. By continuing the solving process of the LP, we would obtain an optimal solution with an objective value not smaller than the current objective value of the LP, hence also not smaller than the primal bound. If the LP contains all variables explicitly, i.e., no pricers are active, the current node can be pruned since the current problem cannot contain solutions that are better than the incumbent.

If pricers are active, however, the current LP is just a restricted version of the implicitly given actual LP. Therefore, by adding more variables to the restricted LP, we can improve its optimal objective value and will finally get an optimal solution for the actual LP. This solution can have a better objective value than the optimal solution of the restricted LP, hence the node must not be pruned. Instead, in order to obtain this optimal solution of the actual LP, pricing has to be performed. If and only if the optimal objective value is then still greater or equal to the primal bound, the node can be pruned.

When the solving process of the LP was stopped due to the objective limit, however, we do not need to solve the restricted LP to optimality. We can use the current dual solution values, since this solution has to be forbidden in the dual of the LP in order to obtain a primal LP solution that is better than the primal bound. If we find variables with negative reduced costs, we add them to the LP and iterate the process. If we do not find any variable with negative reduced cost, then the current dual solution is feasible in the dual of the actual LP and thus, the optimal solution value of the actual LP is greater or equal to the primal bound and the node can be pruned.

4.4 Computational Results

In this section, we describe our experiences of the solving process of the master problem. We investigate different settings and variants mentioned in the last section and compare their effectiveness.

We focus on the master problem at the root node and investigate the effort

needed to obtain feasibility of the RMP in Section 4.4.1. In Section 4.4.2, we look at the process of solving the master problem to optimality. However, we do not regard the impact on the performance of the total branch-and-price process. We cannot guarantee that pricing strategies that dominate the solving process of the master problem at the root node are also superior for the branch-and-price process. For example, it could be an advantage to create more variables at the root node even if the pricing process at the root node takes slightly longer in this case, since the additional variables could speed up the solving process of the master problem at subsequent nodes. On the other hand, adding much more variables also slows down the simplex algorithm and it is not clear whether these variables help in finding good primal feasible solutions or actually derange this process.

Since the branch-and-bound process and different branching rules were not yet introduced (this will be done in the next chapter), we will in the following regard several possible pricing strategies and name a subset that seems to perform well. In Chapter 7, we will then compare some strategies with respect to the branch-and-price process.

4.4.1 Farkas Pricing

We ran benchmarks to compare the performance of the default Farkas pricing settings (see Algorithm 1) with the following variations.

- "All vars": all pricing problems are solved in each pricing round and all variables are added to the RMP that cut off the dual ray, i.e., fulfill condition (4.10). This corresponds to $M_f = \infty$.
- "No sort.": like in the default setting, just one variable is added to the RMP per pricing round, but the pricing problems are not sorted w. r. t. the dual Farkas multipliers v, instead they are solved in their natural order.
- "2 vars": the pricing problems are sorted as for the default settings, but instead of adding just the first variable, we add up to two variables. With this setting, we want to investigate, whether adding not just one, but also not all variables found is a good idea. This corresponds to $M_f = 2$.
- "2%": the pricing problems are sorted as for the default settings, but instead of adding just the first variable, we solve 2% of the pricing problems. The pricing round is stopped, if variables were added so far, otherwise, it is continued until the first variable is added. We obtain this behavior for $M_f = \infty$ and $R_f = 0.02$.
- "1 prob": the pricing problems are sorted as for the default settings, but instead of adding just the best solution of a pricing problem, we add all solutions of the problem that have a positive objective function

value. As soon as a pricing problem gave rise to new variables, the pricing round is stopped. This corresponds to $M_f = \infty$ and $R_f = 0$.

Table 4.1 summarizes the results, further details can be found in Tables D.1 to D.10 in Appendix D. The computational environment and the test sets are described in Section 3.3.

The upper and middle part of Table 4.1 picture the number of pricing rounds and solved pricing problems, respectively, the lower part the time that was needed to achieve feasibility of the RMP at the root node. We excluded the time needed for presolving, problem creation, primal heuristics etc. and list only the sum of pricing and LP solving time. The solving process was stopped after at most 5 minutes.

The first column shows—in absolute numbers—for each test set the shifted geometric mean of these values when performing Farkas pricing with default settings. The columns "all vars", "no sort.", "2 vars", "2%", and "1 prob", show the percental change in the shifted geometric mean for the variations compared to the default settings.

We used the convexification approach for these computations so the identical blocks in the vertex coloring and the bin packing problem where treated independently. Hence, for these problems, all pricing problems have equivalent optimal solutions, so the pricing problems with greater Farkas multipliers have better optimal objective values. The same holds for the capacitated p-median problem (CPMP): The only difference between the blocks is the objective function, which is disregarded for the Farkas pricing. For the resource allocation problems, obtaining feasibility is rather easy since setting all variables to zero in the original problem gives us a feasible solution. Therefore, for each of the pricing problems, the zero solution has to be found and the corresponding variable has to be added to the RMP in order to restore feasibility. Nevertheless, the presented results give us some indication for the impact of the different settings.

For the default settings (column "default"), we solved exactly one pricing problem per round, which was to be expected due to the similar structure of the pricing problems.

When adding all variables (column "all vars"), the shifted geometric mean of the number of pricing rounds is decreased by 68%. This was to be expected since we add more variables in each round, in particular, in each round we also add the variable that would be added when performing this pricing round with default settings. The additional variables potentially cut off other dual rays that would have caused an infeasibility in a subsequent iteration. In return, we have to solve all pricing problems in each pricing round, hence we solve about sixteen times as many pricing problems in spite of the considerably smaller number of pricing rounds. Because of this, the total running time increased more than sevenfold.

When not sorting the pricing problems (column "no sort."), the number of pricing rounds, the number of pricing problems that are solved, and the

	test set	default	all vars	no sort.	two vars	2 %	1 prob
rounds	CPMP50S	238.4	−60	+763	−25	−32	−32
	CPMP100S	571.3	−67	+1038	−30	−49	−41
	CPMP150S	1011.4	−77	+424	−35	−57	−48
	CPMP200S	1457.7	−82	+126	−38	−60	−52
	COLORING	241.1	−40	+829	−10	−8	−4
	RAP32S	37.7	0	0	0	0	0
	RAP64S	19.0	0	0	0	0	0
	BINDATA1-N1S	227.3	−73	+771	−24	−11	−4
	BINDATA1-N2S	545.6	−78	+684	−24	−37	−15
	BINDATA1-N3S	1463.0	−84	+237	−24	−56	−8
	mean	385.4	−68	+353	−23	−38	−25
problems	CPMP50S	238.4	+1895	+1938	+773	−32	−32
	CPMP100S	571.3	+3209	+3453	+1215	+3	−41
	CPMP150S	1011.4	+3428	+2315	+1553	+30	−48
	CPMP200S	1457.7	+3563	+1831	+1880	+60	−52
	COLORING	241.1	+1251	+1444	+163	+9	−4
	RAP32S	37.7	+3444	+1744	+3444	+23	0
	RAP64S	19.0	+1721	+885	+1721	0	0
	BINDATA1-N1S	227.3	+906	+6052	+319	−3	−4
	BINDATA1-N2S	545.6	+1517	+5372	+505	+27	−15
	BINDATA1-N3S	1463.0	+2273	+1618	+809	+57	−8
	mean	385.4	+1690	+1849	+745	+16	−25
time	CPMP50S	0.5	+891	+2603	+76	−30	−31
	CPMP100S	3.6	+4964	+7982	+95	−40	−58
	CPMP150S	15.2	+1964	+1854	+52	−31	−62
	CPMP200S	50.7	+501	+481	+21	−26	−72
	COLORING	3.9	+903	+5294	+117	+10	−3
	RAP32S	0.1	+193	+99	+194	+2	+1
	RAP64S	0.0	+79	+40	+80	0	0
	BINDATA1-N1S	0.2	+151	+10832	+40	−1	+1
	BINDATA1-N2S	1.1	+852	+27542	+177	−14	−23
	BINDATA1-N3S	8.5	+701	+3315	+134	−31	−1
	mean	5.2	+602	+1857	+63	−21	−44

Table 4.1. Performance effect of different variants of the Farkas pricing for obtaining feasibility of the RMP at the root node. The first column denotes the shifted geometric means of the number of pricing rounds (top), the number of pricing problems that were solved (middle) and the master problem solving time in second (bottom) for the default settings. The other columns denote the percental changes in the shifted geometric mean of the values for the other settings. Positive values represent a deterioration, negative values an improvement.

solving time of the master problem are increased by far. The increase of the number of pricing problems is consequential, since for the default settings, the pricing problem that is solved first has the best solutions due to the structure of the problems in our test set. Hence, we only solve one pricing problem per pricing round for the default settings. When not sorting the pricing problems, it can happen that we need to solve several pricing problems until we find a solution with positive objective function value. However, also the number of pricing rounds is more than quadrupled. This can be explained by the fact, that we add a variable with negative objective value in its pricing problem, but there also exists an equivalent variable in the pricing problem with the highest corresponding Farkas multiplier, which thus has a better objective function value and which is added in the default settings. This shows, that also for the Farkas pricing, adding variables that have a better objective function value in the pricing problem reduces the number of pricing rounds by far. A motivation for this is that a variable with better objective value in its pricing problem corresponds to a potentially tighter constraint in the dual of the RMP, at least, it only allows smaller multiples of the ray given by the dual Farkas multipliers than a solution with smaller objective function value.

Adding at most a given number of variables in each pricing round—we tried adding two variables in each round (column "two vars")—helps in decreasing the number of pricing rounds, but it has an essential drawback: If there only exists one variable that fulfills condition (4.10), we have to solve all the pricing problems. This results in an eightfold increase of the number of pricing problems that are solved, although the number of pricing rounds is decreased by 23% in the shifted geometric mean. Therefore, the shifted geometric mean of the total time is increased by 63%, too.

In order to overcome this drawback, we limit the number of pricing problems that are solved and not the number of variables added. Column "2%" shows the results for solving 2% of the pricing problems and adding all variables that are found during this process. If no variable was found so far, we continue the pricing round until variables are added, otherwise, we stop the pricing round. This way, we obtain a decrease in the shifted geometric mean of the number of pricing rounds of 38% with a moderate increase of only 16% in the number of pricing problems that are solved. The shifted geometric mean of the total time is even decreased by 21%.

Finally, we tried to add all variables of a pricing problem instead of only the first one (column "one prob"). This way, we still solve just one pricing problem per pricing round, but we add more variables and thus, we can reduce the number of pricing rounds by 25% and the total time by 44%. Hence, this variant seems to be superior to the default settings.

Finally, let us note that adding two variables led to one timeout, adding all variables resulted in 38 timeouts and without sorting the pricing problems, the time limit was reached by 95 out of 198 instances, so the results for these settings would have been even worse without imposing a time limit.

We draw the following conclusions that are valid at least for our test sets:

- Adding better variables, i.e., variables with higher objective function value in the pricing problems, significantly reduces the number of pricing rounds and the number of pricing problems that are solved as well as the total time.
- Although one variables suffices to cut of the dual ray, adding more variables per pricing round typically reduces the number of pricing rounds.
- However, a fixed limit greater than one—also infinity—of variables to add typically increases the number of pricing problems, that are solved and also the total time, since all problems have to be solved if this number is not exceeded. Imposing a limit on the number of pricing problems is therefore superior.

Aggregating blocks in the discretization approach

Until now, we only investigated the Farkas pricing strategies for the convexification approach. In the following, we give a comparison between convexification and discretization approach. When not aggregating blocks, the pricing process is exactly the same for both approaches. Hence, we only investigate the vertex coloring and the bin packing instances in this subsection, for which blocks are identical and can thus be aggregated. In Table 4.2, we picture the number of pricing rounds, the number of pricing problems that are solved and the solving time of the master problem for setting "1 prob", which outperformed the default settings for the convexification approach. Since we only have one pricing problem in the discretization approach, we do not have the variety of possible pricing strategies as in the convexification approach. We solve one—the only—pricing problem. We tried two possibilities of how many variables are added. On the one hand, we can add to the RMP just one variable corresponding to an optimal solution. This is done in setting "disc best". On the other hand, we can also add all variables corresponding to solutions with positive objective function value that were found during the solving process of the pricing problem. The results for this approach are listed in column "disc all".

Setting "disc all" resembles "1 prob" for the convexification approach, where we solve the pricing problem with the most positive dual Farkas multiplier and add all variables corresponding to solutions with positive objective function value, afterwards. Although this sounds similar, setting "disc all" is superior in terms of pricing rounds, pricing problems that are solved and total time needed for Farkas pricing at the root node. In the convexification approach, it may happen that a variable representing a point was created in one of the blocks, but a variable corresponding to the same point will be needed in another block later on. Then, in another pricing round, this variable has

	test set	1 prob	disc all	disc best
rounds	COLORING	232.4	–26	–24
	BINDATA1-N1S	219.1	–72	–71
	BINDATA1-N2S	462.1	–74	–72
	BINDATA1-N3S	1352.8	–83	–81
	mean	442.5	–69	–67
problems	COLORING	232.4	–26	–24
	BINDATA1-N1S	219.1	–72	–71
	BINDATA1-N2S	462.1	–74	–72
	BINDATA1-N3S	1352.8	–83	–81
	mean	442.5	–69	–67
time	COLORING	3.7	–24	–20
	BINDATA1-N1S	0.2	–15	–14
	BINDATA1-N2S	0.8	–73	–64
	BINDATA1-N3S	8.4	–95	–91
	mean	2.9	–73	–68

Table 4.2. Performance effect of the discretization approach for obtaining feasibility of the RMP at the root node. The first column denotes the shifted geometric means of the number of pricing rounds (top), the number of pricing problems that were solved (middle) and the runtime (bottom) for the convexification approach with setting "1 prob", The next columns denote the percental change in the geometric mean of these values when using the discretization approach and adding only the best variable (second column) or all variables that cut off the dual ray (third column). Positive values represent a deterioration, negative values an improvement.

to be created, too. This cannot happen in the discretization approach with aggregated blocks since we do not distinguish the blocks and only have one pricing problem. In some sense, the pricing in the discretization approach is similar to pricing in the convexification approach, thereby adding not just variables for one block, but the equivalent variables for all the other blocks, too. However, this leads to a much bigger RMP slowing down the simplex method and also the number of constraints in the RMP is slightly smaller for the discretization approach since we have just one convexity constraint.

Like for the convexification approach, adding just one variable per round is inferior to adding all variables with positive objective function value of one problem (which this time is the only one). Thus, we should think about changing the default setting from adding just one variable to adding all variables of one problem with positive objective function value for both approaches. However, we do not know whether adding all variables is also superior w. r. t. the branch-and-price process. We performed computations about this and answer this question in Section 7.1.

To sum up, the discretization approach outperforms the convexification approach in the case of identical blocks as it was to be expected. Especially for test sets BINDATA1-N2S and BINDATA1-N3S, where each instance has 100

and 200 identical blocks, respectively, feasibility of the RMP is obtained about four and twenty times faster, respectively.

4.4.2 Reduced Cost Pricing

In this subsection, we present some computational results for the reduced cost pricing. We just list the effort of the reduced cost pricing process of the master problem at the root node, so we excluded the effort for the Farkas pricing process. Furthermore, for the time listed in the tables, we also eliminated the time needed for presolving, problem creation, primal heuristics etc. and list only the time needed for the solving process of the master problem, i.e., the sum of pricing and LP solving time. We imposed a time limit of five minutes for each instance, except for the resource allocation problems where we set a time limit of thirty minutes as the solving process of the master problem takes much longer for these instances.

Again, we start with a comparison of some fundamental strategies. The default pricing routine adds at most 100 variables to the RMP in each pricing round ($M_r = 100$). The pricing process is stopped after this number of variables was added, so we set $onlybest = FALSE$. We do neither limit the number of variables created per pricing problem ($M_p = \infty$) nor the number of pricing problems that are solved in a single pricing round ($R_r = 1$). Table 4.3 summarizes the results of the computations. More details can be found in Tables D.11 to D.20.

A first observation is the considerably smaller number of pricing rounds for the default settings compared to Farkas pricing, which corresponds in parts to the fact that we add much more variables in each pricing round. In the shifted geometric mean, about 30 pricing problems are solved in each pricing round and the time needed for reduced cost pricing is about six times higher than the time needed for Farkas pricing. This shows, that the reduced cost pricing is much more important than the Farkas pricing for the overall performance. This holds even more for the branch-and-price process since at the subsequent nodes, we start with the set of variables that were created before so we have to perform just a few—if any—Farkas pricing rounds to repair an infeasibility caused by the branching restrictions.

Pricing strategy "only best" solves all pricing problems and adds the variable with the most negative reduced cost. This corresponds to the pricing process as we described it from the theoretical point of view in Section 4.2. We set $M_r = 1$ and $onlybest = TRUE$. This pricing method does not perform well: The number of pricing rounds, the number of pricing problems that are solved and the time needed for reduced cost pricing are increased by far. Actually, the values would have been even worse if we had not imposed a time limit: 101 out of 198 instances hit the time limit and were thus stopped before the master problem was solved to optimality. This is also the explanation for the decrease of the number of pricing rounds for test set bindata1-n3s. As conjectured in Section 4.3.2, adding more than just the

	test set	default	only best	all vars	100best	one prob
rounds	CPMP50S	21.7	+1319	−15	+3	+2245
	CPMP100S	35.8	+1761	−34	+5	+2890
	CPMP150S	51.3	+662	−46	+5	+2979
	CPMP200S	69.6	+118	−55	−4	+2160
	COLORING	40.5	+12	−21	−22	+21
	RAP32S	48.4	+332	−63	−51	+602
	RAP64S	35.3	+190	−56	−45	+324
	BINDATA1-N1S	34.2	+55	+5	+7	+7
	BINDATA1-N2S	79.1	+64	−2	+18	−6
	BINDATA1-N3S	132.6	−19	−26	−14	−14
	mean	52.2	+237	−32	−10	+511
problems	CPMP50S	753.3	+1916	+20	+42	+333
	CPMP100S	1539.0	+4214	+51	+136	+712
	CPMP150S	2515.1	+2210	+62	+212	+671
	CPMP200S	3534.6	+729	+73	+273	+402
	COLORING	487.3	+71	+24	+24	−83
	RAP32S	792.2	+831	−14	+12	+390
	RAP64S	321.9	+453	−6	+14	+159
	BINDATA1-N1S	937.4	+77	+22	+25	−92
	BINDATA1-N2S	3370.1	+166	+55	+89	−95
	BINDATA1-N3S	6902.7	+128	+98	+142	−96
	mean	1438.0	+551	+35	+81	+9
time	CPMP50S	1.9	+1548	+27	+49	+462
	CPMP100S	4.5	+3608	+84	+159	+788
	CPMP150S	9.3	+2920	+112	+237	+799
	CPMP200S	16.5	+1270	+129	+281	+622
	COLORING	34.1	+52	+23	+21	−82
	RAP32S	178.0	+684	−16	+4	+427
	RAP64S	289.7	+342	−10	+7	+198
	BINDATA1-N1S	6.8	+149	+25	+25	−90
	BINDATA1-N2S	51.1	+190	+64	+82	−88
	BINDATA1-N3S	106.7	+160	+138	+127	−90
	mean	33.8	+438	+37	+62	+47

Table 4.3. Performance effect of different variants of reduced cost pricing for solving the master problem to optimality at the root node. We list the shifted geometric means of the number of pricing rounds (top), the number of pricing problems that were solved (middle) and the runtime (bottom) for reduced cost pricing with default settings (first column). The other columns denote the percental changes in the shifted geometric mean of these values for the other settings. Positive values represent a deterioration, negative values an improvement.

variable with the most negative reduced cost speeds up the solving process by far.

Strategy "all vars" is the opposite: We solve all pricing problems and add all variables to the RMP that are found during the solving process and that have negative reduced costs. Therfor, we set $M_r = \infty$, $M_p = \infty$, and do not limit the number of pricing problems that are solved in each pricing round ($R_r = 1$). As for the Farkas pricing, this typically reduces the number of pricing rounds compared to the default setting, since more variables are added in each round. The number of pricing problems that are solved increases and so does the total time. However, for the sets of RAP instances, this setting performes better than the default setting. We will give an interpretation for this later, but it already shows, that there is a big variation on the reduced cost pricing performance for the different test sets.

We also tested whether adding not the first 100 variables that are found, but the 100 variables with the most negative reduced costs improves the performance (column "100best"). The only difference to the default settings is that we set $onlybest = TRUE$. This increases the number of pricing problems that are solved, since we have to solve all pricing problems in order to determine the 100 "best" variables. Hence, also the total time is increased by about 50%.

Finally, for pricing strategy "one prob", we perform the reduced cost pricing similar to the best setting for the Farkas pricing: We stop the pricing round as soon as a pricing problem has a negative optimal objective value and add all variables with negative reduced costs that were found during the solving process of this problem. The shifted geometric mean of the number of pricing rounds is increased by far, as it was to be expected since we add a smaller number of variables in each pricing round. This pricing method performs very well for the vertex coloring and the bin packing instances. In return, for the other instances, it performs badly.

We can see that due to the rather different structure of the problem classes, we get different results for the pricing methods that we investigate. This shows that the performance of a pricing strategy is always problem dependent and there does not exist a dominating pricing method. However, we can draw some conclusions based on the presented computational results. For the vertex coloring and the bin packing instances, pricing strategy "one prob" performs best. This can be explained by the identical blocks: As all problems are identical and only the dual solution values of the corresponding convexity constraints differ, the pricing problems are rather similar. Therefore, also the solutions that are found during the solution process of the pricing problems are also rather similar. Thus, spending more effort to solve more than one pricing problem leads to equivalent variables corresponding to other blocks that just slow down the simplex algorithm.

For the class of resource allocation problems, adding all the variables that are found and which have negative reduced cost in each pricing round seems to perform best and adding the 100 best variables is competitive to the

default settings. These problems have rather different blocks. Each block has the same structure, i.e., it consists of a number of capacity constraints, but the variables corresponding to the blocks are different, they have arbitrary objective function values and different coefficients in the constraints of the master problem. Hence, optimal solutions to different blocks are independent and differ in the master problem's coefficients, so each variable is important and can help during the solving process.

In contrast to that, for the capacitated p-median instances, the blocks are distinct, but still related. Each solution to a pricing problem is also valid for all other pricing problems and the corresponding variables have the same coefficients in the constraints of the master problem, they only differ in terms of the objective function coefficients. Therefore, adding more variables to the RMP in each round than just a few helps since it allows more sophisticated pivot rules in the simplex method and multiple simplex steps after one pricing round. Nevertheless, since the variables are somehow similar, adding all variables leads to many similar variables out of which only a few can be used, so it does not pay off.

Aggregating blocks in the discretization approach

Again, we investigate the effect of the discretization approach on the performance. We only regard the problems that have identical blocks in the following, since for the other problems, the pricing process is the same for convexification and discretization approach. For the regarded problems, pricing strategy "one prob" performed best for the convexification approach: The master problem was solved five to ten times faster with it than with the default settings. Therefore, we take the performance for this strategy as the reference value and compare it to the discretization approach in Table 4.4. More details can be looked up in Tables D.5 and D.8 to D.10. In particular, we regard two different strategies for the pricing routine: "disc all", where we add all variables to the RMP that were found during the pricing process and that have negative reduced costs and "disc best", where only one variable with the most negative reduced cost is added to the RMP.

The first observation is that the number of pricing rounds needed for reduced cost pricing is higher for the discretization approach. However, this is not an attribute of the reduced cost pricing routines, but it arises from the different Farkas pricing methods used for the convexification and the discretization approach. The default Farkas pricing method for the convexification approach takes longer than its counterpart for the discretization approach, but it also creates more variables. Due to the smaller number of variables contained in the RMP at the beginning of the reduced cost pricing process, more variables are created afterwards for the discretization approach and so the number of pricing rounds is higher, too.

However, the number of pricing problems that are solved in the reduced cost pricing process is slightly reduced when using the discretization ap-

	test set	one prob	disc all	disc best
rounds	COLORING	49.0	+43	+61
	BINDATA1-N1S	36.7	+102	+165
	BINDATA1-N2S	74.3	+102	+200
	BINDATA1-N3S	114.0	+149	+312
	mean	66.0	+98	+178
problems	COLORING	82.8	−14	−3
	BINDATA1-N1S	77.1	−3	+27
	BINDATA1-N2S	152.3	−1	+47
	BINDATA1-N3S	276.5	+3	+70
	mean	135.5	−3	+36
time	COLORING	6.0	+5	+22
	BINDATA1-N1S	0.7	−26	−5
	BINDATA1-N2S	6.1	−63	−33
	BINDATA1-N3S	11.1	−40	+14
	mean	5.5	−34	+1
total time	COLORING	12.2	−7	+2
	BINDATA1-N1S	1.1	−37	−18
	BINDATA1-N2S	8.6	−59	−37
	BINDATA1-N3S	31.3	−50	−30
	mean	10.8	−38	−21

Table 4.4. Performance effect of different variants of the reduced cost pricing for solving the master problem to optimality at the root node. We list the shifted geometric means of the number of pricing rounds (top), the number of pricing problems that were solved (second part) and the time needed for reduced cost pricing (third part) and the total runtime (bottom) for reduced cost pricing with setting "one prob" (first column). The next columns denote the percental changes in the shifted geometric mean of these values for the discretization approach and adding all variables with neg. reduced cost (second column) or adding only one variable with the most negative reduced cost (third column). Positive values represent a deterioration, negative values an improvement.

proach and adding all variables. In case we only add the best variable, it is increased by a third. The better relation between pricing problems and pricing rounds in the discretization approach can be explained by the fact, that in each pricing round, one—the only—pricing problem is solved, while in the convexification approach, all pricing problems are solved in the last pricing round when no variable is found.

The shifted geometric mean of the pricing time is reduced by 34% when adding all variables with negative reduced cost. For test set COLORING, it increases by 5%, which can be explained by the smaller number of variables created in the Farkas pricing process before. Therefore, we also list the total time, including presolving, problem creation and Farkas pricing. Now, the discretization approach with setting "disc all" is superior to the convexification approach for all test sets.

Setting "disc best" performs better than the convexification approach, too, but it is inferior to setting "disc all". Hence, it pays off to add more variables with negative reduced costs than just the "best" one, also for the discretization approach. Since solving the pricing problems consumes most of the time needed for pricing, we should therefore add all variables with negative reduced costs to the RMP that are found during the pricing process, hoping that this reduces the number of pricing rounds and thus also the number of pricing problems that need to be solved.

Chapter 5

Branching

In Chapter 4, we have described how we can solve the master problem to get a lower bound on the optimal objective value of the original problem (Model 2.1). The optimal solution to the master problem that is computed this way fulfills all constraints in the extended problem, except for the integrality restrictions. Transferring this solution to the original variable space gives rise to a solution of the original problem that does not neccessarily fulfill the integrality restrictions (2.4), but all the other constraints contained in the original problem. If it additionally fulfills the integrality restrictions, we have solved the original problem to optimality. Otherwise, we use a branch-and-bound process to obtain integrality.

The special thing about branch-and-bound in our approach is the fact, that we do not solve one problem, but two equivalent problems simultaneously: the original problem and the extended problem. This can be interpreted in two ways, as we explain in the following.

Let us start, however, with the connection between integrality in the original problem and integrality in the extended problem. For the convexification approach, the connection is clear. Since the integrality in the extended problem is defined in terms of the original variables, a solution to the extended problem is integral if and only if its counterpart for the original problem is so, too. Therefore, when using the convexification approach, it is natural to perform branching in terms of the original variables, but this is also possible for the discretization approach. How branching can be performed in terms of the original variables and how the branching decisions influence the extended formulation is described in Section 5.1.

This leads us to the first interpretation of the branch-and-bound solving process: what we do is solving the original problem with a branch-and-bound method. In contrast to the LP-based branch-and-bound algorithm used by most state-of-the-art MIP solvers like `Cplex`, `Gurobi`, `CBC` or `SCIP`, we do not solve the standard LP relaxation of the original problem at each node of the branch-and-bound tree. In order to compute a lower bound and a solution that fulfills all but the integrality constraints, we use the Dantzig-Wolfe decomposition at each node. We reformulate the current problem,

obtain a master problem that is then solved and that provides a lower bound for that node. The solution of the master problem can then be transferred to a solution of the original problem that does not neccessarily fulfill the integrality restrictions. It can be used in the same way as the LP solution: It is checked for integrality and guides the branching decisions. We can say, we replaced the LP relaxation by a special relaxation that solves the master problem.

For the discretization approach, however, we also obtain integrality of the original solution by enforcing integrality of the solution to the master problem. Hence, we can in this case also perform branching in terms of the variables of the master problem. Branching rules that work in this way are presented in Sections 5.2, 5.4 and 5.5. In fact, we do not need do consider the original problem anymore during the solving process, we just solve the extended problem to optimality and transfer the optimal solution of the extended problem to an optimal solution of the original problem.

This is the second interpretation: at the beginning of the solving process, we transform the original problem into an equivalent problem, the extended problem. This problem is solved with a branch-and-price approach, the computed optimal solution is then retransformed into the original variable space and gives us an optimal solution to the original problem. Strictly speaking, one could say that we only reformulated the problem and solved this reformulation.

It is not possible to draw a distinct line between the two interpretations, since most branching decision can be expressed in both formulations. Branching decisions in the original problem have to be expressed in terms of the variables of the extended problem and added to it. This must be done to assure that the master problem gives rise to an original solution that fulfills all but the integrality constraints, thus in particular also the branching restrictions. Nevertheless, most of the branching decisions are derived from either the original or the master problem and then transfered to the other problem. Hence, they typically have the more natural expression in one of the problems.

However, we will still consider the original problem, even if we derive the branching decisions from the extended problem and use the additional information provided by the original problem to speed up the solving process. For example, we do not always need to solve the extended problem to optimality. Since our prior target is to solve the original problem to optimality, we do not insist on achieving an optimal solution to the extended formulation. As fractional solutions to the extended problem can lead to an integral solution of the original problem—a nontrivial combination of extreme points can for instance represent an interior point $x^k \in X_k$ like in the convexification approach—we consider a problem to be solved once the solution to the master problem transfers to an integral solution of the original problem, even if the master solution is not integral itself.

Finally, let us clarify, that the variables added to the RMP at the root

node do not suffice for the further solution process. In order to solve the master problem of a node, we always have to perform pricing and add variables to the RMP that can improve the current solution or fix an infeasibility arising from the branching restrictions.

5.1 Branching on Original Variables

As described above, the solving method can be interpreted as a branch-and-bound process on the original formulation that uses the master problem as a relaxation rather than the standard LP relaxation. In this context, it would be natural to perform branching in terms of the original variables. Most state-of-the-art MIP solvers use a branching on variables in the branch-and-bound process, e.g., *most infeasible branching*, *pseudocost branching*, *strong branching*, or *reliability branching* (see [5]). In this section, we describe how branching on variables of the original problem can be transferred to the Dantzig-Wolfe decomposition approach presented in this thesis. This branching scheme is widely proposed in the lierature, see for instance [47, 10, 29].

We start with the application to the convexification approach, since this is the natural branching scheme for this approach. After that, we describe how this branching scheme can be used in the discretization approach, too.

After solving the master problem to optimality, we get a solution $\bar{x}$ of the original problem according to the coupling constraints (2.13). The integrality restrictions in the extended problem are expressed in terms of the original variables, so if they are fulfilled, $\bar{x}$ is an optimal solution of the current original problem and we do not need to perform branching at the current node. Otherwise, there exists a variable $x_i^k, k \in [K], i \in [n_k^\star]$ of the original problem with fractional value $\bar{x}_i^k$.

Branching on this variable means creating two children, one with an additional constraint

$$x_i^k \leq \lfloor \bar{x}_i^k \rfloor \tag{5.1}$$

which we call the *down-branch*, and the *up-branch* with the additional constraint

$$x_i^k \geq \lceil \bar{x}_i^k \rceil. \tag{5.2}$$

Ways to Enforce the Branching Decision

We have two possibilities, where to add the branching constraints (5.1) and (5.2) in the down-branch and up-branch, respectively. As a branching constraint contains variables of just one block, namely exactly one variable x_i^k, we can add it to the structural constraints (2.2) of this block. On the other hand, we can also interpret the branching constraint as a part of the linking constraints (2.1) and add it to this part of the constraints.

In both cases, we get a modified original problem corresponding to the newly created child node. We can again apply the Dantzig-Wolfe decomposition for MIPs on this problem, like described in Chapter 2, in order to get an extended formulation as well as a master problem that both consider the branching decision. In fact, we do not need to do the reformulation from scratch, we can adapt the current extended formulation and master problem to the new original problem.

In the following, we describe how this can be done and which effect it has on the pricing subproblems. We do this on the example of the up-branch, the down-branch is handled similar.

Adding Branching Constraints to the Linking Constraints On the one hand, we can add the branching constraint (5.2) to the linking constraints (2.1). Hence, we get an additional constraint

$$\sum_{p \in P_k} p_i \lambda_p^k + \sum_{r \in R_k} r_i \lambda_r^k \geq \lceil \bar{x}_i^k \rceil$$

in the extended formulation and the master problem. For this constraint, we get an additional dual variable σ which has to be respected in the pricing routine. It is simply subtracted from the objective value of x_i^k in the pricing problem so that the pricing problem changes to the following:

$$\bar{c}_k^\star = \min \left\{ \left({c_k}^T - \pi^T A^k - e_i \sigma \right) x - \rho_k \mid x \in X_k \right\}.$$

Thus, the reduced costs of the potential variables change, but any method that was used before to solve the pricing problems can still be used since the structure of the problem stays the same. The sets of variables corresponding to each block stay the same, in particular, all variables that we created so far are still valid.

Adding Branching Constraints to the Structural Constraints On the other hand, if we add the branching constraint (5.2) to the structural constraints (2.2), the set X_k is changed and so are the sets P_k and R_k of variables for this block. In the pricing subproblem for block k, the constraint (5.2) is added. When the pricing problems are solved by a MIP solver, this constraint can be handled easily, but if we solve the pricing problems with a special purpose solver, this solver has to be able to take into account upper and lower bounds on the variables. This is, however, the case in most of the applications: In the bin packing problem or the capacitated p-median problem, where the pricing problem is a knapsack problem, we remove items in the preprocessing, that have a fixed value and subtract the sizes of the items that have value fixed to 1 from the total capacity. For the vertex coloring problem, the pricing problem is a stable set. Forcing a node to be part of the stable set corresponds to removing all nodes from the problem that are

adjacent to it. If a variable is fixed to 0, the corresponding node can simply be removed.

In addition to that, we have to make sure, that we only use variables corresponding to points and rays that fulfill the branching constraint. If we do not start with an empty set of variables at each node, but use variables that were created before at other nodes, we have to check these variables for their feasibility w. r. t. the new pricing problem. If they do not fulfill the new constraint, they are not allowed to be used in any solution of the master LP at the current node and thus, they have to be removed or forbidden for the current node.

Except for the modified sets P_k and R_k, the extended formulation and the master problem stay the same, in particular, the number of constraints stays the same, so we have the same dual variables as before and the calculation of the objective function coefficients of the variables in the pricing subproblems does not change.

Comparison Adding the constraints to the structural constraints usually results in a tighter lower bound than adding it to the linking constraints, since the latter only cuts off points that do not satisfy the new constraint while the former also forbids points that fulfill the constraint but do no longer lie in the convex hull of the points that fulfill both the structural constraints—including the new one—as well as the integrality restrictions.

The better lower bound is achieved at the cost of a changed structure of the subproblem, possibly preventing the usage of a special purpose solver and the effort that has to be spent to assure the feasibility of all variables and to fix variables that do not fulfill the new constraint.

For the convexification approach, this branching scheme is the most natural one as the integrality restrictions of the extended problem are formulated in terms of the original variables, anyway. Nevertheless, it can also be applied when using the dicretization approach.

Application to the Discretization Approach

For the discretization approach, when adding the branching decisions to the structural constraints, the pricing subproblems are modified. This way, we handle the problem of variables related to interior points: for an optimal solution to the extended problem, we probably need variables corresponding to interior points of the pricing polyhedra. The pricing routine, however, does only find variables corresponding to extreme points or extreme rays of the current polyhedra. By modifying the pricing polyhedra according to the branching decisions, each initially interior point will become an extreme point after a number of branching constraints are added to the subproblems.

Nevertheless, adding the branching decisions to the linking constraints is also a valid branching scheme that allows us to compute the optimal integer solution to the original problem. However, since we create no variables corre-

sponding to interior points this way, we can typically not achieve an optimal solution to the extended problem since we miss out variables that are needed for this. For the master problem, these variables are redundant since they can be expressed as a combination of extreme points and extreme rays, so we will get fractional solutions to the master problem that correspond to interior points and translate into integer solutions of the original problem. Therefore, the branching restrictions suffice to ensure integrality in the original problem and we do not need an optimal solution to the extended problem.

If we use the discretization approach and have identical blocks, branching on variables corresponding to one of these blocks means modifying the block so that it is no more identical to the others. We have to remove the modified block from the class of identical blocks and treat it separately, thereby reintroducing symmetry.

In the case of identical blocks, in both the convexification as well as the discretization approach, branching on the original variables leads to the same symmetry as if we would solve the original problem with an LP-based branch-and-bound approach: After bounding some variable in one of the identical blocks, we usually get an equivalent solution that only permutes the points chosen in the identical blocks.

One of the main advantages of the Dantzig-Wolfe decomposition is that we can overcome symmetry by using the discretization approach and aggregating identical blocks. Additionally, to achieve integrality, we need a branching scheme that changes all of the blocks in the same way, so that the identity of the blocks is preserved during the branch-and-bound process. We will present branching schemes that fulfill these conditions in the next sections.

For a long time, symmetries in integer programs where typically handled this way: an alternative formulation—which is often exactly the extended formulation obtained by the Dantzig-Wolfe decomposition—was proposed, usually together with a problem specific branching scheme. Alternatively, problem specific symmetry breaking constraints were used, see for example [44, 24, 25].

In the recent years, general problem independent methods were proposed that handle symmetries, e.g., *isomorphism pruning* [61], *orbital branching* [70, 71], *orbitopal fixing* [49], and *shifted column inequlities* [50]. Using these methods would allow to break symmetry even when using the convexification approach or branching on original variables. In this thesis, however, we do not go into detail about these methods and restrict ourselves to the discretization approach together with the branching rules presented in the next sections in order to overcome symmetry.

5.2 Branching on Variables of the Extended Problem

The discretization approach does not distinguish between identical blocks, so it can be used to break symmetry. However, as described in the last section, branching on original variables reintroduces the symmetry so it is rather inefficient for problems with identical blocks. Hence, we need a branching scheme that conserves the identity of the blocks. Since we have integrality restrictions on the λ variables for the discretization approach, the most natural alternative would be to branch on variables of the extended problem.

Branching on these variables was applied to branch-and-price algorithms for instance in [73, 89]; it has, however, two major disadvantages regarding the structure of these problems (see [91, 10]):

- First, it generally leads to an unbalanced branch-and-bound tree. The master problem has a huge number of variables but only a very small part of them will have a strictly positive value in an optimal solution, so most of the variables will be zero. When imposing a new upper bound on a variable—fixing the variable to 0 in the binary case—this has in most cases no big effect, since this variable is likely to have value 0 in an optimal solution, anyway. Introducing a new lower bound on a variable—fixing it to 1 in the binary case—has much more effect since one of the small set of variables that have a strictly positive value is determined. Therefore, we get a very unbalanced branch-and-bound tree by branching this way.

- The second disadvantage arises when the master problem is solved after imposing a new upper bound on one of the variables. In the pricing problem, we have to forbid the point or ray represented by this variable. Otherwise, the same variable without tightened upper bound could be created again which would allow to restore the old solution, the variable that was bounded for branching would get redundant and the problem would actually not change. In order to exclude the point or ray corresponding to the bounded variable from the pricing problem, we can use *bound disjunction constraints* (see [1, Section 11.2.4]). These are nonlinear constraints, however, `SCIP` incorporates this kind of constraints. If we want to preserve the MIP-nature of the pricing problem, we can linearize the bound disjunction constraints: for binary variables, this can be done by adding a single linear constraint to the pricing problem, for general MIPs, however, it is much more complicated. How it can be done is described in [1, Section 11.2.4].

  Furthermore, if the pricing problem is solved by a special purpose solver, i.e., by a knapsack solver for the bin packing problem, excluding solutions often means looking for the k-th best solution if the $(k-1)$ first solutions are associated to points and rays that must not be regen-

erated. Typically, this is computationally harder than finding only one optimal solution and might destroy the structure of the pricing problem with the consequence that the special purpose solver is not capable of solving the problem anymore.

Due to these drawbacks, we do not consider branching on the variables of the master problem and need one of the more sophisticated branching schemes presented in the next sections.

5.3 Branching on Aggregated Variables

When solving an integer problem with identical blocks and using the aggregated extended formulation for the discretization approach (Model 2.10), we need a branching scheme that modifies all pricing problems of one identity class in the same way.

Instead of branching on a single original variable like proposed in Section 5.1, one possibility is to branch on a sum of original variables (see [95]). This sum has to be defined in a way such that the identity of the blocks is maintained. Therefor, we sum up the same variable over all blocks of one class K_ℓ and get aggregated variables y^ℓ which are defined as

$$y^\ell = \sum_{k \in K_\ell} x^k \tag{5.3}$$

for each class of identical blocks K_ℓ, $\ell \in [L]$.

Each integral solution of the original problem then leads to integral values of the aggregated variables, but the reversal does not hold: a fractional solution can also give rise to integral values of the aggregated variables. Nevertheless, if the aggregated variables have fractional values, we can branch on them to enforce their integrality even if this does not guarantee to obtain integral values of the single variables.

Let the current solution of the master problem translate into a solution $\bar{x}^k$, $k \in [K]$ of the original problem. If the aggregated solution value $\bar{y}_i^\ell = \sum_{k \in K_\ell} \bar{x}_i^k$ is fractional for some $\ell \in [L], i \in [n_\ell]$, we branch on the aggregated variable by creating two branch-and-bound nodes and adding to these nodes the branching constraints

$$\sum_{k \in K_\ell} x_i^k \geq \lceil \bar{y}_i^\ell \rceil \text{ and } \sum_{k \in K_\ell} x_i^k \leq \lfloor \bar{y}_i^\ell \rfloor. \tag{5.4}$$

We get a new linking constraint in the original problem and a corresponding constraint in the extended problem and the master problem. The additional dual variable can easily be respected in the objective function of the pricing problem, we simply handle it like the dual variables associated with the other linking constraints. The blocks are not changed and so are

the pricing problems, too. Note that the branching restriction cannot be enforced in the pricing problems since it affects multiple problems.

This is a basic approach for enforcing integrality that has the advantage that it maintains the identity of blocks and the structure of the pricing problems. However, as described above, it is often not sufficient in order to enforce integrality of the original variables. In the bin packing problem, for example, the original problem contains for each item a constraint enforcing that this item is assigned to exactly one bin. Hence, the aggregate variables for the items always have value 1, which can, however, still result in fractional values for the original variables that assign an item to an individual bin.

Nevertheless, there are also problems that can be formulated in a way such that enforcing integrality of the aggregate variables suffices to ensure integrality of the original variables, e.g., the vehicle routing problem (see [95, Proposition 1]).

And even if this is not the case for a special problem, due to its simplicity, this branching scheme can be used to eliminate the first fractional solutions until a more sophisticated branching scheme is needed that might change the structure of the pricing problems.

5.4 Ryan and Foster Branching

In the 1980s, Ryan and Foster proposed a branching scheme for problems with a set partitioning structure [82]. This branching scheme was successfully applied to the many problems with identical blocks that were aggregated, using the discretization approach (see [90, 92]). It is applicable to extended formulations of BPs for which all blocks are identical, i.e., $L = \{1\}$, $K_1 = K$. For ease of presentation, we define $X := X^1$, $P := P^1$, and $\lambda_p := \lambda_p^1$. Since the problem is binary, there do not exist rays in the pricing problem and we thus do not have variables in the extended problem corresponding to rays.

Furthermore, the problem must have a set partitioning structure, that means constraints (2.27) in the extended formulation (Model 2.10) must have the form

$$\sum_{p \in P} p\lambda_p = 1. \tag{5.5}$$

This property is always given if the original problem (Model 2.1) is a BP and has a set partitioning structure with coefficients given by identity matrices for each variable vector x^k, i.e.,

$$\sum_{k \in [K]} I_{n_k} x^k = 1. \tag{5.6}$$

Many practical applications have constraints of type (5.6) in the original formulation and therefore also the appropriate structure in the extended formulation, e.g., the bin packing problem (see Section C.1), the vertex coloring problem (see Section C.2), and the vehicle routing problem [88].

Ryan and Foster observed, that for a feasible integer solution to a problem with set partitioning structure, the following holds: Each constraint contains one variable with value one in the solution, all other variables contained in that constraint have solution value zero. We say that the variable with solution value one fulfills the constraint. When regarding two distinct constraints, they are either fulfilled by the same variable, i.e., a variable that is contained in both of the constraints, or by different variables. In this case, one variable has solution value one that is contained in the first constraint but not in the second one and one variable has value one that is contained in the second, but not in the first one.

Furthermore, a solution that fulfills constraints (5.5) is integral if and only if for each pair of two constraints, the sum of the values of variables contained in both constraints is integral. Of course, this only holds if all variables are pairwise different, i.e., no two variables have the same coefficients in all of the constraints (5.5). This is the case for the Dantzig-Wolfe approach presented in this thesis since in the master problem, we have exactly one variable for each point in the set X.

So for each solution of the aggregated master problem (Model 2.14) that is not integral, there exist two constraints $i, j \in [m_A]$ such that the sum of the values of variables that cover both constraints is fractional, i.e.,

$$0 < \sum_{\substack{p \in P: \\ p_i = p_j = 1}} \lambda_p < 1. \tag{5.7}$$

We can now branch on this constraint, forcing

$$\sum_{\substack{p \in P: \\ p_i = p_j = 1}} \lambda_p \leq 0 \tag{5.8}$$

in one child, and

$$\sum_{\substack{p \in P: \\ p_i = p_j = 1}} \lambda_p \geq 1 \tag{5.9}$$

in the other child. Due to the set partitioning constraints, (5.9) is equivalent to

$$\sum_{\substack{p \in P: \\ p_i \neq p_j}} \lambda_p \leq 0. \tag{5.10}$$

We call the child where we add constraint (5.8) the *DIFFER-child*, since by fixing all variables to zero that cover both constraints i and j, they will be fulfilled by two *different* variables with solution value one. To the other child, we add constraint (5.10) which fixes all variables to zero that cover exactly

one of the two constraints. So, both constraints have to be fulfilled by the *same* variable with solution value one and we call this child the *SAME-child*.

Since the problem has a finite number of constraints, there exist only finitely many combinations of two constraints and hence, the solving process is finished after a finite number of branch-and-bound nodes.

Each variable in the master problem corresponds to a point $x \in X$, i.e., a binary vector that fulfills the structural and the integrality constraints of the blocks. Removing variables from the master problem means that these points are excluded in the compact formulation from the sets X_k, $k \in [K]$ and thus forbidden in the pricing problem.

In Section 5.2, we stated that it is difficult to forbid the recreation of a single master variable in the pricing process and that this can destroy the structure of the pricing problems. This time, we exclude a set of variables in the pricing problem and only allow variables with a specific attribute that can easily be expressed as a linear constraint in terms of the variables of the original problem.

For the SAME-child, we add the constraint

$$x_i^k = x_j^k \tag{5.11}$$

to all blocks $k \in [K]$. This assures that in none of the blocks, a solution is chosen that fulfills only one of the two constraints i and j. In the DIFFER-child, we add the linear constraint

$$x_i^k + x_j^k \leq 1 \tag{5.12}$$

to all blocks $k \in [K]$ which forbids that a solution fulfills both constraints i and j of block k.

These constraints should be treated as structural constraint and added to the subproblems corresponding to the blocks.

Adding them to the linking constraints would corrupt the identity of the blocks, since for all blocks, we add one constraint, that contains only variables of this block and the blocks would not have the same coefficients in all linking constraints, anymore.

In addition to the linear constraint added to the pricing problem, we also have to remove variables form the RMP that were created before and do not fulfill the branching restrictions.

Let us mention, that although it is only one additional linear constraint that is added to the pricing problem, this additional constraint can destroy the structure of the subproblem that is needed for a special purpose solver. In the bin packing problem, for instance, constraints of type (5.11) can be handled by a general knapsack solver by "merging" items i and j. However, constraint (5.12) cannot be respected by the solver directly. One possibility in this case is to solve two problems, one without item i, the other without item j, but this gets costly when multiple branching constraints of this type have to be enforced. For example, for n active branching decisions of

type (5.12), we have to solve up to 2^n times the same pricing problem with slight modifications.

In contrast to that, for the vertex coloring problem where the pricing problem is to find a stable set in a graph with maximum weight, both types of branching restrictions can be handled without changing the subproblem's structure. Constraints of type (5.12) can be enforced by adding an edge between nodes i and j while constraints of type (5.11) are imposed by contracting the two nodes i and j (see [63]).

Finally, let us remark, that the Ryan and Foster branching scheme can also be applied to some problems with a setpacking ($\sum_{p \in P} p\lambda_p \leq 1$) or a setcovering ($\sum_{p \in P} p\lambda_p \geq 1$) extended formulation, if the objective function assures that there exists at least one optimal solution that fulfills all these constraints with equality.

One possible formulation for the vertex coloring problem, for example, is a setcovering formulation that requires that each node gets at least one color. In an optimal solution, more than one color can then be assigned to to a node, but choosing exactly one of these possible colors for the node leads to a feasible coloring of the graph. Therefore, it is clear, that there always exists an optimal solution that fulfills the inequalities with equality. Thus, the Ryan and Foster branching scheme can be applied to the vertex coloring problem, even when using set covering constraints like it is done in [63]. We also used set covering constraints instead of set partitioning constraints when modeling the problems of our bin packing and vertex coloring test sets. Since each solution of these formulations can easily be transferred to a solution of the set partitioning formulations with the same objective function value (see Appendix C), the Ryan and Foster branching scheme can be used for both problems, nevertheless.

5.5 Other Branching Rules

We have mentioned several possibilities to perform branching when we use the discretization approach with identical blocks.

The first option is branching on the original variables (Section 5.1). This leads to additional constraints in the master problem or minor changes in the pricing problems—only lower and upper bounds of variables are changed. Nevertheless, it is in general not the method of choice since it destroys the identity of the blocks so that we can not treat them together and therefore reintroduces symmetry. Breaking this symmetry was one of the main reasons for using the discretization approach, so we would like to conserve the identity of the blocks.

Alternatively, we can branch on the variables of the master problem (Section 5.2), but this leads to a highly unbalanced branch-and-bound tree and to serious changes in the structure of the pricing problems.

Branching on the aggregate variables (Section 5.3) is a simple scheme

that does not change the pricing problem, but it is no complete branching scheme so we cannot guarantee to eliminate all fractional solutions using it.

The Ryan and Foster branching scheme (Section 5.4) is only adaptable to binary problems with a set partitioning structure in the extended problem (or set packing / covering structure if the objective function guarantees that an optimal solution exists which fulfills the restrictions with equality). The branching scheme is complete, i.e., each fractional solution can be eliminated using it, and it leads to moderate changes in the pricing subproblem: we only have to add one additional linear constraint.

In this section, we describe further branching schemes proposed in the literature that can be applied to general bounded integer programs, do not reintroduce symmetry, and are complete branching schemes.

First, we give an alternative interpretation of Ryan and Foster's branching scheme. Ryan and Fosters scheme regards a subset of the variables, namely all variables that have value 1 in constraint i as well as in constraint j. This corresponds to all variables representing a point $p \in P_\ell$ with $p_i = p_j = 1$. So it can be interpreted as defining a subset $\tilde{P}_\ell = \{p \in P_\ell \mid p_i = p_j = 1\}$ of the points of this class of blocks and enforcing integrality of the sum of the values of variables in this subset.

Vanderbeck [92] proposes to derive branching schemes in a similar way for general IPs. For a block $\ell \subseteq [L]$ of a general (bounded) integer program, we define a subset $\tilde{P}_\ell \subseteq P_\ell$ of the points corresponding to that block. Now, we can enforce

$$\sum_{p \in \tilde{P}_\ell} \tilde{\lambda}_p^\ell \in \mathbb{Z}.$$

Hence, if this sum has a fractional value v in the current solution, we create two children and add branching constraints of the form

$$\sum_{p \in \tilde{P}_\ell} \tilde{\lambda}_p^\ell \geq \lceil v \rceil \text{ and } \sum_{p \in \tilde{P}_\ell} \tilde{\lambda}_p^\ell \leq \lfloor v \rfloor. \tag{5.13}$$

Vanderbeck [92] specifies several ways to define the subsets $\tilde{P}_\ell$, including a partition based on a hyperplane and a partition based on a set of bounds on the components of the points p.

The latter was developed before by Vanderbeck and Wolsey [97] and Barnhart et al. [10] and can be interpreted as a generalization of Ryan and Foster's branching scheme. We will only give a short survey here, for a more detailed review, we refer to [97, 10].

For each solution of the master problem, if one variable corresponding to block $\ell \in [L]$ has fractional value, then there exists a subset $\tilde{P}_\ell \subseteq P_\ell$ of the points of this block, that is defined by lower bound constraints and that corresponds to variables with a fractional aggregated value (see [97]). By creating two children for the current node and enforcing the constraints (5.13)

in the two children, respectively, we forbid the current fractional solution in the master problem.

The branching constraints of type (5.13) can, however, in general not be enforced in the pricing subproblems and have to be added to the master problem. This leads to an additional dual variable ν which has to be taken into account in the pricing subproblems. Since we cannot formulate an equivalent linear constraint in the original variable space, this dual variable cannot be considered directly in the objective function of the corresponding pricing problem. Therefore, we have to introduce an additional binary variable y with objective function coefficient $-\nu$ that gets value one in a solution if and only if the solution fulfills all the bound constraints. This is ensured by introcucing an additional binary variables z_i for each bound constraint that defines the subset $\tilde{P}_\ell$.

Hence, sets defined by a small number of bound constraints cause less modifications to the pricing problems and are used in priority. Empirically, they also lead to a more balanced branch-and-bound tree.

Nevertheless, this branching scheme can yield to serious modifications to the pricing problems which can make them much harder to solve and which typically cannot be handled by a special purpose solver.

This drawback is avoided by another branching scheme proposed by Vanderbeck [95]. In return, this one can lead to a non-binary branch-and-bound tree and for each class of identical blocks, multiple pricing problems have to be solved. However, the number of pricing problems to solve for a class K_ℓ of blocks is limited by $|K_\ell|$, so we do not solve more pricing problems in one pricing round than we would solve in the convexification approach or after having branched some times on original variables. We will only present the general idea in the following, for a detailed description, we refer to [95].

This scheme also bases on the definition of subsets $\tilde{P}_\ell$ by bound constraints and the enforcement of an integral value of variables corresponding to these subsets. However, a maximum value of the sum of all variables corresponding to $\tilde{P}_\ell$ is not enforced directly but a minimal value of the sum of variables in the complement $\bar{P}_\ell = P_\ell \setminus \tilde{P}_\ell$ is enforced.

It is not always possible to describe this complement $\bar{P}_\ell$ by bound constraints, but it can be described as the union of sets that can be described by bound constraints, say $\bar{P}_\ell = \bar{P}^i_\ell, i \in I$. With each of these sets as well as the set $\tilde{P}_\ell$, a pricing problem is associated. We do not get a binary branch-and-bound tree since for enforcing a lower bound in the complement, we need to distinguish different cases. For example, when we force that at least one point of the complement should be chosen, then for each of the problems $\bar{P}^i_\ell$, we create a subproblem in which we require, that at least one variable of this set is chosen.

5.6 Implementation Details

In this section, we describe how branching rules can be integrated into the branch-cut-and-price solver GCG. Furthermore, we survey the functions they can provide in addition to the functions of a standard branching rule in SCIP, which make it possible to define branching rules that use the information of the two alternative formulations.

In the current version of GCG, we included two of the branching rules that were presented in the last sections, the branching on original variables (Section 5.1) and Ryan and Foster's branching scheme (Section 5.4). In Sections 5.6.1 and 5.6.2, we give some details about the two integrated rules and the parameters that can be used to tune them.

As mentioned in Chapter 3, we use in our implementation two SCIP instances, one representing the original problem, the other representing the extended problem. Both problems are solved simultaneously and for each node in the former instance there exists a corresponding node in the latter SCIP instance. In the following, we identify these two nodes, so that the current node always corresponds to a subproblem of the original problem (in the original SCIP instance) as well as to a subproblem of the extended problem (in the extended SCIP instance). For the current node, we thus have the original subproblem, the extended subproblem, and the current master problem that are used for the solving process.

In order to obtain correct dual bounds, it is important, that all branching restrictions are enforced in the extended problem. Thus, each restrictions that a branching rule adds to the original problem at a node has to be enforced by it in the extended problem of that node, too. This assures, that by transferring the current master problem's solution into a solution to the original problem, this transferred solution fulfills all but the integrality constraints of the current original problem. On the other hand, if the branching rule adds restrictions to the extended problem, it does not necessarily need to transfer these restrictions and to add them explicitly to the original problem. As the restrictions are respected in the master problem, the solution obtained by the relaxation respects these constraints, so we implicitly treat them in the original problem, anyway. The explicitly stated original subproblem then corresponds to a relaxation of the extended subproblem and each solution to the master problem still transfers into a solution that fulfills all but the integrality constraints of the original subproblem.

In this case, we primary solve the extended problem. Nevertheless, when the current solution to the master problem transfers into a solution to the original problem that fulfills the integrality restrictions, the current subproblem is solved to optimality. It can contain no solution with better objective value than the solution obtained by the transformation—this solution is an optimal solution of the relaxation, hence giving a valid lower bound—so we can stop the solving process of the current node.

However, the SCIP instance representing the original problem is the coor-

dinating one, so branching rules have to be included into this instance. They can, of course, create two nodes in the original problem that do not have any further restrictions and only impose restrictions on the corresponding extended problems.

SCIP does not store the whole problem corresponding to a node explicitly, it only stores for each node the modifications of the problem that were done at this node. When changing from one node to another, it deactivates all nodes from the current one up to the deepest common ancestor node and from this node on, it activates all nodes on the way to the new node. Whenever deactivating a node, all modifications that were performed at this node are reverted, when a node is activated, the corresponding changes are applied.

A branching rule can store at each node of the branch-and-bound tree information about the modifications of the problem. Additionally, it can provide methods that are called whenever a node is activated or deactivated which enforce these changes in the master problem or the pricing problems. For example, when branching on the original variables and enforcing the branching decisions in the pricing problems, these methods can add and remove the bound constraints in the pricing problems.

At each node of the branch-and-bound tree, SCIP performs *domain propagation*, also called *node preprocessing*. Its goal is to tighten the domains of variables for the current problem in the branch-and-bound tree. For a detailed description how this is done, we refer to [1, Chapter 7].

We make use of this concept in two ways. On the one hand, a branching rule can implement a method that is called during the domain propagation process and that can change the bounds of variables for the current master problem. We need to do this if branching restrictions are enforced in the pricing problems. Since we do not start the solving process of the RMP at each node with an empty set of variables, but use all the variables created before at other nodes, we have to make sure that only variables are used in the master problem that correspond to valid solutions of the modified pricing problem. So each time a node is activated and new variables were created since its last deactivation, we mark this node to be repropagated. In the subsequent domain propagation process, the new variables are checked for feasiblity w. r. t. the current pricing problem and fixed to zero for the current problem if they are not feasible. Hence, we do not remove the variables completely from the current problem but only forbid them locally, since they are possibly needed in other parts of the branch-and-bound tree.

On the other hand, we keep the domain propagation of SCIP activated in the original problem. Thus, especially when branching on the original variables, the branching restrictions lead to other domain reductions in the original problem. When propagating the corresponding node of the extended problem, besides the domain changes due to the branching decision, we can enforce these domain changes in the pricing problems, too, and forbid all variables in the master problem of the current node that do not respect them. The remaining feasible variables, i.e., variables that also fulfill bounds derived

from the original constraints, are called *proper* (see [96]). This functionality can of course be disabled by the parameter *enfoproper*, but as we will see in the computational results presented in Section 7.3, using only proper variables slightly improves the performance of GCG.

In this context, we benefit from the fact that our implementation of the branch-cut-and-price solver is integrated into SCIP which includes several domain propagation methods that are automatically called for each node of the branch-and-bound tree in the original instance. Furthermore, each time the domain propagation methods in SCIP are enhanced, this will potentially improve the performance of GCG as well.

5.6.1 Branching on Original Variables

Branching on the original variables has the highest priority in our implementation so it is performed whenever possible. However, we only branch on variables if the corresponding block was not aggregated, hence in the current version of GCG, we do not support a dynamic disaggregation. If branching should be performed on variables of identical blocks, the aggregation of blocks has to be disabled in advance. We do not consider a dynamic disaggregation, since aggregating blocks leads in this case only to a temporary speedup of the master problem's solving process, later on, the pricing problems have to be split anyway and the variables of one class have to be distributed among the single blocks or copied and added to all of the blocks. Furthermore, branching on the original variables in case of identical blocks reintroduces the symmetry and is thus rather ineffective as we stated in Section 5.1.

In the following, we thus assume that there exist integer variables that do not belong to an aggregated block and that have fractional value in the current solution. The task of the branching rule is to select one of these variables and to perform branching on it. Most state-of-the-art MIP solvers use branching on variables as well, and it is well-known, that the choice of the variable on which to branch has a big impact on the performance of the branch-and-bound process (see [5]). Different strategies are used in practice, ranging from simple methods like *most fractional branching* to more sophisticated methods like *pseudocost branching*, *strong branching* and *reliability branching*.

Most fractional branching—or most infeasible branching—selects the variable that has the most fractional value, i.e., for which the fractional part is closest to 0.5. This branching rule is easy to implement but it leads in general to a performance that is not better than selecting the variables randomly (see [5]).

Pseudocost branching keeps a history of the success of branching on a variables at previous nodes. In particular, it stores the objective gain per unit change of the variable for up-branch and down-branch, independently. At subsequent nodes, out of the fractional variables, one is selected that has the highest score w. r. t. these pseudocosts. The score is a combination of

the average increase of the dual bound at both the child nodes. Pseudocost branching performs in general remarkably better than most fractional banching, Achterberg et al. [5] report a speedup of about 200%. At the beginning of the branch-and-bound process, however, there are no pseudocosts available for the branching candidates so the variable has to be selected in a different way.

Before branching is performed, strong branching checks which of the fractional candidates leads to the best progress. This is done by temporarily adding the bound constraint to the LP and solving it for both the down-branch as well as the up-branch. These values are combined to a score as for the pseudocost branching and branching is performed on a variable with the highest score. Strong branching performs well in term of the branch-and-bound nodes that have to be processed, but the time needed for branching is very high.

Reliability branching is a combination of strong branching and pseudocost branching. The pseudocosts corresponding to a variable are only used if they are reliable, i.e., they were updated a specific number of times for this variable. Otherwise, the score for this variable is computed using strong branching. Reliability branching performs in general better than the alternatives mentioned previously and it is also the default branching scheme in `SCIP`.

For a more detailed description of these and further branching rules, we refer to [5] and [1, Chapter 5].

We transferred two of these possibilities to our implementation, most fractional branching and pseudocost branching. We refrained from providing possibilities to perform strong branching on variables, as solving the master problem after adding a branching decision is much more costly than solving the LP with an additional bound constraint in an LP-based branch-and-bound algorithm. Furthermore, due to the stronger dual bound obtained by the master problem, the branch-and-price approach needs on average considerably less nodes than an LP based branch-and-bound method (see Section 7.4). At each of the nodes, the relaxation is solved, so when we perform strong branching a number of times before the pseudocosts are reliable, this weights more for the branch-and-price approch than solving the same number of LPs for strong branching in a branch-and-bound approach since much more relaxations are solved there, anyway.

Per default, we use pseudocost branching, most fractional branching can be used by setting the parameter *usepseudocosts* to *FALSE*. When using pseudocost branching, we make use of the structure of `SCIP` that provides the possibility to store pseudocosts associated with variables of the problem. Furthermore, `SCIP` also computes the score for branching on a variable with given fractional value w. r. t. the pseudocosts associated with that variable. The function used to determine this score can be changed, but we use the default function incorporated in `SCIP` that calculates the score via a product (see [1, Chapter 5]). We perform branching on a fractional variable with

the highest score. SCIP automatically updates the pseudocost of variables after the LP relaxation of a node has been solved the first time. Since we do not use the LP relaxation in the original SCIP instance, the pseudocosts are not stored automatically. Instead, we defined an additional callback function that can be implemented by the branching rules and that is called after the relaxation is solved. We introduced this callback function for GCG to provide a possibility to store pseudocosts after the master problem is solved to optimality at a node. For the branching on original variables, we compute the change between the dual bound of the father node and the dual bound of the current node and update the pseudocost of the variable on which we performed branching w. r. t. this change.

As described in Section 5.1, branching decisions can be enforced either in the master problem or in the pricing problems. In the default settings, we enforce the branching restrictions in the pricing problems. Therefor, for each of the child nodes in the original SCIP instance that are created by the branching rule, we change the bound of the variable on which branching is performed. Furthermore, we also store the information about the bound change at the child node and implemented the callback methods, that are called whenever a node is activated or deactivated. When a node is activated, the corresponding method changes in the pricing problem the bound of the variable on which branching was performed and stores the old bound. This is needed in order to enable the deactivation callback method to restore the old bound when the node is deactivated. We did not have to implement the callback method that is called during the domain propagation process since the propagation is automatically performed by the constraint handler that is responsible for the linkage between nodes in the original and the extended instance. It identifies all branching bound changes at the corresponding original node and fixes all variables in the master problem to 0 that correspond to solutions that do not fulfill these bound changes.

Alternatively, the branching decisions can also be enforced as linking constraints in the master problem. In this case, we add the branching restrictions as local constraints to the child nodes. The branching decisions are stored at the nodes, again. Whenever a node is activated, the corresponding callback method reformulates the branching constraint according to the Dantzig-Wolfe decomposition and adds it locally to the node in the extended instance. When domain propagation is performed in the original instance, the branching constraint will lead to a bound change of the branching variable and this might cause bound changes of other variables as well. As we do not want to modify the pricing problems when enforcing branching decision in the master problem, we have to disable the enforcement of proper variables in the master problem. Otherwise, these bound changes would automatically be transferred to the pricing problems and variables in the master problem that correspond to solutions of the pricing problems that violate these bounds would be fixed to 0. This way, we would enforce the branching restrictions in the pricing problem as well as in the master problem which is redundant.

In Section 5.7, we present computational results that compare most fractional branching and pseudocost branching as well as the enforcement of the branching restrictions in the pricing problems and the master problem.

5.6.2 Ryan and Foster Branching

We also provide a rather fundamental implementation of Ryan and Foster's branching scheme. In case all blocks are identical and the master problem has a set partitioning structure, this branching rule is called.

In order to determine the two constraints of the master problem that are examined in the branching process, we first look for a variable x that has fractional value (say v) in the current solution of the master problem. Then, we take one constraint that contains this variable and look for another variable y contained in this constraint that has a fractional value in the current solution, say w. Such a variable must exist since the values of the variables contained in the constraint sum up to 1. If such a variable would not exist, the solution would not fulfill the constraint. Now, we look for a second constraint that contains exactly one of these two variables. Such a constraint must always exist since otherwise, both variables correspond to the same point in the pricing polyhedron. However, we create at most one variable per point, so this cannot happen. Thus, we found two constraints for which the sum of the variables contained in both constraints is fractional: Say x is contained in both constraints, then the value of the variables contained in both constraints is at least $v > 0$. On the other hand, since y was contained in the first constraint but not in the second one, the sum is at most $1-w < 1$.

After finding two constraints that we will branch on, we additionally need the variables in the pricing problem that are associated with these constraints. Each constraint in the extended problem corresponds to a variable in the pricing problem: it contains all variables that corresponds to a solution with value 1 for the pricing variable. So let x_i^k and x_j^k be the two variables that correspond to the selected constraints.

After selecting the two constraints, we create, in the original problem, two child nodes SAME-child and DIFFER-child. As described in Section 5.4, we add a constraint $x_i^k = x_j^k$ for each block $k \in [K]$ at the SAME-child; for the DIFFER-child, we add the linear constraints $x_i^k + x_j^k \leq 1$ for $k \in [K]$. Furthermore, we store information about the branching decisions at the nodes.

We implemented the callback functions that are called when nodes are activated or deactivated and the callback function that is called during the domain propagation process.

The callback that is called when nodes are activated creates a constraint in the pricing problem corresponding to the branching constraints added in the corresponding node in the original instance. Since all blocks are identical and the pricing problem that belongs to the first block represents all blocks, we create in this pricing problem the constraint $x_i^1 = x_j^1$ or $x_i^1 + x_j^1 \leq 1$

depending on whether it is a SAME-child or a DIFFER-child. This is only done the first time a node is activated, later on, the constraint already exists, but it is not active in the SCIP instance representing the pricing problem. Therefore, we activate the constraint in the pricing problem when the node is activated and deactivate the constraint in the deactivation callback.

In the propagation callback, we have to assure that only variables are active at the current node and its subtree that respect the branching restriction. Therefore, we iterate over all variables that were created after the last deactivation of the node and check whether they correspond to a solution of the pricing problem that fulfills the branching constraint. All variables for which this does not hold are locally fixed to zero. We do not have to check variables that were created before the node's last deactivation, since these variables were either checked when the node was activated the last time and are thus automatically fixed to 0 when the node is activated again, if needed, or they were created in the subtree defined by this node and therefore respect the branching decision.

5.7 Computational Results

In this section, we compare computational results for the two branching rules integrated in GCG as well as different settings for these branching rules. We give no comparison of the performance of GCG to the performance of SCIP, this will be done in Chapter 7. The test sets and the computational environment used in this chapter are described in Section 3.3. For all computations presented in this section, we set a time limit of one hour per instance.

5.7.1 Branching on Original Variables

In this subsection, we present computational results concerning the integrated branching rule that branches on variables of the original formulation. We picture the effect of using pseudocosts in the branching routine and the effect of where to enforce branching restrictions: In the pricing problems or in the master problem.

In this context, the branching rule provides two major parameters: On the one hand, the method that is used to choose the variable to branch on can be specified. On the other hand, we have to decide whether the branching restrictions are enforced in the pricing problems or in the master problem.

Variable Selection For the selection of a variable, we provide a most fractional selection rule and a selection rule that makes use of the pseudocosts of the original variables. The latter is used per default. A comparison of these two possibilities for the p-median test sets is presented in Table 5.1. We list the shifted geometric mean of the number of branch-and-bound nodes and of the total solving time in seconds as well as the absolute number of instances

test set	pseudocost nodes	pseudocost time (outs)	most fractional nodes	most fractional time (outs)
CPMP50S	44.9	12.8 (0)	82.5	20.7 (0)
CPMP100S	587.1	184.7 (1)	1962.6	469.5 (6)
CPMP150S	847.6	493.5 (5)	2211.7	920.3 (10)
CPMP200S	1753.3	1243.9 (3)	5577.7	2978.0 (10)
mean	461.8	220.1 (9)	1216.6	439.7 (26)

Table 5.1. Comparison of the pseudocost and the most fractional variable selection rule for branching on original variables on the p-median test sets. The first two columns denote branch-and-bound nodes and solving time for pseudocost branching, the last two columns branch-and-bound nodes and solving time for most fractional branching. We list shifted geometric means for each test set. Following the time, in brackets, we list the number of timeouts on the test set.

that hit the timelimit of one hour. More details can be found in Tables D.21 to D.24.

We can see that using pseudocosts has a big effect on the performance of GCG. Disabling it and using most fractional branching instead nearly triples the number of branch-and-bound nodes and doubles the total solving time. Also the number of timeouts is increased from 9 to 26. Similar results for an LP based branch-and-cut solver are presented in [5]. Hence, this is an example for a concept used in LP based branch-and-cut that can successfully be transferred to the generic branch-cut-and-price approach presented in this thesis and that leads to similar improvements.

In [5], a further speedup of 40% is reported for using reliability branching in an LP based branch-and-cut solver. It remains to investigate whether this branching rule can successfully be transferred, too. As mentioned in Section 5.6, we doubt this, since the additional effort of the strong branching calls seems to be more serious for the branch-cut-and-price approach as solving the master problem is much more costly and the number of nodes is in general much smaller, anyway.

Enforcement of the branching restrictions Per default, when branching on original variables, the branching restrictions are enforced in the pricing problems. However, we also provide the possibility to enforce them in the master problem. We performed computations that demonstrate the influence of this decision. An overview of the results for the p-median test sets is presented in Table 5.2. We list the shifted geometric mean of the number of branch-and-bound nodes and of the total solving time as well as the absolute number of instances that hit the timelimit of one hour. More details can be found in Table D.21 to D.24.

In Section 5.1, we mentioned that enforcing the branching restrictions in the pricing problems typically leads to a stronger dual bound. In return, we have some effort for ensuring that variables are forbidden that do not respect

test set	pricing problems nodes	pricing problems time (outs)	master problem nodes	master problem time (outs)
CPMP50S	44.9	12.8 (0)	44.6	12.5 (0)
CPMP100S	587.1	184.7 (1)	596.2	181.7 (1)
CPMP150S	847.6	493.5 (5)	830.4	501.0 (5)
CPMP200S	1753.3	1243.9 (3)	1926.6	1343.6 (3)
mean	461.8	220.1 (9)	471.8	223.8 (9)

Table 5.2. Comparison of two ways to enforce the branching restrictions for the branch-cut-and-price solver `GCG` on the p-median test sets. The first two columns denote branch-and-bound nodes and solving time when restrictions are enforced in the pricing problems, the last two columns branch-and-bound nodes and solving time when enforcing the restrictions in the master problem. We list shifted geometric means for each test set. Following the time, in brackets, we list the number of timeouts on the test set.

the active branching constraints at a node.

In our computations, it turned out that these effects are rather small. The shifted geometric mean of the number of nodes is increased by 2.1% when branching restrictions are enforced in the master problem. This could be an indication for the better dual bound that allows to prune more nodes. However, since it is a rather small difference, it could also be a random variation. The change in the total running time is even smaller—it is increased by 1.7% when branching restrictions are enforced in the master problem—and the number of time outs stays the same. Hence, this difference could also be a random variation. However, it can also be a sign that enforcing the restrictions in the pricing problems and spending the effort to forbid some of the variables pays off due to the stronger dual bound even in terms of total running time.

To sum up, for the CPMP instances, the decision where to enforce branching restrictions is nearly irrelevant for the number of branch-and-bound nodes and the total running time.

Results for the set of RMP instances We also performed computational experiments concerning variable selection and enforcement of branching restrictions for the test sets of RMP instances. Table 5.3 summarizes the results, more details can be found in Tables D.25 and D.26. For these instances, the dual bound obtained by the Dantzig-Wolfe decomposition is rather strong such that we need only a small number of branch-and-bound nodes. When grouping 64 constraints per block, we need in the geometric mean less than ten nodes to solve an instance. As pseudocosts for a variable are obtained only when branching on this variable has been performed, the first branchings have to be performed without the guidance of pseudocosts. Therefore, for this small number of branch-and-bound nodes, pseudocosts do not pay off, we even have slight improvements when using the most fractional variable selection rule, probably due to random variations. When enforcing

test set	default		most fractional		master problem	
	nodes	time (outs)	nodes	time (outs)	nodes	time (outs)
RAP32S	26.8	549.0 (0)	32.9	615.8 (1)	27.3	569.8 (0)
RAP64S	9.5	557.5 (1)	9.4	547.7 (1)	8.8	548.1 (1)

Table 5.3. Performance effect of variable selection and enforcement of branching restrictions for the test sets of RAP instances. The first two columns denote branch-and-bound nodes and solving time for the default settings, i.e., pseudocost variable selection and enforcement in the pricing problems. The next two columns list nodes and time when using most fractional instead of pseudocost variable selection. The last two columns present these values when branching restrictions are enforced in the master problem, using again pseudocost variable selection. The values are shifted geometric means for the test sets. Following the time, in brackets, we list the number of timeouts on the test set.

the branching restrictions in the master problem, we also get a slightly smaller shifted geometric mean of the solving time and a decrease of the shifted geometric mean of the number of nodes of about 7%. Due to the small number of nodes however, this is not convincing as it can for instance also be caused by accidentially finding an optimal solution earlier in the solving process.

When grouping 32 constraints per block, the dual bound is not as good as for 64 constraints, but the solving process of the master problem gets easier. Therefore, the total solving time is slightly reduced for the default settings although the number of nodes is tripled. Due to the larger number of nodes, the effects of the branching rule can better be detected. The number of nodes is increased by about 23% when using the most fractional variable selection rule, the total time is higher by 12%. The effect is not as big as for the p-median instances, since the number of nodes is still rather small so the pseudocosts are less effective as they get more reliable the more branchings have been performed on the corresponding variable. When branching restrictions are enforced in the master problem, the number of nodes and the total time are slightly increased as for the p-median instances. However, this change is not significant.

5.7.2 Ryan and Foster Branching

In Section 4.4.2, we showed that for the bin packing and vertex coloring instances, using the discretization approach and aggregating the identical blocks speeds up the solving process of the master problem by far. Now, we compare the performance of both approaches in the branch-and-price process. For the convexification approach, we use reduced cost pricing strategy "one prob" that performed best for these problems (see Section 4.4.2) and perform branching on variables of the original problem. For the discretization approach, we use the default pricing settings, i.e., $M_r = 100$, which performed best in this case. We cannot branch on the original variables since this would destroy the identity of the blocks, so we use the Ryan and Foster branching scheme as both problems have a set partitioning master problem.

test set	convexification nodes	convexification time	discretization nodes	discretization time
BINDATA1-N1S	11.0	2.1	2.2	1.1
BINDATA1-N2S	20.8	15.1	10.9	5.6
BINDATA1-N3S	31.4	61.2	8.9	17.3
mean	19.9	17.9	6.9	6.8

Table 5.4. Comparison of the convexification approach with branching on original variables and the discretization approach with Ryan and Foster's branching scheme for the test sets of bin packing instances. All values are shifted geometric means.

Results for the bin packing instances In Table 5.4, we present a summary of the results on the test sets of bin packing instances. The discretization approach and the aggregation of blocks primary affects the time since the master problem at each node is solved faster. However, the dual bound of a node stays the same, so it has no direct influence on the number of nodes. The latter is primary affected by the branching rule. It turns out, that Ryan and Foster's branching scheme (column "discretization") performs remarkably better than branching on original variables (column "discretization"), the number of nodes is reduced by about 65% in the discretization approach. Hence, despite of the lower bound being better, branching on original variables still suffers from the symmetry contained in the problem. Due to the smaller number of branch-and-bound nodes, the speedup obtained by the better pricing performance is enhanced and we achieve a speedup of about 60%. In order to assure that "one prob" performs well for the convexification approach with identical blocks not only for the master problem of the root node but also for the branch-and-price process, we performed computations with the default pricing settings, too. The number of nodes essentially stays the same, but the total time is nearly quintupled. More details about all these results can be found in Tables D.27 to D.29.

Results for the vertex coloring instances Finally, in Table 5.5, we present a summary of the results for the vertex coloring test set, further details can be found in Table D.30.

Compared to the convexification approach with default pricing settings (columns "conv. def."), the discretization approach with aggregation of blocks (columns "discretization") is clearly superior in terms of the number of nodes as well as of the total running time.

However, we observed in Section 4.4, that solving just one pricing problem in each pricing round performs better at the root node than the default settings for identical blocks in the convexification approach. For the branch-and-price process, this pricing strategy (columns "conv. one prob") dominates the standard pricing method of the convexification approach, too. Compared to the discretization approach, it leads to a higher number of nodes, but the total time is reduced.

conv. def.		conv. one prob		discretization		disc. no aggr.	
nodes	time	nodes	time	nodes	time	nodes	time
84.4	151.2	78.2	60.6	52.3	65.9	69.3	56.0

Table 5.5. Comparison of branching on original variables and Ryan and Foster's branching scheme for the test set of vertex coloring instances. All values are shifted geometric means.

Even better than "conv. one prob" performs the discretization approach without aggregation of blocks, when using the same pricing strategy (columns "disc. no aggr."). Compared to the results for the former variant, the shifted geometric means of the number of nodes and the solving time are reduced by eleven and seven percent, respectively. The reason for that is that primal heuristics are used in the extended formulation of the discretization approach that find feasible solutions which allows to prune nodes earlier.

Now, we compare the discretization approach with aggregated blocks and Ryan and Fosters's branching scheme on the one hand and the discretization approach without aggregation and with branching on original variables on the other hand. This gives us some more insight into the advantages and disadvantages of Ryan and Fosters's branching scheme. The branching scheme can be used after the aggregation of blocks and does not suffer from the symmetry contained in the vertex coloring problem. So it typically helps in improving the dual bound faster.

However, most of the instances contained in our test set have a rather tight dual bound at the root node (see also Table D.37) so that the problem is solved as soon as an optimal solution is found. For these instances, branching on original variables performes better. The branching restrictions lead to less modifications of the pricing problem and invalidate a smaller number of variables already contained in the RMP.

In Section 4.4, we observed that for the vertex coloring instances, the pricing performance of the discretization approach is not much better than the performance when blocks are not aggregated and just one problem is solved each round. So due to the smaller impact of the branching restrictions, the pricing effort per node is smaller when branching on the original variables. This advantage overweights the disadvantage that without aggregation of blocks, we need more nodes to find an optimal solution and prove its optimality.

However, if the dual bound is not that tight, branching on the original variables suffers from the symmetry contained in the problem and cannot improve the dual bound as fast as Ryan and Fosters's branching scheme. This is the case for one of the instances where both approaches find an optimal solution early in the branch-and-price process. While we can prove its optimality after a small number of nodes when using Ryan and Foster's branching scheme, the optimality cannot be proven within the time limit of one hour when branching on original variables.

To summarize, it seems that branching on original variables performs better for the coloring instances (also since the pricing effort is not increased that much by treating the blocks independently) when the dual bound is (nearly) optimal and we just have to find an optimal solution. However, when this is not the case and we also have to improve the dual bound, it suffers from the symmetry contained in the problem due to the identical blocks and is clearly inferior to Ryan and Foster's branching scheme. Therefore, if it is applicable, using aggregation of blocks and Ryan and Foster's branching scheme is in general superior to treating blocks independently and performing branching on the original variables.

Chapter 6

Separation

In the last two sections, we have presented the pricing process and different branching schemes. These are the essential ingredients needed to turn a branch-and-bound method into a branch-and-price method. In this chapter, we present the integration of cutting planes into a branch-and-price algorithm which leads to a branch-cut-and-price algorithm.

Cutting plane separators have proven to be one of the most important features of MIP solvers [12]. After solving the LP relaxation at a node, separators try to rule out the computed optimal solution of the relaxation if it does not satisfy the integrality restrictions. This is done by constructing linear constraints that forbid the current fractional solution, but are typically valid for all integer solutions. These constraints are called *cutting planes* (or *cuts*), since they "cut off" the current fractional solution. In some cases, cutting planes even forbid a subset of the integer solutions, e.g., in the context of symmetry breaking. Then, they must keep at least one optimal primal solution. We do not handle this type of separation in this chapter but restrict ourselves to cutting planes whose only task is to strengthen the LP relaxation and that do not cut off feasible primal solutions.

The algorithm that results from integrating cutting plane separation into a branch-and-bound algorithm is called *LP based branch-and-cut*. For a survey of the theory of cutting planes, we refer to [54, 59]. A detailed description of the cutting planes incorporated in `SCIP` and the underlying theory can be found in [98, 6].

We just give a short explanation here of what we mean when we say "cutting planes are derived or constructed using a specific set of constraints". The set S of points that satisfy this specific set of constraints as well as the integrality restrictions is a superset of the solutions to the complete MIP containing all constraints. Its convex hull, i.e., the polyhedron $conv(S)$, is a relaxation of the MIP. It is tighter than the polyhedron described by just the set of constraints without regarding integrality restrictions. The solution of the LP relaxation lies in this larger polyhedron. We try to cut it off by constructing valid inequalities that are known to be valid for the tighter set $conv(S)$. Some cutting planes are constructed for a set containing just one

type of constraints, i.e., knapsack constraints, using the special structure of these constraints. General cutting plane separators regard an arbitrary subset of constraints and construct valid inequalities for instance by aggregating them and rounding the coefficients in a specified way.

For the branch-cut-and-price approach presented in this thesis, the relaxation is given by the master problem. So in order to strengthen this relaxation, we have to add cutting planes to the master problem that cut off solutions corresponding to fractional solutions of the original problem. For the discretization approach, this corresponds to cutting off fractional solutions of the master problem.

After cutting planes have been added, the master problem is reoptimized. For this purpose, column generation has to be used again, since due to the new constraints, the variables already contained in the RMP do not have to be sufficient to construct an optimal solution of the new master problem. Therfore, we have to make sure that the dual variables corresponding to the cutting planes are also considered in the pricing procedure and newly created variables get correct coefficients in the cuts.

When we derive cutting planes from the original formulation (see Section 6.1), this can easily be assured. However, since we do not solve the LP relaxation of the original problem by a simplex algorithm, we cannot use any cutting plane separator that relies on information contained in the simplex tableau, like the Gomoriy mixed integer cut separator.

For the discretization approach, we can also derive cutting planes from the extended problem. These inequalities cannot necessarily be expressed in terms of the original variables, so considering them in the pricing problem is more sophisticated. We briefly discuss this issue in Section 6.2.

We implemented a separation method that derives cutting planes from the original problem. and discuss implementational details in Section 6.3. Computational results concerning the impact of cutting planes for the branch-cut-and-price solver `GCG` are presented in Section 6.4.

6.1 Separation of Cutting Planes in the Original Formulation

In this section, we describe the separation of solutions in the original formulation. That means, we are given a solution to the master problem and retransform it into the original variable space. If this solution does not satisfy the integrality restrictions in the original problem, we look for a cutting plane that separates it from the feasible region of the original problem. This approach is widely used in practice, see for example [56, 33, 7].

Let us note that all valid inequalities that can be derived using only the structural constraints are satisfied by all points and rays represented by variables of the master problem, since the integrality restrictions are respected in the pricing problems. Hence, the current original solution which is given

as a convex combination of these points satisfies all these inequalities, too. Therefore, we only look for cutting planes that can be constructed using linking constraints or a combination of linking and structural constraints.

In this context, we have to get along with a handicap that arises from the Dantzig-Wolfe approach: We do not know an LP basis corresponding to the current original solution because we do not solve the LP relaxation of the original problem. In most cases, thw transferred solution is not even a basic solution w. r. t. the LP relaxation of the original problem but an interior point of the polyhedron decribed by the LP relaxation. It would only be a basic solution if we would explicitly replace the structural constraints of each block k by a complete description of the convex hull of solutions X_k of this block. The master problem, however, implicitly optimizes over this tighter polyhedron, due to the definition of the pricing problems. Having no basis is the tradeoff for the better dual bound achieved by the master problem.

Since we do not have a basis at hand, we cannot use separation methods that need the current basis or basis inverse like it is for instance the case for *Gomory mixed integer cuts* and *strong Chvátal-Gomory cuts*.

Instead, we can use all kinds of cutting planes that do not need any further information besides the problem and the current solution in their separation routine. This applies, for example, to *knapsack cover cuts*, *mixed integer rounding cuts*, and *flow cover cuts*.

Because the master problem provides a much tighter relaxation than the LP relaxation and because of the absence of the LP basis, we expect cutting planes derived from the original problem to be not as important for the solving process as in the general LP based branch-and-cut method. Anyway, limited but fast cutting plane separation in the original formulation could still lead to a slightly improved dual bound and we want to benefit from this possibility.

After finding valid cutting planes, we add them to the original problem. We treat them as a linking constraints, so that they are transformed according to the Dantzig-Wolfe decomposition and added to the extended formulation and the master problem. Each constraint gives rise to an additional variable in the dual of the master problem, that has to be respected in the pricing problems. However, we can handle these additional dual variables in the same way as the dual variables corresponding to the other linking constraints by considering them in the objective function coefficients of the pricing problems. The structure of the pricing problems does not change.

Again, we have to distinguish between the case that blocks are treated independently on the one hand and the aggregation of blocks on the other hand.

For the *convexification approach* or the *discretization approach without aggregation of blocks*, the whole separation rountine works straightforward as described previously.

For the *discretization approach with aggregated blocks*, it does not work that well. We can transform the master problem's solution into a solution to

the original problem, even though this transformation is not unique. If we find a cutting plane $\sum_{k\in[K]} \alpha_k^T x^k \leq \beta$ that cuts off this solution, this linear constraint damages with high probability the equality of the identical blocks, i.e., $\alpha_{k_1} \neq \alpha_{k_2}$ for two identical blocks $k_1, k_2 \in K_\ell, \ell \in [L]$.

Therefore, we have to disaggregate these blocks and reintroduce one pricing problem for each of them. Furthermore, we need to differentiate variables of different blocks representing the same point or ray of the pricing polyhedron. Hence, we reintroduce the symmetry that we eliminated by using the discretization approach and by aggregating identical blocks. Besides, due to the symmetry, this single cutting plane will probably not tighten the dual bound; it is likely that an equivalent solution can be constructed by permuting the former identical blocks. Thus, a set of similar cutting planes has to be added before the whole class of equivalent solutions is cut off.

For that reason, in case of a problem with identical blocks that are aggregated by the discretization approach, computing cutting planes in the original formulation seems to have more disadvantages than advantages. Therefore, we do not consider the creation of these cutting planes in this case.

6.2 Separation of Cutting Planes in the Extended Problem

When using the discretization approach, we have integrality restrictions on the variables of the extended problem, too. Hence, we can also derive cutting planes from the extended formulation.

Although only a subset of the variables is contained in the RMP, cutting planes have to be defined in terms of all variables of the master problem. Furthermore, the additional dual variable corresponding to the cut has to be respected in the pricing problems. This is the crucial part about cutting plane separation in the extended problem: Obtained cuts can typically not be expressed in terms of the original variables. Hence, the dual variables associated with these cuts cannot be respected in the pricing problems by just changing the objective function coefficients as it is done for cuts derived from the original problem (see Section 6.1). Instead, we typically need to introduce one or more additional variable in the pricing problems for each cutting plane. Together with some additional constraints, these variables are used to model the decision whether a new variable is contained in the cut or not. This can destroy the special structure of a pricing problem and increase its complexity (see [87]).

Nevertheless, this approach was successfully applied to the vehicle routing problem with time windows. A small subset of the Chvátal-Gomory cuts, called subset-row inequalities, are separated in the extended problem [45]. Each cut implies a modification of the pricing problem, which is a resource constrained shortest path problem: An additional resource is added to the problem for each cut.

In [74], this work is extended to include all Chvátal-Gomory rank-1 cuts, but this also implies more modifications to the pricing problems and seems to be computationally inferior.

For more details about deriving cutting planes from the extended formulation, we refer to [87]. In particular, Chapter 6 presents a general framework that allows to formulate cuts derived from the extended problem in the original problem. However, this includes additional variables and constraints that are added to the original problem.

In this thesis, we focused on separation of cutting planes in the original formulation and will not regard separation in the extended problem in the following.

6.3 Implementation Details

We included a separation routine into `GCG` that derives cutting planes from the original problem. It is compatible with both the convexification as well as the discretization approach. However, for the reasons described in Section 5.1, it is not used in combination with an aggregation of blocks. We implemented a separation plugin that is included into the `SCIP` instance representing the extended problem. As the master problem is solved in this `SCIP` instance and the separation routine is to be called during this process, the separator has to be added here.

The separator is called by `SCIP` as soon as the master problem has been solved to optimality using column generation. The computed optimal solution of the master problem is transferred into a (possibly fractional) solution of the original problem.

In the original `SCIP` instance, all default separation plugins of `SCIP` are activated. These plugins provide two essential callbacks: One for the separation of an LP solution (called *SEPALP*) and a second one that separates an arbitrary solution (called *SEPASOL*). These two callbacks are distinguished, since for the separation of an LP solution, the separator can access information about the current basis of the LP. The SEPASOL callback can be called without having solved the LP before so it must not access information about the LP basis. Both types of callbacks can be provided by constraint handlers, too, in order to separate constraint specific cutting planes. Some of the default separators do not implement the SEPASOL callback since they need information about the basis, e.g., the `gomory` and the `strongcg` separators, that separate *Gomory mixed integer cuts* and *strong Chvátal-Gomory cuts*, respectively.

When `SCIP` is used as an LP based branch-and-cut solver, it only uses the SEPALP callback. The new separator that we included into the extended `SCIP` instance implements this callback, too, since it separates the solution of the master problem, i.e., the LP relaxation of the extended problem. However, in order to separate the transferred master solution in the

original SCIP instance, we just need the SEPASOL callbacks of the default separators, since we separate an arbitrary solution that was not computed by the LP relaxation of the original problem. The default separators that implement this callback are the *clique* separator, the *cmir* separator, and the *flowcover* separator, that construct *clique cuts*, *mixed integer rounding cuts*, and *flow cover cuts*. The *mcf* separator implements this callback, too, and constructs cutting planes for problems with a multi-commodity flow structure. Furthermore, the *knapsack* constraint handler, which generates *lifted cover inequalities*, also provides this type of callback. For more details about these separators, we refer to [98] and [6].

After transferring the master solution into the original variable space, we let SCIP separate this solution by calling all supported SEPASOL callbacks. If cuts are found, we transfer them into the variable space of the extended problem as it is done for the linking constraints of the original formulation (see Chapter 2).

The transferred cuts are added to the LP relaxation of the SCIP instance representing the extended problem, which is the master problem. In fact, we do not directly add them to the master problem, but we add them to the separation storage. Since adding just one cut and resolving the LP afterwards is rather ineffective, separation is performed in rounds. In each round, the separators try to generate various cutting planes and add them to the separation storage. However, by adding all of these cuts to the problem, the LP size would increase too much. Therefore, SCIP selects only a subset of the cutting planes contained in the separation storage and adds them to the LP at the end of each separation round; remaining cuts are discarded. The cuts to be added are automatically selected by SCIP with respect to their strength and numerical stability. For more details, we refer to [1].

The new separator stores both the transferrd cut and the corresponding original cut. This way, the dual variables of the latter can be respected in the pricing problems. We have to distribute it among the variables of the pricing problems in the same way as described in Section 4.2 for the original linking constraints. Furthermore, we need the original counterparts in order to determine the coefficients of the variables in the extended problem w. r. t. the cutting planes.

If cutting planes were added, SCIP automatically solves the new master problem using the implemented variable pricer plugin (see Section 4.3) and the process is iterated.

6.4 Computational Results

In this section, we present computational results concerning the separation of cutting planes in the original problem.

Unfortunately, for the four problems classes described in Appendix C, the separators do not find any cutting planes in the original formulation. This

can be explained by the structure of the problems, especially the structure of the linking constraints, as we will describe in the following.

Since we respect the integrality constraints in the pricing problems, each solution of the master problem corresponds to an original solution candidate that already satisfies all valid inequalities that can be derived using only the structural constraints (see Section 6.1). Hence, cutting planes have to be derived using only linking constraints or by a combination of linking and structural constraints.

Three of the problems—the bin packing problem, the vertex coloring problem, and the capacitated p-median problem—have set partitioning linking constraints. The RAP instances have another special structure in the linking constraints, namely constraints forcing two variables to have the same value, i.e., constraints of the form $x - y = 0$.

As mentioned in the last section, we can use five of the default separators incorporated in `SCIP`. However, the structure of the problems does not fit for three of them. First of all, the linking constraints do not contain knapsack constraints, so the separation method of the `knapsack` constraint handler is not able to derive cutting planes using these constraints. Furthermore, the problems do not have a flow or multi-commodity flow structure, so the `flowcover` separator and the `mcf` separator do not find cutting planes as well.

For the last two separators, the `clique` separator and the `cmir` separator, we have to take into account the relation between linking and structural constraints, too. Each linking constraint contains at most one variable of each block and each variable of a block is contained in at most one linking constraint. We conjecture that this together with the integrality of the data typically makes it impossible to create `clique` or `cmir` cuts that are not satisfied by the transferred master solution.

In order to support this conjecture, we performed computations concerning the dual bound at the root node. The results are summarized in Table 6.1, further details can be found in Tables D.31 to D.40 in Appendix D.

We solved the master problem of the root node with `GCG` and list the shifted geometric mean of the gap between the dual bound and the solution value of the optimal (or best known) primal solution for each test set. Additionally, we also list shifted geometric mean of the time needed for the solving process of the root node. Furthermore, we solved these instances with plain `SCIP`, too, using either all cutting plane separators (columns “SCIP all cuts”) or only those separators that can also be used by `GCG` (columns “SCIP no base cuts”). For both `SCIP` settings, we list the shifted geometric mean of the gap to the optimal or best known solution, the number of times, `SCIP` obtained the better dual bound (column “b”) and the worse dual bound (column “w”) as well as the shifted geometric mean of the solving time of the root node. We disabled reliability branching and used most infeasible branching, since otherwise, much time was spent by `SCIP` for the strong branching calls for branching at the root node. Since the strong branching calls can further im-

	GCG		SCIP all cuts				SCIP no base cuts			
test set	gap	time	gap	b	w	time	gap	b	w	time
CPMP50	2.15	2.6	2.51	12	24	1.7	2.91	0	35	0.5
CPMP100	2.63	9.4	2.89	13	27	6.3	3.24	0	39	2.7
CPMP150	1.83	27.6	2.09	12	28	16.8	2.48	0	40	7.7
CPMP200	3.19	63.1	3.88	12	28	42.7	4.44	0	40	16.1
RAP32	0.04	224.9	1.17	0	70	23.5	1.92	0	70	16.0
RAP64	0.02	366.0	1.17	0	70	23.5	1.92	0	70	16.0
COLORING	3.55	11.4	24.81	0	24	35.7	25.42	0	23	28.5
BINDATA1-N1S	0.37	0.7	5.59	0	17	0.7	5.44	0	16	0.4
BINDATA1-N2S	0.57	3.5	2.91	0	15	3.0	2.91	0	15	2.1
BINDATA1-N3S	0.25	15.6	3.22	0	15	21.3	3.22	0	15	17.4

Table 6.1. Comparison of the dual bound obtained at the root node by GCG and SCIP using either all separators, or only those separators that do not need basis information. We list the shifted geometric means of the gap to the optimal or best known solution and the solving time (in seconds) of the root node for each test set. Furthermore, for both SCIP settings, we list the number of times, SCIP obtained a better dual bound (b) or a worse dual bound (w).

prove the dual bound, but we just want to compare the dual bounds obtained by solving the root node *relaxation*, we disabled this feature.

We can see that for all instances, the dual bound obtained by the master problem is at least as tight as the dual bound obtained by the LP relaxation when using just the cutting plane separators that are used by GCG, too. This supports the conjecture that these cutting plane separators are not able to forbid the transferred optimal solution of the master problem.

Besides, the master problem gives rise to the tighter lower bound in the shifted geometric mean even when compared to SCIP with all cutting plane separators. For the bin packing, vertex coloring and RAP instances, the dual bound computed by SCIP is never better than the bound obtained by GCG. For the CPMP instances, the dual bound computed by SCIP is in most cases worse than the one computed by GCG but for some of the instances, SCIP obtains better bounds than GCG.

This shows that the computed optimal solution of the master problem can theoretically be cut off by the cutting plane separators incorporated in SCIP. Since the cutting planes created by the gomory and strongcg separators are special kinds of cmir cuts, the transferred master solution could theoretically also be ruled out by the separators that are used by GCG. However, finding these cuts without the guidance usually provided by the simplex tableau seems unlikely. Furthermore, it is also not clear whether constructing the cuts is possible with nothing but the constraints that are initially contained in the original problem or whether those inequalities that cut off the transferred master solution are created with the aid of other cutting planes that were created in former separation rounds. For the branch-and-price approach, it is possible that these cuts are implicitly treated in the master problem and

would therefore not be violated by the transferred master solution. Hence, they would not be generated in the separation process and thus, we cannot derive stronger cuts with their help.

In order to enhance the separation capabilities of `GCG`, we could think about adding valid inequalities to the original problem even if they do not cut off the current transferred master solution. This way, we might support the derivation of cutting planes from combinations of these inequalities and the initial constraints. It should be studied whether this helps for the separation process.

Finally, let us note that we also performed test runs for some general MIPs taken from the MIPLIB 2003 [5]. Blocks were detected by a method of the `mcf` separator that looks for a multi-commodity flow structure in the model. Hence, blocks corresponded to minimum cost flow problems. Since these problems were defined by integral data for the regarded instances, the subproblems possessed the integrality property and the bound obtained by the master problem equaled the LP bound (see Section 2.3). For these instances, we were able to find cutting planes in the original problem.

Chapter 7

Results

In the last three chapters, we presented the most important components of the branch-cut-and-price solver GCG:

- The column generation—or *pricing*—process (Chapter 4), that is used to solve a relaxation of the problem in order to obtain strong dual bounds.
- Different *branching* schemes (Chapter 5), that split the current problem into subproblems. Due to the integrality restrictions, this is needed to solve the problem to optimality with a branch-and-price approach.
- The separation of *cutting* planes (Chapter 6), that allows to tighten the relaxation by adding additional valid inequalities.

In each of these chapters, we presented the theoretical background and details about the implementation in GCG. Furthermore, we presented "local" results, i.e., computational experiments concerning just the specific part of the solver treated in the respective chapter.

Now, we present further "global" computational results, i.e., we regard the performance of GCG for the entire solving process.

We start with a comparison of some strategies for Farkas and reduced cost pricing in Section 7.1. In Chapter 4, we did this only for the solving process of the root node relaxation, now, we regard the effect for solving the relaxations of all nodes of the branch-and-bound tree.

After that, in Section 7.2, we demonstrate the impact of using a problem specific pricing problem solver rather than solving the pricing problems with a general MIP solver.

In Section 7.3, we review some advanced features that should speed up the solving process and present their impact on the performance of GCG.

Finally, in Section 7.4, we compare GCG to SCIP and evidence that the generic branch-cut-and-price approach is competitive to an LP based branch-and-cut MIP solver for the regarded problem classes.

The computational environment and the test sets used for all computations presented in this Chapter are described in Section 3.3.

7.1 Impact of the Pricing Strategy

In Section 4.3, we compared different pricing strategies for Farkas and reduced cost pricing with respect to the solving process of the master problem at the root node. Now, we present results for the complete branch-and-price process for some of these settings. Since we are not able to create cutting planes for the regarded problem classes, it is just a branch-and-price and no branch-cut-and-price process.

Farkas pricing

In Section 4.3.1, it turned out, that for the solving process of the master problem at the root node, adding all variables of one problem, that cut off the dual ray performed remarkably better than adding just one variable. We performed computations to investigate whether this still holds for the branch-and-price process. The results are summarized in Table 7.1. We used the convexification approach for these computations, i.e., blocks were not aggregated, and branching on original variables with pseudocost variable selection rule was used.

We see that also for the branch-and-price process, adding all variables of one problem that are found in the pricing process performs better than adding just one of these variables. For the CPMP test sets, the solving time is reduced by five to twelve percent, for the vertex coloring test set it is decreased by four percent. The results for the RAP and bin packing test sets essentially stay the same. Therefore, we will change the default Farkas pricing settings in the next release of `GCG`.

	Farkas pricing: one var		Farkas pricing: one prob		
test set	nodes	time	nodes	time	Δ(%)
CPMP50S	44.9	12.8	42.7	12.1	-5
CPMP100S	587.1	184.7	529.1	165.5	-10
CPMP150S	847.6	493.5	801.4	464.6	-6
CPMP200S	1753.3	1243.9	1638.8	1094.8	-12
COLORING	60.9	88.2	55.2	85.0	-4
RAP32S	26.8	551.5	26.8	551.1	0
RAP64S	9.4	559.3	9.4	560.5	0
BINDATA1-N1S	3.6	0.2	3.6	0.2	0
BINDATA1-N2S	7.1	1.1	7.1	1.1	0
BINDATA1-N3S	13.2	6.7	13.2	6.7	0

Table 7.1. Farkas pricing performance for the default settings (adding just one variable per round) and when adding all variables found in one problem. We list the shifted geometric means of the number of nodes and the solving time for each test set. Furthermore, in column "Δ(%)", we state the percental change in the shifted geometric mean of the solving time.

Reduced cost pricing

We evaluated different reduced cost pricing strategies in Section 4.3.2. It turned out that the performance of a setting depends on the problem structure, i.e., the type of the linking constraints, the size of the master problem, the kind of pricing problems etc. In this thesis, we do not elaborate on "optimal" settings for single problem classes since we regard a generic approach. We rather try to find reasonable settings that seem to perform well for most types of problems. Anyway, GCG provides the parameters to adjust the pricing settings for a specific class of problems, if needed.

First, let us notice that most of the solving time is spent for solving the pricing problems. As a typical example we regard instance p50150-27 of test set CPMP150S, that is solved by GCG in 1325.9 seconds. 94% of the time (1248 seconds) is spent for pricing, most of this time is needed to solve the pricing problems. In particular, 1082 and 447,545 pricing problems are solved by Farkas and reduced cost pricing, respectively, consuming 5.9 and 1078.7 seconds, respectively. Besides verifying that reduced cost pricing is much more important for the overall solving process, we can see that over 80% of the total solving time is spent for solving the pricing problems. Similar numbers hold for the other CPMP instances. For the RAP instances, the pricing problem is harder and solving it consumes an even larger part of the total solving time.

Therefore, it typically pays off to add all variables with negative reduced cost that are found during the solving process of a pricing problem. This way, we profit from the effort as far as possible. However, adding too many variables may slow down the simplex method but for the problems we regarded in this thesis, this effect is very small compared to the time saved in the pricing routine. Hence, we do not limit the number of variables created per block in one pricing round.

The default settings impose a limit of 100 on the number of variables created per pricing round. We also tested some lower and higher numbers, but setting the limit to 100 performed best.

If we have just one pricing problem, like it is the case for the discretization approach with aggregated blocks, this corresponds to limiting the number of variables created for this pricing problem. In most cases, the number of 100 variables is not exceeded and all variables found by the pricing routine are added. However, if a huge number of variables is found in one pricing round, we limit the increase in the size of the RMP that way. Hence, the default settings seem to be reasonable for problems with aggregated blocks.

For the convexification approach or the discretization approach without aggregation of blocks, we additionally have to decide whether we solve all pricing problems or just a subset of them. For the default settings, we solve just as many pricing problems as needed to find 100 variables with negative reduced cost.

Stopping the pricing round after one pricing problem with negative op-

test set	default nodes	default time (outs)	all vars nodes	all vars time (outs)
RAP32S	26.8	551.5 (0)	32.1	609.7 (1)
RAP64S	9.4	559.3 (1)	9.3	524.7 (1)

Table 7.2. Performance of setting "all vars" for the reduced cost pricing compared to the default pricing settings on the RAP test sets for the branch-and-price process..

timum was solved and adding all variables with negative reduced cost computed by this pricing problem performed best for the master problem at the root node of the vertex coloring and bin packing instances, which have identical blocks. In Section 5.7 we showed that this pricing method performs well for these instances also for the branch-and-price process. However, due to the identity of the blocks, we suggest using the discretization approach with aggregation of blocks for these instances, anyway. For the problems with different blocks, these pricing settings performed much worse than the default settings.

For the master problem at the root node, solving all pricing problems and adding all variables found performed best for the RAP instances, but worse for the other ones (see Section 4.4.2). We tested whether this still holds for the branch-and-price process, the results are summarized in Table 7.2. For test set RAP64S, adding all variables is slightly superior, for test set RAP32S, it is slightly inferior. The time per node is decreased, in return, the number of nodes is increased. Hence, we can conclude that these settings do not dominate the default settings as well.

Anyhow, the question is whether the same pricing setting should be used for the reduced cost pricing at the root node as at all other nodes of the branch-and-bound tree. At the root node, when the reduced cost pricing process is started, we just obtained feasibility of the RMP using Farkas pricing, but the optimal solution to the RMP is far from being optimal for the master problem. So we probably have to perform many pricing rounds, thereby adding a big number of variables which then also serve as an adequate foundation for the following solving process.

At subsequent nodes, we do not have to solve the master problem from scratch, since we already solved the master problem at the father of the current node. Although the branching restrictions forbid this former solution, we probably have to perform just a few simplex iterations to restore optimality. Furthermore, the RMP already contains many variables created at the root node and other previously solved nodes, so we probably have to add just a few variables to the RMP.

This conjecture is confirmed by the results obtained in the previous chapters concerning Farkas and reduced cost pricing at the root node and the complete branch-and-price process (see Tables D.1 to D.26). We only regard the CPMP and RAP test sets, since for the other instances, the blocks should be aggregated, anyway.

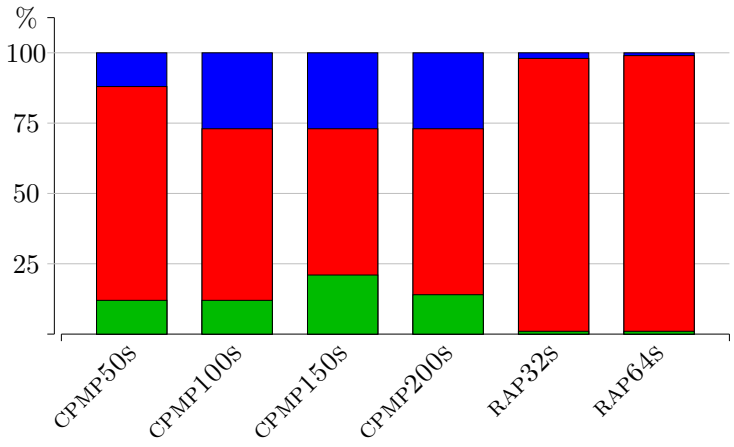


Figure 7.1: Quota of the average number variables created by Farkas pricing at the root node (■), reduced cost pricing at the root node (■), and after the root node (■).

Figure 7.1 pictures the quota of the variables that were created at the root node by Farkas and reduced cost pricing, respectively. Furthermore, we picture the part of the variables that were created at subsequent nodes. For the CPMP test sets, about 15% to 20% of the variables are created by Farkas pricing at the root node. For the RAP test sets, this ratio is much smaller since feasibility is obtained more easily (see Section 4.4.1).

For all test sets, the average number of variables created by reduced cost pricing at the root node exceeds the number of variables created at all subsequent nodes. In particular, several thousand variables are created at the root node on average for each instance while the average number of variables created per node in the following branch-and-price process ranges from two to six variables (see Figures 7.2 and 7.3). Furthermore, the more nodes are needed to solve the problems, the smaller is the average number of variables created per node after the root node. This is an indication for the decrease of the number of variables created at each node when going deeper into the branch-and-bound tree. It is consequential since the number of already created variables is higher for these nodes than for those treated at the beginning of the branch-and-price process.

The number of pricing rounds per node is also rather small after the root node, we need on average at most three pricing rounds per node, and, again, the average number of rounds is smaller for test sets that need more nodes. Hence, less than two variables are added per pricing round on average. In comparison, let us note that the reduced cost pricing process at the root node needs 20 to 70 rounds and in each of them, between 70 and 90 variables are created on average.

Therefore, we have to solve more pricing problems at subsequent nodes

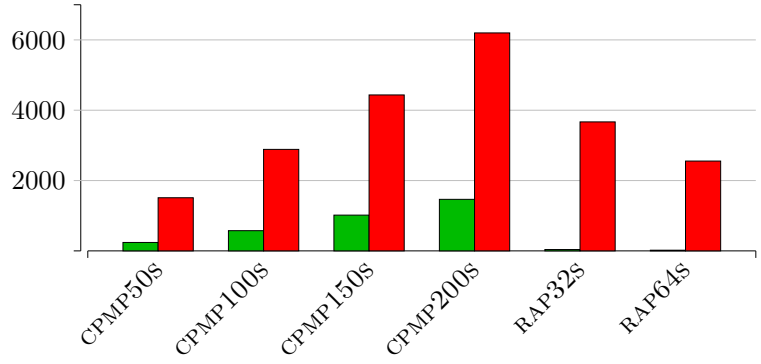


Figure 7.2: Shifted geometric mean of the number of variables created by Farkas pricing (■) and reduced cost pricing (■) at the root node.

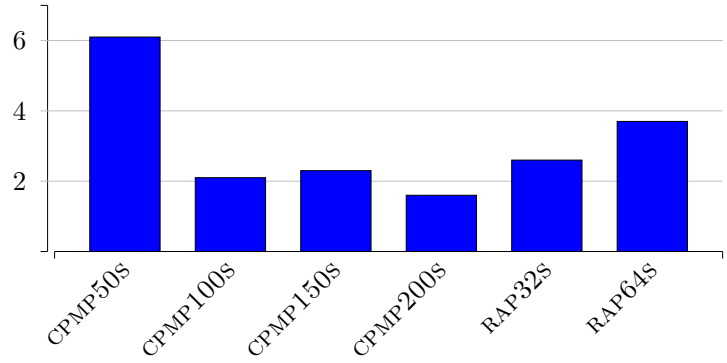


Figure 7.3: Shifted geometric mean of the number of variables created per node after the root node.

in order to find an improving variable. At the root node, we find for the CPMP instances about two variables per pricing problem that is solved in the reduced cost pricing process. For the RAP instances, this number is even higher. At subsequent nodes, we solve 8 to 150 times more pricing problems than we find variables. Again, instances that need more nodes need on average more problems to find one variable, which is of course also caused by the higher number of pricing problems for these instances.

Since the pricing process at the root node obviously differs highly from that at subsequent nodes, we decided to introduce additional parameters for the pricing process. The limits on the number of variables added per pricing round and the relative number of pricing problems solved per round can now be specified independently for the root node and subsequent nodes. In the following, the parameters M_r and R_r still represent the limits at the root node. Furthermore, we denote by $\tilde{M}_r$ the limit on the number of variables

	test set	default	20 vars	10%	no limit
nodes	CPMP50S	44.9	−2	0	−5
	CPMP100S	587.1	−1	−8	−10
	CPMP150S	847.6	+2	+6	−15
	CPMP200S	1753.3	−2	−3	+7
	RAP32S	26.8	−1	−4	+8
	RAP64S	9.4	0	−1	+3
	mean	171.1	−1	−2	−3
time	CPMP50S	12.8	−1	+9	−3
	CPMP100S	184.7	+3	+8	−6
	CPMP150S	493.5	+2	+12	−10
	CPMP200S	1243.9	0	+16	+3
	RAP32S	551.5	0	−10	+7
	RAP64S	559.3	−1	−9	+6
	mean	300.5	0	+3	−1

Table 7.3. Impact of different reduced cost pricing settings. At the root node, we use the default reduced cost pricing setting. We list the shifted geometric mean of the number of nodes and the solving time for using the same setting at subsequent nodes (first column). The other columns denote the percental changes in the shifted geometric mean of the values for the other settings. Positive values represent a deterioration, negative values an improvement.

created per round at subsequent nodes and by $\tilde{R}_r$ the relative limit on the number of pricing problems solved in each pricing round at subsequent nodes. Again, if we do not find a single variable before this limit is exceeded, the pricing round is continued until a problem with negative optimum was found and all variables of this problem are added (if their number does not exceed $\tilde{M}_r$). A listing of all parameters of GCG discussed in this thesis can be found in Appendix B.

We performed computational experiments concerning different values of the new parameters. They are summarized in Table 7.3. Further details can be looked up in Tables D.41 to D.46.

In the default settings, we choose $\tilde{M}_r = 100$ and $\tilde{R}_r = 1$, i.e., we do not distinguish between the pricing settings at the root node and those at all other nodes.

First, we studied whether setting a smaller absolute limit for the number of variables created in each round speeds up the solving process. Therefore, for the results presented in column "20 vars" of the table, we set $\tilde{M}_r = 20$ and $\tilde{R}_r = 1$. We mentioned that the average number of variables created per node ranges from two to six, but this is just an average value. There exist nodes where no further variable is found as well as nodes where more variables are created, especially those nodes treated first in the solving process. However, the "20 vars" setting does not change the average performance of GCG, we only have slight variations for the single test sets. This shows that there is

only a small number of nodes where many variables are created so that the limit of 20 variables created per pricing round is hardly ever exceeded.

Column "10%" presents the results for defining $\tilde{R}_r = 0.1$ and leaving $\tilde{M}_r = 100$ unchanged. The number of nodes is slightly decreased for the CPMP instances as well as for the RAP instances. The time is increased for the CPMP instances while being decreased for the RAP instances. For $\tilde{M}_r = \infty$ and $\tilde{R}_r = 0.1$ (column "no limit"), we obtain opposing results: The number of nodes and the solving time for the CPMP instances is decreased while these values are increased for the RAP instances.

This is a substantiation of our decision to introduce independent parameters for the root node on the one hand and all other nodes on the other hand. For the RAP instances, we have seen in Section 4.4 that adding all variables at the root node performs better than the default settings, while adding just a small number performs much worse. However, for the subsequent nodes, the dominance is completely reverted: Adding a small number is remarkably better than adding all variables. The CPMP instances profit from a higher limit in the branch-and-price process while a higher limit at the root node leads to a deterioration.

One reason for the aforementioned results might be the early termination of the pricing process, which is enabled per default and was thus used for these computations. By setting no limit on the number of variables we probably intensify its impact, as described in the following.

Suppose we are at some node in the branch-and-bound tree and we have to solve several pricing problems before finding a variable that can improve the current solution. Without early termination, we have to perform another pricing round after adding this variable, regardless of whether we continue solving the current pricing round and potentially add further variables, or not. In particular, even if there do not exist further variables, we have to solve all pricing problems to prove this in the next pricing round.

When enabling early termination, this changes: If we stop the pricing round, then we cannot compute a valid lower bound on the optimal objective value of the master problem. In order to do this, we need the optimal solution values of all pricing problems (see Section 4.2). Hence, we have to resolve the RMP and perform another pricing round. On the other hand, if we continue the current pricing round and solve all pricing problems to optimality, we can profit from that, even if we do not find further variables, since we get a valid lower bound. With this bound, we can possibly stop the pricing process at the current node due to early termination. It seems that this pays off for the CPMP instances since imposing no limit on the number of variables added and therefore, solving all pricing problems in each round reduces the solving time in the shifted geometric mean.

However, the RAP instances have rather high optimal objective function values so that even a small relative gap between dual bound and objective function value of the RMP does not necessarily correspond to a small absolute gap that could allow the early termination of the pricing process. Hence, it

often does not pay off to solve all pricing problems, especially since solving a pricing problem takes much longer for the RAP instances than for the CPMP instances. Therefore, the impact of early termination is smaller for the RAP instances and is probably overweighted by the additional effort for solving all pricing problems.

To sum up, we did not find a pricing strategy that dominates the default settings for all problems classes or that performs at least considerably better on average. We found strategies that are superior to the default settings for one class of problems but in return, these strategies are inferior to the default settings for the other class of problems. This shows again, that the performance of reduced cost pricing settings are highly problem dependent. The default settings, i.e., adding at most 100 variables per pricing round seems to be reasonable at least for those problems that we regarded in this thesis.

Finally, let us note that we would have expected that it is more efficient to set the limit on the number of variables added in each round depending on the size of the problem and the number of pricing problems. However, our computational experiments—we listed only the most interesting ones here—did not support this conjecture; an absolute limit of 100 performed rather well on average. In some sense, this is actually consequential, since we look for variables that can enter the basis in the next simplex iteration. The simplex method, however, does not consider the size of the problem for one iteration, it always substitutes just one basic variable by one nonbasic variable. The size of the problem only affects the number of simplex iterations. Although the simplex method can potentially perform multiple iterations between two pricing rounds when we add multiple variables, the reduced costs of the newly created variables change with each iteration. Therefore, variables that are created because they have negative reduced costs with respect to the current solution of the master problem will possibly not be able to improve the solution after some simplex steps have been performed. Hence, when adding a huge number of variables for a huge problem, the reduced costs can have completely changed after a small number of simplex iterations and the remaining variables do actually only slow down the simplex method. Therefore an absolute limit on the number of variables added per round seems to be reasonable.

7.2 Problem Specific Pricing Solvers

GCG provides the possibility to include problem specific solvers for solving the pricing problems. Since most of the time is spent for the solving process of the pricing problems, using a problem specific pricing solver can improve the performance of GCG by far. We demonstrate this using the example of a knapsack solver for the CPMP instances. Our implementation of the knapsack solver uses a method provided by the knapsack constraint handler

test set	MIP solver nodes	MIP solver time (outs)	knapsack solver nodes	knapsack solver time (outs)
CPMP50S	44.9	12.8 (0)	42.1	1.8 (0)
CPMP100S	587.1	184.7 (1)	491.6	37.8 (0)
CPMP150S	847.6	493.5 (5)	1228.6	253.8 (1)
CPMP200S	1753.3	1243.9 (3)	2338.0	519.3 (0)
mean	461.8	220.1 (9)	515.1	84.3 (1)

Table 7.4. Performance effect of using a knapsack solver to solve the pricing problems for the CPMP test sets. We list the shifted geometric means of the number of nodes and the total solving time for GCG when using the MIP pricing problem solver (first two columns) and when using a specialized knapsack solver (last two columns).

of SCIP that solves a knapsack problem to proven optimality with a dynamic programming method.

The results of our computations (see Table 7.4 and Tables D.47 to D.50 for more details) report a decrease of about 62% in the shifted geometric mean of the solving time. Furthermore, when using the knapsack solver, GCG was able to solve all instances but one, while nine instances could no be solved within the time limit of one hour when using the the MIP pricing solver.

For the larger instances, however, the number of nodes is increased, which can be explained by the fact that the pricing solver returns only one optimal solution so that the number of variables created at one node is typically smaller. Since certain variables are needed to construct an optimal solution, the solving process cannot finish before these variables are added. In total, the number of variables created per instance is similar for both pricing solvers, the knapsack solver just needs more nodes to create these variables. If we would run primal heuristics not only on the extended, but also on the original problem, we could find an optimal solution without creating specific variables in the master problem. Hence, a further subject of research should be to design and implement primal heuristics that run on the original problem.

However, even without further heuristics, the knapsack pricing solver reduces the computational effort by far.

7.3 Selected Accelaration Strategies

Apart from the pricing strategy, we mentioned several advanced features that have an impact on the performance of the branch-cut-and-price solver GCG. In this subsection we present computational results concerning their impact on the solving process of the instances of the CPMP test sets. We used these instances, since they are the ones that need the highest number of branch-and-bound nodes to be solved to optimality. Therefore, the impact of these features that are primarily used to speed up the solving process after the root node is pictured best by these instances.

Table 7.5 summarizes the results: We list the change in the shifted geo-

	test set	pseudo	enfo.pric.	proper	early	heurs	all
nodes	CPMP50S	+84	−1	+1	−5	−2	+93
	CPMP100S	+234	+2	−2	−8	−3	+224
	CPMP150S	+161	−2	+7	−6	+4	+104
	CPMP200S	+218	+10	−5	−15	+1	+130
	mean	+163	+2	0	−9	0	+129
time	CPMP50S	+62	−2	+1	+4	−1	+87
	CPMP100S	+154	−2	−1	+8	−1	+216
	CPMP150S	+86	+2	+7	+10	+5	+100
	CPMP200S	+139	+8	−4	+9	+2	+154
	mean	+100	+2	+1	+7	+1	+124

Table 7.5. Performance effect of disabling selected acceleration strategies incorporated in GCG. The values denote the percental change in the shifted geometric mean of the number of nodes (top) and the total solving time (bottom) compared to the default settings in which all these features are activated. Positive values represent a deterioration, negative values an improvement.

metric means of branch-and-bound nodes and total solving time for the four small CPMP test sets when turning a single feature off. The columns are labeled by the feature that is disabled. The results for the default settings, where all these features are enabled, serve as reference values. More details can be found in Tables D.47 to D.50.

First, we repeat the effect of the branching strategy, in particular the impact of the variable selection rule and the impact of how branching restrictions are enforced (see Chapter 5). Using the most fractional variable selection rule (column "pseudo") doubles the average solving time. The enforcement of the branching restrictions has no big impact, enforcing them in the master problem (column "enfo.pric.") increases the shifted geometric means of the nodes and the running time by 2%, which is no significant change and could also be due to random variations.

The same holds for the enforcement of proper variables, i.e., performing domain propagation in the original problem and enforcing the domain changes in the pricing problem (see Section 5.6). When it is disabled (column "proper") the average number of nodes does not change, the solving time is increased by just one percent.

The early termination of the pricing process has a bigger impact. As we have mentioned in Section 4.3.3, using early termination decreases the effectiveness of pseudocosts, since the increase in the dual bound of a node is not computed exactly. Therefore, the average number of nodes is decreased when disabling early termination (column "early") and computing the exact lower bound of all nodes. Nevertheless, the increased time needed for the pricing process outweights the savings due to the smaller number of nodes, so the shifted geometric mean of the solving time is increased by 7% when disabling early termination. Furthermore, if we would use the most fractional

variable selection rule rather than the pseudocost rule (although the former is inferior to the latter), the performance effect of early termination would probably be higher. Disabling early termination and obtaining more reliable pseudocosts would then not help the branching rule to decrease the number of nodes, hence the time saved by early termination in the pricing process would pay off even more.

Additionally, we tested the impact of using the convexification approach instead of the discretization approach. Since the master problem essentially stays the same, this actually just disables the use of primal heuristics in the extended SCIP instance. The results are listed in column "heurs". We can see that the primal heuristics in the extended problem do not have a big impact on the performance, most of the primal solutions are found as an integral solution to the RMP, anyway. This shows that the default SCIP heuristics are not very effective on the extended problem and we should think about implementing further primal heuristics that take into account the branch-cut-and-price approach and use information of both the original es well as the extended problem.

Finally, turning all these features off (column "all") results in an increase of more than 120% for the shifted geometric means of both the number of nodes as well as the total running time. The smaller increase in the number of nodes compared to disabling just the pseudocost variable selection rule follows from the slowdown due to disabling the other features so that a smaller number of nodes could be solved in case the timelimit was hit.

To summarize, the pseudocost variable selection rule has by far the biggest impact on the performance of GCG for the regarded test sets of CPMP instances, followed by the early termination of the pricing process. The other features have a rather small impact on the performance of GCG.

7.4 Comparison to SCIP

In the last sections, we just tuned the performance of GCG. In this section, we finally answer the question whether the generic branch-cut-and-price approach is competitive to an LP based branch-and-cut MIP solver. Since we based our implementation on the branch-cut-and-price framework SCIP, and SCIP with default plugins is a state-of-the-art branch-and-cut based MIP solver, we compare this MIP solver to GCG on the complete test sets of the four problem classes that we regard in this thesis. For both GCG and SCIP, we used the default settings. Like for all computations presented in this thesis, we used GCG version 0.7.0 and SCIP version 1.2.0.5. Further information about the test sets and the computational environment can be looked up in Section 3.3.

test set	items	nodes total	nodes mean	time total	time mean
BINDATA1-N1	50	520	1.7	196.5	1.2
BINDATA1-N2	100	1681	3.4	922.7	4.4
BINDATA1-N3	200	6222	11.3	4202.9	20.3

Table 7.6. Computational results for GCG on the 180 instances of each size in the complete bin packing test sets.

Identical Blocks: Bin Packing and Vertex Coloring

Bin packing and vertex coloring instances have identical blocks and a set partitioning master problem, so the variables were aggregated and Ryan and Foster's branching scheme was used. It is well-known that the Dantzig-Wolfe decomposition of the bin packing problem leads to strong dual bounds [60], so we were able to solve all 540 instances (180 instances with 50, 100, and 200 items, respectively) in less than 90 minutes altogether. For each number of items, GCG solves the whole test set faster than SCIP solves the first instance of the test set. The results for GCG are summarized in Table 7.6.

Also for our vertex coloring test set, GCG performs remarkably better than SCIP which solved no more than twelve instances within the time limit of one hour per instance. GCG was able to solve 33 of the 39 instances. Furthermore, SCIP needs much more nodes than GCG and also the shifted geometric mean of the solving time per instance is about six times higher for SCIP than for GCG. These results are summarized in Table 7.7, more details can be found in Table D.51.

SCIP gap	SCIP nodes	SCIP time (outs)	GCG gap	GCG nodes	GCG time (outs)
19.5	3256.9	1073.3 (27)	8.7	64.3	176.1 (6)

Table 7.7. Comparison of SCIP and GCG on the complete coloring test set (COLORING-ALL). We list the shifted geometric means of the final gap, the number of branch-and-bound nodes and the solving time. Following the time, in brackets, we list the number of timeouts.

Different Blocks: The Capacited p-Median Problem

The capacitated p-median problem (CPMP) has multiple, different blocks, so they cannot be aggregated and branching on original variables is used. In the last sections, we already presented the impact of different settings on the performance on GCG. Now, we want to compare it to SCIP, so we performed computational tests on the complete CPMP test sets, each of which consists of 40 instances. We used default settings for both GCG and SCIP and imposed a time limit of one hour. Table 7.8 summarizes the results, more details can be found in Tables D.52 to D.55.

	SCIP			GCG		
test set	gap	nodes	time (outs)	gap	nodes	time (outs)
CPMP50	0.1	1652.9	25.8 (1)	0.0	80.9	23.9 (0)
CPMP100	0.8	28557.8	436.9 (11)	0.5	929.5	298.0 (5)
CPMP150	2.6	22250.3	846.8 (19)	0.6	979.1	513.9 (8)
CPMP200	8.0	28550.0	2350.7 (28)	11.4	2248.2	1728.1 (23)
mean	2.8	13172.8	414.2 (59)	3.0	650.9	302.3 (36)

Table 7.8. Comparison of SCIP and GCG on the complete CPMP test sets. For each test set, we list the shifted geometric means of the final gap, the number of branch-and-bound nodes and the solving time. Following the time, in brackets, we list the number of timeouts.

The tighter dual bound obtained by the master problem results in a decrease of the number of nodes: In the shifted geometric mean, GCG needs just the twentieth part of the nodes needed by SCIP. In return, the solving time per node is much higher due to the column generation process. Nevertheless, the average time needed by SCIP is more than one third higher than the time spent by GCG. Furthermore, the number of instances that could not be solved within the time limit is about two thirds higher for SCIP than for GCG.

On the other hand, the shifted geometric mean of the final gap is higher for GCG than for SCIP. However, this is only caused by five instances of test set CPMP200, for which GCG did not find a solution. In this case, we charged a gap of 100%, which increased the average gap for this test set. We see this as a motivation to develop better primal heuristics for the decomposition approach. Currently, we just run the default SCIP heuristics on the extended formulation; primal heuristics that exploit the original formulation or even both formulations could further increase the performance of GCG.

Finally, let us clarify that we used the MIP pricing problem solver for these computations. As mentioned in Section 7.2, we can improve the performance of GCG by far for the CPMP instances by using the knapsack pricing problem solver. Therefore, when using this solver, GCG would probably be much faster and would outperform SCIP even more.

No Block Structure: The Resource Allocation Problem

The resource allocation problem (RAP) does not possess a block structure from the beginning. However, it has a very special form (see Figure C.1), due to which we can transfer it into a bordered block diagonal form by assigning constraints to blocks and copying variables that are contained in constraints of different blocks. Furthermore, we have to add constraints that force all copies of a variable to have the same value. This results in a bordered block diagonal structure of the constraint matrix. A detailed description of the transformation process can be found in Appendix C.

We ran benchmarks to assess the effectiveness of the Dantzig-Wolfe approach for these problems. We studied two sizes of blocks, grouping either 32

solver	gap	nodes	time	timeouts
SCIP	1.4	97564.0	2772.4	65
GCG (32)	0.0	32.2	727.4	14
GCG (64)	0.8	11.5	670.7	14

Table 7.9. Comparison of SCIP and GCG on the complete RMP test set. We grouped 32 (second row) and 64 (third row) constraints per block for solving the instances with GCG. We list the shifted geometric means of the final gap, the number of branch-and-bound nodes and the solving time. The timeouts are given in absolute numbers.

or 64 constraints per block. A summary of the results for both these alternatives is given in Table 7.9 as well as the results for solving these instances with SCIP. A detailed listing of the results can be found in Table D.56. We imposed a timelimit of one hour per instance.

Grouping more constraints per block results in less pricing problems that are in return harder to solve. Therefore, the time needed to solve the master problem is increased. This is weighted up by a tighter dual bound. Hence, when grouping 64 constraints instead of 32, the shifted geometric mean of the number of nodes is decreased by more than 60%, resulting in a slightly decreased average solving time. For both variants, GCG is able to solve 56 of 70 instances within the time limit. The shifted geometric mean of the gap is less than 0.05% when grouping 32 constraints, when grouping 64 constraints, it accounts 0.8%. However, this is caused by one single instance for which the master problem of the root node could not be solved within one hour, so that no feasible dual bound was computed and the trivial dual bound was used to compute the gap.

This is a difference between row generation (branch-and-cut) and column generation (branch-and-price). While for a branch-and-cut based algorithm the optimal LP value is a valid dual bound after each separation round, the optimal value of the RMP is not a valid dual bound after each pricing round, but only when the column generation process has finished. Hence, in case we do not finish the solving process of the root node within the time limit, we do not obtain a valid dual bound. Only if we were able to compute the bound LB_{RMP} (see Section 4.2) in one of the previous pricing rounds, this gives us a valid dual bound but this bound is typically rather weak and gets stronger only when the column generation process is nearly finished.

To sum up, GCG performs better than SCIP for both variants. SCIP was not able to solve more than five instances to optimality, the shifted geometric mean of the final gap accounts 1.4% and the number of nodes is far higher than for both GCG variants. The total solving time is about four times higher for SCIP than for GCG, and it would probably be even worse if we would not have imposed the time limit.

This shows that the branch-and-price approach can successfully be applied even to problems that do not have a bordered block diagonal structure from the beginning. When obtaining the block diagonal structure by a trans-

formation, a further direction of research should be to investigate how many constraints should be assigned to one block. As aforementioned, assigning more constraints to one block makes the pricing problems harder to solve, but leads to better dual bounds. These properties have to be weighted up against each other.

Chapter 8

Summary, Conclusions and Outlook

In this thesis, we presented the theoretical background, implementational details and computational results concerning the generic branch-cut-and-price solver GCG which was developed by the author of this thesis.

The foundation for this is the Dantzig-Wolfe decomposition for MIPs which we presented in Chapter 2. It allows us to reformulate a general MIP, the *original problem*, into a problem with a huge number of variables, the *extended problem*. A subset of the constraints is transferred to the extended problem, the remaining constraints are implicitly treated and become part of the definition of the variables of the extended problem. We described two ways to perform this reformulation, the *convexification* and the *discretization* approach. The former is the more general one while the latter can reduce symmetry, if contained in the original problem. For both approaches, the dual bound—obtained by solving the relaxation of the extended problem—equals the Lagrangian dual bound and is typically stronger than the bound obtained by the LP relaxation. This is one of the main motivations for using the Dantzig-Wolfe decomposition and the branch-cut-and-price approach.

The branch-cut-and-price solver GCG is able to handle both approaches. The user has to provide the original problem and to define its structure, i.e., which of the constraints are treated implicitly and which of them are transferred to the extended problem. GCG then performs the reformulation and solves both problems—the original and the extended problem—simultaneously. The general structure of GCG and the solving process was presented in Chapter 3, as well as a short overview of the non-commercial branch-cut-and-price framework SCIP on which our implementation is based. SCIP with default plugins is a state-of-the-art non-commercial MIP solver and was therefore also used to compare the branch-cut-and-price solver GCG to an LP based branch-and-cut MIP solver.

Since the extended problem has a huge number of variables, we do not solve its LP relaxation, the *master problem*, explicitly, but use the concept of

column generation in order to do so. That is, we regard a *restricted master problem (RMP)* which contains just a subset of the variables and add variables to the problem during the solving process whenever needed. Variables which potentially improve the solution are computed by a *pricing* process. For this, we solve one or more pricing problems which are given as MIPs and contain the constraints that were not transferred to the extended problem. The concept of column generation, its application to the presented approach, and implementational details were presented in Chapter 4. Furthermore, we performed computational experiments to assess the effectiveness of various pricing strategies.

In order to compute an optimal integer solution, we use a branch-and-bound process. In combination with column generation, we thus obtain a *branch-and-price* algorithm. In Chapter 5, we described several branching rules that fit into the branch-and-price approach. We implemented two of them, a branching on variables of the original problem and Ryan and Foster's branching scheme. The former one is used by most LP based branch-and-cut MIP solvers and can easily be transferred to the branch-cut-and-price approach. Ryan and Foster's scheme can only be used for problems with a special structure in the extended problem; it can reduce symmetry when used in combination with the discretization approach. Furthermore, we tested whether the concept of *pseudocosts*, that is used by most state-of-the-art branch-and-cut MIP solvers to select the variable on which branching is performed, can successfully be transferred to the branch-cut-and-price approach. It turned out that this is the case and using pseudocosts halves the shifted geometric mean of the solving time for the class of capacited p-median problems. The other class of problems for which we tested the impact of pseudocosts, the RAP instances, are solved after a very small number of nodes so that the use of pseudocosts does not pay off.

In Chapter 6, we described how the branch-and-price method can be turned into a *branch-cut-and-price* method by performing cutting plane separation. Cutting planes can be derived from either the original or the extended problem. We named pros and cons of both variants and described our implementation of the former one. For general MIPs, we were able to find cutting planes this way, but the problems regarded in this thesis have a structure which does not allow the employed cutting plane separators to find cutting planes in the original problem. We used this as an occasion to compare the dual bounds at the root node obtained by `GCG` and `SCIP`. Even though `SCIP` tightened the dual bound by adding cutting planes, `GCG` obtained on average a remarkably tighter dual bound for all regarded problems classes.

Finally, in Chapter 7, we presented some computational results concerning the overall performance of `GCG`. We tested the performance of pricing strategies for the branch-cut-and-price process instead of doing so just for the column generation process at the root node as it was done in Chapter 4. Additionally, we pictured the impact of using a *problem specific solver* to solve the pricing problems rather than solving them with a general MIP

solver. GCG provides the possibility to include such a solver and we did so for a knapsack solver which reduced the shifted geometric mean of the solving time by more than 60% for the class of capacited p-median problems. Furthermore, we reviewed some acceleration strategies, e.g., pseudocosts, early termination and the enforcement of proper variables, and their impact on the performance of GCG.

Additionally, we compared the performance of GCG for the four problems classes regarded in this thesis with the performance of SCIP. These problems were previously known to fit well into the branch-cut-and-price approach. However, former implementations for the problems classes were problem specific. They used knowledge about the problem such as specialized pricing solvers that can improve the performance by far as aforementioned. Therefore, one of the main goals of this thesis was to investigate whether the branch-cut-and-price approach in the implemented generic form is still competitive to a state-of-the-art branch-and-cut MIP solver for these problems. It turned out that the generic branch-cut-and-price solver GCG still outperforms SCIP although it does not use any problem specific knowledge except for the structure of the problem.

Our main contribution is the development of the generic branch-cut-and-price solver GCG.

It provides the possibility to easily check whether a branch-cut-and-price approach for a problem is promising. Instead implementing a branch-cut-and-price solver form scratch or at least adding some problem specific plugins into a branch-cut-and-price framework, one only needs to define the structure for the decomposition in an additional file. The problem is then automatically reformulated by GCG and solved; pricing, branching and cutting plane separation is performed in a generic way. If the approach turns out to be promising, the performance of GCG can further be enhanced by replacing generic parts of the solver by problem specific ones.

We studied how some concepts that are successfully used in LP based branch-and-cut MIP solvers can be transferred to the branch-cut-and-price approach, using the example of the capacited p-median problem. By performing domain propagation in the original problem and transferring its results into the extended problem, we enforce *proper variables* [96]. This slightly improved the performance of GCG. It turned out that using *pseudocosts* to select the variable to branch on has a big impact: it essentially halved the solving time. Furthermore, we tried to separate cutting planes in the original problem, but did not find any for the problems classes regarded in this thesis. It seems that the reason for this is the special structure of the problems. However, for more general problems, cutting planes were found and could improve the solving capabilities of GCG.

For the RAP instances, we demonstrated that the branch-cut-and-price approach can perform well even if the problem does not have a bordered block diagonal structure right from the beginning. Since this problem has another special structure, we see this as an encouragement that this approach should

be tested for further problems with other special structures.

There are still many concepts that should be investigated and integrated to further enhance the solving capabilities of GCG.

For the column generation process, apart from incorporating further problem specific pricing solvers, stabilization of the dual variables [64, 57] is one of the most promising concepts. During the column generation process, the dual variables may highly oscillate. Through stabilization techniques, e.g., penalizing big differences to the value of the dual variable in the last solution, this is reduced which typically improves the column generation performance.

For the branching process, it remains to be investigated whether reliability branching—one of the most effective branching rules in LP based branch-and-cut—can successfully be transferred to the branch-cut-and-price approach. Furthermore, some more of the branching rules described in Chapter 5 should be incorporating in GCG in future releases.

Also the separation capabilities of GCG could be further enhanced by enabling separation of cutting planes in the extended problem and adding further separators for the original problem that do not need any information about the LP basis.

Finally, one of the primary targets for the future will be the development of further primal heuristics. On the one hand, we need heuristics that work on the original formulation and do not need to solve its LP relaxation. On the other hand, further heuristics in the extended problem that consider the branch-cut-and-price approach could be investigated. For example, after finishing the column generation process at a node, one could try to solve the corresponding extended problem, that contains just the subset of variables also contained in the RMP, to optimality with a branch-and-cut approach without further pricing of variables.

The most challenging question, however, is to investigate whether there really is a significant share of problems for which the generic branch-cut-and-price approach is more effective than an LP based branch-and-cut algorithm, even when one does not know about a possibly contained structure. This requires detecting whether it may pay off to reformulate the given MIP and to decide on how the reformulation is done.

Appendix A

Zusammenfassung (German Summary)

Viele Optimierungsprobleme lassen sich als gemischt-ganzzahlige Programme (MIPs) fomulieren. Obwohl das Lösen von MIPs $\mathcal{NP}$-schwer ist [86], lassen sich viele dieser Probleme von aktuellen MIP-Lösern in angemessener Zeit lösen (siehe z.B. Koch [55]). Die meisten dieser Löser basieren auf einem Branch-and-Cut-Ansatz, der eine Kombination aus der exakten Branch-and-Bound-Methode und Schnittebenenverfahren ist.

In dieser Arbeit haben wir eine andere Methode zum Lösen von MIPs betrachtet, die Branch-Cut-and-Price-Methode. Genau wie der Branch-and-Cut-Ansatz kombiniert sie Branch-and-Bound und Schnittebenenverfahren, zusätzlich verwendet sie jedoch noch das Konzept der Spaltengenerierung. Ihr Erfolg basiert auf einer Dekomposition des Problems in ein koordinierendes Problem und ein oder mehrere Subprobleme. Das koordinierende Problem enthält exponentiell viele Variablen, die jedoch nicht explizit behandelt werden, sondern implizit betrachtet und nur dann per Spaltengenerierung zum Problem hinzugefügt werden, wenn sie dessen aktuelle Lösung verbessern können. Um solche Variablen zu finden, werden die Subprobleme gelöst, was oft sehr effektiv mit einem Algorithmus erfolgen kann, der eine spezielle Struktur der Probleme ausnutzt.

Diese Arbeit behandelt den vom Autor entwickelten generischen Branch-Cut-and-Price-Löser `GCG`.

Nach grundlegenden Definitionen und einem kurzen Überblick über die Geschichte des Branch-Cut-and-Price-Ansatzes in Kapitel 1 stellen wir in Kapitel 2 die Dantzig-Wolfe-Dekomposition für MIPs vor, auf der unsere Implementation beruht.

Im 3. Kapitel präsentieren wir zunächst das Branch-Cut-and-Price-*Framework* `SCIP`, auf dem unser Branch-Cut-and-Price-*Löser* `GCG` aufbaut. Danach beschreiben wir die Struktur und den allgemeinen Lösungsablauf des Lösers `GCG`. Er benötigt neben dem originalen Problem weitere Informationen bezüglich dessen Struktur, anhand derer eine automatische Dekomposition

durchgeführt wird. Im Anschluß werden sowohl das originale als auch das durch die Dekomposition erhaltene erweiterte Problem simultan gelöst.

Die folgenden drei Kapitel behandeln die wichtigsten Bestandteile des Branch-Cut-and-Price-Lösers GCG. In Kapitel 4 widmen wir uns dem Lösungsprozess der LP-Relaxierung des erweiterten Problems mit Hilfe von Spaltengenerierung. Der Branching-Prozess ist Thema des 5. Kapitels, während wir in Kapitel 6 die Generierung von Schnittebenen behandeln. In jedem dieser Kapitel präsentieren wir zunächst einen Überblick der zugrunde liegenden Theorie, legen dann implementatorische Entscheidungen dar und schließen mit Ergebnissen unserer Testrechnungen sowie deren Auswertung.

Im 7. Kapitel präsentieren wir weitere Testergebnisse. Wir betrachten verschiedene Strategien bezüglich des Spaltengenerierungsprozesses. Nachdem wir einige Varianten bereits im 4. Kapitel aufgrund ihrer schlechten Effektivität ausschließen konnten, zeigt sich, dass keine der anderen getesteten Varianten eindeutig überlegen ist. Die standardmäßig verwendete Strategie liefert im Allgemeinen gute Ergebnisse, es gibt jedoch auch Alternativen, die auf einzelnen Klassen von Problemen überlegen, auf anderen dafür jedoch unterlegen sind. Dies zeigt, dass die Effektivität dieser Strategien sehr stark vom Problem und seiner Struktur abhängt. Um die Struktur der Subprobleme auszunutzen, bietet GCG die Möglichkeit, *problemspezifische Lösungsmethoden* einzubinden. Wird diese Möglichkeit nicht genutzt, werden die Subprobleme für die Spaltengenerierung, aufgrund des von GCG verfolgten generischen Ansatzes, standardmäßig als allgemeines MIP mit Hilfe von SCIP gelöst. Wir testen den Einfluss einer problemspezifischen Lösungsmethode auf die Rechenzeit für eine der betrachteten Klassen von Problem und erzielen dadurch eine Beschleunigung von ca. 160%.

Des Weiteren untersuchen wir den Effekt einiger weiterer Methoden, die den Lösungsprozess beschleunigen sollen. Es ergibt sich, dass die Verwendung von *Pseudokosten* im Zusammenhang mit dem Branching-Prozess eine ähnliche Verbesserung der Leistung bewirkt, wie sie schon für Branch-and-Cut-Löser gezeigt wurde: Die durchschnittliche Lösungszeit für die betrachtete Klasse von Problemen wird halbiert. Das *frühzeitige Beenden* des Spaltengenerierungsprozesses unter bestimmten Umständen führt immerhin noch zu einer Reduzierung der durchschnittlichen Lösungszeit von ca. 7%. Die weiteren getesteten Methoden haben wenig bis gar keinen Einfluss auf die durchschnittliche Rechenzeit.

Abschließend ziehen wir einen Vergleich zwischen dem Branch-and-Cut-Löser SCIP und dem Branch-Cut-and-Price-Löser GCG. Die Eignung der verwendeten Probleme für den Branch-Cut-and-Price-Ansatz war schon vorher aufgrund problemspezifischer Implementationen bekannt. Wir zeigen, dass GCG für diese Probleme – trotz des generischen Ansatzes – bessere Ergebnisse als ein aktueller Branch-and-Cut-basierter MIP Löser erzielt.

Wir sehen das als Motivation, diesen Ansatz in Zukunft weiter zu verfolgen und den generischen Branch-Cut-and-Price Löser GCG weiter auszubauen.

Appendix B

Notation and List of Parameters

We review the most important symbols used in this thesis. We start with the most important general mathematical notation. After that, we list the symbols used in the context of the Dantzig-Wolfe decomposition and their meaning. Finally, we present the parameters that can be used to tune `GCG`, the corresponding symbol used in the thesis and their effect.

General Mathematical Notation

Symbol	Definition
$\mathbb{N}$	the natural numbers $\mathbb{N} = \{1, 2, 3, \dots\}$
$\mathbb{Z}$	the integer numbers $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
$\mathbb{Z}_+$	the non-negative integer numbers $\mathbb{Z}_+ = \{0, 1, 2, 3, \dots\}$
$\mathbb{Q}$	the rational numbers
$\mathbb{Q}_+$	the non-negative rational numbers: $\mathbb{Q}_+ = \{x \in \mathbb{Q} \mid x \geq 0\}$
$\mathbb{R}$	the real numbers
$\mathbb{R}_+$	the non-negative real numbers: $\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$
$[n]$	the set $[n] := \{1, \dots, n\}$ for $n \in \mathbb{N}$

Symbols used in in the context of the Dantzig-Wolfe decomposition

Symbol	Definition
K	Number of blocks in the original problem.
x^k	Variable vector corresponding to block $k \in [K]$.
n_k	Number of variables corresponding to block $k \in [K]$.
$n_k^\star$	Number of integer variables corresponding to block $k \in [K]$.
c_k	Objective function coefficients corresponding to x^k.
m_A	Number of linking constraints.
m_k	Number of structural constraints of block $k \in [K]$.
A^k	Coefficient matrix of x^k w. r. t. the linking constraints.
b	Right-hand side of the linking constraints.
D^k	Coefficient matrix of x^k w. r. t. the strcutural constraints of block $k \in [K]$.
d^k	Right-hand side of the strcutural constraints of block $k \in [K]$.
X_k	The set of solutions x^k that fulfill the integrality restrictions as well as the structural constraints of block $k \in [K]$.
$conv(X_k)$	The convex hull of X_k.
P_k	Set of points represented by variables in the extended problem (extreme points for the convexification approach, extreme points and some inner points for the discretization approach).
R_k	Set of rays represented by variables in the extended problem.
RMP	The restricted master problem.
$\bar{P}_k$	Set of points with corresponding variables contained in the RMP.
$\bar{R}_k$	Set of rays with corresponding variables contained in the RMP.
λ^k	Vector of variables corresponding to points and rays of block $k \in [K]$ (or a subset of them for the RMP).
λ_p^k	Variable corresponding to point $p \in P_k$ (or $p \in \bar{P}_k$ for the RMP).
λ_r^k	Variable corresponding to ray $r \in R_k$ (or $r \in \bar{R}_k$ for the RMP).
c_p^k	Coefficient of variable λ_p^k in the objective function.
a_p^k	Coefficient of variable λ_p^k in the linking constraints.
c_r^k	Coefficient of variable λ_r^k in the objective function.
a_r^k	Coefficient of variable λ_r^k in the linking constraints.
L	Number of identity classes (w.l.o.g. represented by the first L blocks).
K_ℓ	Set of blocks that are identical to block $\ell \in [L]$ (including ℓ).
$\tilde{\lambda}_p^\ell$	Aggregated variable of identity class $\ell \in [L]$ corresponding to a point $p \in P_\ell$.
$\tilde{\lambda}_r^\ell$	Aggregated variable of identity class $\ell \in [L]$ corresponding to a ray $r \in R_\ell$.
π	Dual variables corresponding to the linking constrains in the RMP.
ρ	Dual variables corresponding to the convexity constrains in the RMP.
$\bar{c}_k^\star$	Optimal objective value of the pricing problem corresponding to block $k \in [K]$.

Pricing Parameters

Parameter	Symbol	Description
`maxvarsroundfarkas`	M_f	Maximum number of variables created in one Farkas pricing round.
`mipsrelfarkas`	R_f	Relative number of pricing problems solved in one Farkas pricing round. If this does not suffice to find new variables, pricing is continued until a problem was solved that gives rise to new variables.
`maxvarsroundredcostroot`	M_r	Maximum number of variables created in one reduced cost pricing round (at first for all nodes, since Chapter 7 only for the root node).
`mipsrelredcostroot`	R_r	Relative number of pricing problems solved in one reduced cost pricing round (at first for all nodes, since Chapter 7 only for the root node). If this does not suffice to find new variables, pricing is continued until a problem was solved that gives rise to new variables.
`maxvarsroundredcost`	$\tilde{M}_r$	Maximum number of variables created in one reduced cost pricing round at nodes after the root node.
`mipsrelredcost`	$\tilde{R}_r$	Relative number of pricing problems solved in one reduced cost pricing round at nodes after the root node. If this does not suffice to find new variables, pricing is continued until a problem was solved that gives rise to new variables.
`maxsolsprob`	M_p	Maximum number of variables created per pricing problem in a pricing round.
`maxroundsredcost`	*maxrounds*	Maximum number of reduced cost pricing rounds at a node.
`sorting`	*sorting*	Defines the sequence in which pricing problems are solved.
`onlybest`	*onlybest*	Determines whether the best found variables (*TRUE*) are added instead of the first found ones (*FALSE*).
`useheur`	*useheur*	Determines whether heuristic pricing should be used (*TRUE*).

In `GCG`, all these parameters are named "`pricing/masterpricer/`", followed by the parameter name listed in the table.

Branching Parameters

Parameter	Symbol	Description
`usepseudocost`	*usepseudocosts*	Determines whether the pseudocost (*TRUE*) or the most fractional (*FALSE*) variable selection rule should be used for branching on original variables.
`enforcebycons`		Determines whether the branching restrictions are enforced in the master problem (*TRUE*) or in the pricing problems (*FALSE*).

In `GCG`, all these parameters are named "`branching/orig/`", followed by the parameter name listed in the table.

Relaxation Parameters

Parameter	Symbol	Description
`discretization`	*usedisc*	Determines whether the discretization approach is used, if possible.
`aggregateblocks`	*aggrblocks*	Determines whether identical blocks are aggregated (*TRUE*) or treated independently (*FALSE*) (only when discretization approach used).
`enforceproper`	*enfoproper*	Determines whether proper variables are enforced (*TRUE*) or not (*FALSE*).

In `GCG`, all these parameters are named "`relaxing/gcg/`", followed by the parameter name listed in the table.

Appendix C

Problems

C.1 The Bin Packing Problem

In the (one-dimensional) *bin packing problem (BPP)*, we are given a finite number of items $i \in I$ together with corresponding weights $w_i \geq 0, i \in I$. The goal is to assign all items to bins in a way such that the capacity C of each bin is not exceeded and the number of bins that are used is minimized.

The bin packing problem is known to be $\mathcal{NP}$-hard [34]. Besides many approximation algorithms that have been proposed for the BPP (for a survey, see [19]), there are also some exact solving methods, see e.g., [62, 90]. We will not go into further detail about these solution methods in this section, we only present two different formulations for the BPP and the test sets used for our computational experiments.

The BPP can be modeled as a MIP, using an assignment formulation. Therefor, a finite number n of bins is needed. One possibility is to set $n = |I|$, on the other hand, if we know a feasible solution to the problem, e.g., computed by a heuristic, we can set n to the number of bins used in this solution.

Model C.1 (BPP: assignment formulation)

$$
\begin{aligned}
z^{\star}_{BPP} = min \quad & \sum_{j=1}^{n} y_j & & \\
s.t. \quad & \sum_{j=1}^{n} x_{i,j} = 1 & \forall i \in I \quad & \text{(C.1)} \\
& \sum_{i \in I} w_i x_{i,j} \leq C y_j & \forall j \in [n] \quad & \text{(C.2)} \\
& x_{i,j} \in \{0,1\} & \forall i \in I, j \in [n] & \\
& y_j \in \{0,1\}. & \forall j \in [n] &
\end{aligned}
$$

The x variables represent the assignment of items to bins: $x_{i,j} = 1$ corresponds to assigning item i to bin j. The y variables model the usage of the

bins: bin j is used if and only if $y_j = 1$. Hence, in the assignment formulation, we minimize the number of bins that are used, while constraints (C.1) assure that each item is assigned to exactly one bin and constraints (C.2) guarantee that items are only assigned to bins that are used and that the bin's capacity is not exceeded.

We distinguish the bins in this formulation although there actually is no difference between two bins, so it is highly symmetric: For each solution, we obtain a huge number of equivalent solutions by permuting the bins. This typically leads to a poor performance when solving the BPP with a branch-and-cut method.

An alternative formulation is the so-called set partitioning formulation. It overcomes the symmetry and leads to a stronger dual bound [60], in return, it contains an exponential number of variables.

Model C.2 (BPP: set partitioning formulation)

$$
\begin{aligned}
z^{\star}_{BPP} = min \quad & \sum_{S \in \mathcal{S}} \lambda^S & & \\
s.t. \quad & \sum_{\substack{S \in \mathcal{S}: \\ i \in S}} \lambda^S = 1 & \forall i \in I & \qquad \text{(C.3)} \\
& \lambda^S \in \{0,1\} & \forall S \in \mathcal{S} &
\end{aligned}
$$

with $\mathcal{S} = \left\{ S \subseteq I \mid \sum_{i \in S} w_i \leq C \right\}$.

The set $\mathcal{S}$ contains all subsets of items that fit into one bin. In this formulation, we choose a minimal number of subsets $S \subseteq \mathcal{S}$ and assure by constraints (C.3) that each item is part of exactly one of the chosen subsets.

By applying the Dantzig-Wolfe decomposition presented in Chapter 2 to the assignment formulation of the bin packing problem, treating constraints (C.1) as linking constraints and constraints (C.2) as structural constraints, we can obtain to this formulation, too. Actually, we get the set partitioning formulation mentioned above when we use the discretization approach, aggregate identical blocks and omit the convexity constraints, that are redundant in this case. If we used the convexification approach, we would get a slightly different model with n identical subsets $\mathcal{S}_j$, one for each bin.

For our computational experiments, we used a part of the instances provided at [84], in particular all instances with 50, 100, and 200 items contained in data set 1. We denote by BINDATA1-N1, BINDATA1-N2, and BINDATA1-N3 the sets of instances with 50, 100, and 200 items, respectively. Each set can be devided into nine classes with 20 instances each. The classes are defined by a capacity $C \in \{100, 120, 150\}$ and the interval used to choose the weights. These weights are uniformly distributed random integer numbers chosen from one of the intervals $[1, 100]$, $[20, 100]$, and $[30, 100]$. Because of the large number of instances per test set, we also defined smaller test sets BINDATA1-N1S,

BINDATA1-N2S, and BINDATA1-N3S with 18 instances per test set, two of each class. We will primary use these small test sets for the computations, only in Chapter 7, we present results for the complete test sets, too.

For the assignment formulation of an instance, we need a maximum number n of used bins. We set $n = 1.5 \cdot \text{OPT}$ with OPT being the optimal solution value of the instance (listed at [84]). Furthermore, we used a set covering approach rather than a set partitioning approach, so the assignement constraints (C.1) change to

$$\sum_{j=1}^{n} x_{i,j} \geq 1 \; \forall i \in I \tag{C.4}$$

assuring that each item is assigned to *at least* one bin. For a given solution to the set covering model, a solution to the Model C.1 can be constructed by assigning each item to exactly one of the bins it is assigned to in the given solution. The set covering formulation allows to construct feasible solutions more easily and is numerically more stable.

We treat constraints (C.4) as linking constraints and constraints (C.2) as structural constraints when solving the bin packing problems with `GCG`.

C.2 The Vertex Coloring Problem

The *vertex coloring problem (VCP)* resembles the bin packing problem. Given an undirected graph $G = (V, E)$, we want to color the nodes V such that adjacent nodes get different colors and we use as few colors as possible. Instead of assigning items to bins, we assign nodes to colors. In contrast to the bin packing problem, the set of nodes that may be assigned to one color is not limited by a capacity, but by the edges of the graph: the nodes that get the same color have to form a stable set in the graph, i.e., no two nodes of the set are adjacent.

The VCP is known to be $\mathcal{NP}$-hard (see [34]). For a broader survey of the VCP, its applications as well as algorithmic and computational results, we refer to [72, 58]. In the following, we will present two formulations for the VCP and the set of test instances that we used for our computational experiments.

The first possible formulation of the VCP is the following assignment formulation. It needs a limited number n of colors. For example, we can always set $n = |V|$, which is an upper bound on the number of colors needed. If we know a feasible solution to the problem, e.g., computed by a heuristic, we can also set n to the number of colors needed for this solution.

Model C.3 (VCP: assignment formulation)

$$
\begin{aligned}
z^\star_{VCP} = min \quad & \sum_{j=1}^{n} y_j & & & \\
s.t. \quad & \sum_{j=1}^{n} x_{v,j} = 1 & & \forall v \in V & \text{(C.5)} \\
& x_{v,j} \leq y_j & & \forall v \in V, j \in [n] & \text{(C.6)} \\
& x_{u,j} + x_{v,j} \leq 1 & & \forall e = (u,v) \in E, j \in [n] & \text{(C.7)} \\
& x_{v,j} \in \{0,1\} & & \forall v \in V, j \in [n] & \\
& y_j \in \{0,1\} & & \forall j \in [n] &
\end{aligned}
$$

The x variables represent the assignment of nodes to colors: $x_{v,j} = 1$ if and only if node v gets color j. The y variables correspond to the usage of the colors, color j is used if and only if $y_j = 1$. Constraints (C.5) assure that each node gets exactly one color and constraints (C.6) guarantee that the y variables are set correctly. Finally, constraints (C.7) ensure that adjacent nodes get different colors.

The assignment formulation has two essential drawbacks: There actually is no difference between the colors, but we have to distinguish them in the model, so it contains a high degree of symmetry. For each solution, by permuting the colors, we get a huge set of equivalent solutions. On the other hand, the LP relaxation of Model C.3 is extremely weak: By setting $y_1 = y_2 = 1$ and $x_{v,1} = x_{v,2} = \frac{1}{2}$ for each node $v \in V$, we can construct a feasible solution of the relaxation with value 2 for each graph.

In order to overcome these drawbacks, we can formulate the VCP as a set partitioning problem with an exponential number of variables.

Model C.4 (VCP: set partitioning formulation)

$$
\begin{aligned}
z^\star_{VCP} = min \quad & \sum_{S \in \mathcal{S}} \lambda^S & & & \\
s.t. \quad & \sum_{\substack{S \in \mathcal{S}: \\ v \in S}} \lambda^S = 1 & & \forall v \in V & \text{(C.8)} \\
& \lambda^S \in \{0,1\} & & \forall S \in \mathcal{S} &
\end{aligned}
$$

with $\mathcal{S} = \{S \subseteq V \mid u \notin S \vee v \notin S \; \forall e = (u,v) \in E\}$

A minimum number of subsets $S \subseteq \mathcal{S}$ is chosen in a way such that each node is part of one of the subsets. In fact, this formulation looks quite similar to the set partitioning formulation of the bin packing problem, except for the definition of the set $\mathcal{S}$, that contains only stable sets this time.

name	nodes	edges	name	nodes	edges
1-FullIns_3	30	100	**miles500**	128	2340
1-FullIns_4	93	593	**miles750**	128	4226
2-FullIns_3	52	201	**mulsol.i.1**	197	3925
2-Insertions_3	37	72	**mulsol.i.2**	188	3885
3-FullIns_3	80	346	**mulsol.i.3**	184	3916
4-FullIns_3	114	541	**mulsol.i.4**	185	3946
4-FullIns_4	690	6650	**mulsol.i.5**	186	3973
5-FullIns_3	154	792	myciel3	11	20
anna	138	986	**myciel4**	23	71
david	87	812	**queen6_6**	36	580
DSJC125.9	125	6961	**queen7_7**	49	952
fpsol2.i.1	496	11654	queen8_8	64	1456
games120	120	1276	queen9_9	81	2112
homer	561	3256	queen10_10	100	2940
huck	74	602	**zeroin.i.1**	211	4100
jean	80	508	**zeroin.i.2**	211	3541
miles1000	128	6432	**zeroin.i.3**	206	3540
miles1500	128	10396	will199GPIA	701	7065
miles250	128	774			

Table C.1: Test set COLORING-ALL of vertex coloring instances. Instances printed in **bold face** are the ones contained in test set COLORING, which was used for all computations except for the comparison to SCIP

This formulation is also obtained by applying the Dantzig-Wolfe decomposition presented in Chapter 2 to the assignment formulation of the vertex coloring problem, treating constraints (C.5) as linking constraints and constraints (C.6) and (C.7) as structural constraints. Using the discretization approach, we can aggregate the identical blocks, omit the convexity constraints and obtain Model C.4. For the convexification approach, we would get a slightly different model with n identical subsets $\mathcal{S}_j$, one for each color.

For our computational experiments, we used a subset of the set of vertex coloring instances given at [46]. We list the instances contained in the complete test set COLORING-ALL in Table C.1 as well as the number of nodes and edges for each instance. Instances printed in bold face are the ones contained in the smaller test set COLORING.

In order to obtain the assignment formulation of an instance, we set $n = 1.5 \cdot \text{OPT}$ with OPT being the optimum of the instance, previously computed by a specialized branch-and-price solver. Furthermore, we used a set covering approach rather than a set partitioning approach, so the assignement

constraints (C.5) change to

$$\sum_{j=1}^{n} x_{v,j} \geq 1 \; \forall v \in V \tag{C.9}$$

assuring that each node gets *at least* one color. Given a solution to the set covering model, a solution to the Model C.3 can be constructed by choosing exactly one of the assigned colors. The set covering formulation allows to construct feasible solutions more easily and is numerically more stable.

When solving the coloring instance with `GCG`, we treat constraints (C.9) as linking constraints and constraints (C.6) and (C.7) as structural constraints.

C.3 The Capacitated p-Median Problem

The *p-median problem (PMP)* is a classical location problem. Let N be the set of nodes of a graph, representing sites where users are placed, and $M \subseteq N$ be the set of potential location sites. The objective is to choose p medians (facilities) of this set M in a way such that the sum of the distances from each user to its nearest facility is minimized. The distance from user $i \in N$ to facility $j \in M$ is denoted by $d_{i,j} \in \mathbb{Z}_+$. The PMP is $\mathcal{NP}$-hard [34].

The *capacitated p-median problem (CPMP)* is a generalization of the PMP, where each user $i \in N$ has a demand $q_i \in \mathbb{Z}_+$ while for each node $j \in M$, a facility placed at this node would have a production capacity of at most $Q_j \in \mathbb{Z}_+$. Again, we have to choose p medians and assign users to the medians in a way such that the sum of the distances from each user to the assigned facility is minimized. Additionally, we have to respect the capacity restrictions, i.e., the total demand of nodes assigned to one facility must not exceed its capacity. For more details about the CPMP and solution methods, we refer to [15, 17] and the references given there.

The assignment formulation for the CPMP is the following:

Model C.5 (CPMP: assignment formulation)

$$z^{\star}_{CPMP} = min \quad \sum_{i \in N} \sum_{j \in M} d_{i,j} x_{i,j}$$

$$s.t. \quad \sum_{j \in M} x_{i,j} = 1 \qquad \forall i \in N \tag{C.10}$$

$$\sum_{j \in M} y_j = p \tag{C.11}$$

$$\sum_{i \in N} q_i x_{i,j} \leq Q_j y_j \qquad \forall j \in M \tag{C.12}$$

$$x_{i,j} \in \{0,1\} \qquad \forall i \in N, j \in M$$

$$y_j \in \{0,1\}. \qquad \forall j \in M$$

The x variables represent the assignment of users to facilities: $x_{i,j} = 1$ means that user i is served by facility j. The y variables correspond to the opening of the facilities: $y_j = 1$ if and only if facility j is opened. Constraint (C.11) ensures that exactly p facilities are opened. Again, the constraints (C.10) assure that each user is served by exactly one facility and constraints (C.12) guarantee that users can only be served by opened facilities and that the capacitys of the facilities is not exceeded.

This formulation contains not as much symmetry as the assignment formulations of the bin packing problem and the vertex coloring problem. Facilities placed at different nodes may have different capacities so that a set of users can be assigned to one facility but exceeds the capacity of a facility placed at another node. Moreover, even if all facilities have the same capacity, reassigning a set of users to a different facility or permuting the facilities of two or more sets usually leads to significant changes in the objective function value since the distance between user and assigned facility is respected in the objective function.

Nevertheless, we can formulate the capacitated p-median problem as a set partitioning problem with an exponential number of variables in order to get a tighter lower bound.

Model C.6 (CPMP: set partitioning formulation)

$$
\begin{aligned}
z^{\star}_{CPMP} = min \quad & \sum_{j \in M} \sum_{S \in \mathcal{S}_j} c_S^j \lambda_S^j & & \\
s.t. \quad & \sum_{j \in M} \sum_{\substack{S \in \mathcal{S}_j: \\ i \in S}} \lambda_S^j = 1 & \forall i \in N \quad & \text{(C.13)} \\
& \sum_{S \in \mathcal{S}_j} \lambda_S^j \leq 1 & \forall j \in M \quad & \text{(C.14)} \\
& \sum_{j \in M} \sum_{S \in \mathcal{S}_|} \lambda_S^j = p & & \text{(C.15)} \\
& \lambda_S^j \in \{0,1\} \qquad \forall S \in \mathcal{S}_j, j \in M & &
\end{aligned}
$$

with $\mathcal{S}_j = \left\{ S \subseteq N \mid \sum_{i \in S} q_i \leq Q_j \right\}$ *and* $c_S^j = \sum_{i \in S} d_{i,j}$ *for* $S \in \mathcal{S}_j$.

For each possible location of a facility $j \in M$, the set $\mathcal{S}_j \subseteq 2^N$ contains all subsets of users that have a combined demand that does not exceed the capacity of this facility. For each of these sets $S \in \mathcal{S}_j$, we define a binary variable λ_S^j that represents the decision of placing a facility at node j and assigning all users in S to that facility. Its objective function coefficient is equal to the sum of distances between the users in S and the facility placed at node j. Constraints (C.13) assure that each user is served by one facility, constraints (C.14) and constraints (C.15) ensure that exactly p facilities are opened, but no more than one at each node.

Model C.6 can be obtained by applying the Dantzig-Wolfe decomposition (see Chapter 2) to the assignment formulation (Model C.5), treating constraints (C.10) and (C.11) as linking constraints and constraints (C.12) as structural constraints. This is also the structure that we use when solving the CPMP instances with GCG.

For our computational experiments, we used the set [14] of test instances for the CPMP, that were previously used in [15, 17]. The instances are named $pFN-n$, where $N \in \{50, 100, 150, 200\}$ is the number of users, $F \in \left\{\frac{N}{10}, \left\lceil\frac{N}{4}\right\rceil, \left\lceil\frac{N}{3}\right\rceil, \frac{2N}{5}\right\}$ is the number of facilities that must be opened, i.e., the value p of the problem definition, and n is a consecutive numbering of the ten instances for each combination of N and F. For each instance, all facilities have the same capacity $C = \left\lceil 12\frac{N}{p}\right\rceil$.

We defined four testsets CPMP50, CPMP100, CPMP150, and CPMP200 containing the instances with 50, 100, 150, and 200 users, respectively. In order to reduce the computational effort, we also defined small testsets CPMP50S, CPMP100S, CPMP150S, and CPMP200S.

For each number of users N up to 150, we removed every second instance from the corresponding complete test set and obtained a small test set with 20 instances, five instances for each number of facilities F. The instances with 200 users are the hardest ones, especially those with 50, 66, and 80 facilities hit the time limit in most of the cases. Therefore, we chose only three instances for each number of facilities, and we chose primarily those instances that could be solved within the time limit.

C.4 A Resource Allocation Problem

Finally, we regard a generalized knapsack problem [13] that does not have a bordered block diagonal structure. Given a number of periods $1 \leq n \leq N$ and items $i \in I$, each item has a profit p_i, a weight w_i, and a starting and ending period. The periods are implicitly given by the sets $I(n) \subseteq I$, $n \in [N]$, that contain all items that are alive in period n. In each period, the knapsack has capacity C and items consume capacity only during their *life time*. The goal is to maximize the sum of profits of the selected items in such a way that for each period, the capacity is not exceeded. The problem can be modelled in the following way:

Model C.7 (RAP)

$$
\begin{aligned}
z^{\star}_{RAP} = \max \quad & \sum_{i \in I} p_i x_i & & \\
s.t. \quad & \sum_{i \in I(n)} w_i x_i \leq C & \forall n \in [N] \qquad & \text{(C.16)} \\
& x_i \in \{0,1\} & \forall i \in I. &
\end{aligned}
$$

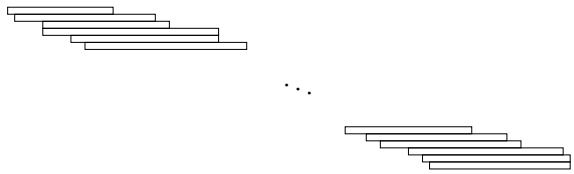


Figure C.1: Structure of the constraint maxtrix of RAP problems. Each rectangle represents the non zero entries of one constraint.

An item i is selected if and only if the corresponding variable x_i has value 1. Constraints C.16 assure that for each period, the capacity is respected.

By ordering the columns, we can obtain a special structure of the constraint matrix for the RAP problems which is pictured in Figure C.1.

The matrix can be transformed into block structure by splitting the periods $[N]$ into groups of size M (see [13]). We set $G = \lceil \frac{N}{M} \rceil$, and define $N(g) = \{(g-1)M + 1, \ldots, gM\} \cap [N]$ for $g \in [G]$.

Additionally, let $G(i)$ be the set of groups in which item i is alive and $I(g)$ be the items that are alive in at least one period of group $g \in [G]$. Furthermore, we denote by $g(i)$ the first group in which item i is active, i.e., $g(i) = \min\{g \in G(i)\}$.

For each item $i \in I$ and group $g \in G(i)$, we create a copy x_i^g of the variable corresponding to that item and ensure that all these copies get the same value in each solution by additional constraints. Furthermore, we substitute the original variables x_i by $x_i^{g(i)}$ in the objective function.

This leads to the following model that has a bordered block diagonal structure.

Model C.8 (RAP: block diagonal)

$$
\begin{aligned}
z^\star_{RAP} = \max \quad & \sum_{i \in I} p_i x_i^{g(i)} & & \\
s.t. \quad & x_i^g = x_i^{g(i)} & \forall i \in I, g \in G(i) \setminus \{g(i)\} \quad & \text{(C.17)} \\
& \sum_{i \in I(n)} w_i x_i^g \leq C & \forall g \in G, n \in N(g) \quad & \text{(C.18)} \\
& x_i^g \in \{0,1\} & \forall i \in I, g \in G(i) &
\end{aligned}
$$

Constraints (C.17) assure that all copies corresponding to the same item get the same value. An item i is chosen if and only if all copies x_i^g, $g \in G(i)$, have value 1. Constraints (C.18) are equivalent to constraints (C.16) and enforce that the capacities are respected for each period.

When solving the RAP problems with `GCG`, we use Model C.8 and perform the Dantzig-Wolfe decomposition on it. Constraints (C.17) are treated as linking constraints and transferred to the master problem, constraints (C.18)

go into the blocks. For the discretization approach, we obtain the following extended problem:

Model C.9 (RAP: extended problem)

$$
\begin{aligned}
z^{\star}_{RAP} = \max \quad & \sum_{i \in I} p_i x_i & & \\
s.t. \quad & \sum_{\substack{S \in \mathcal{S}_g: \\ i \in S}} \lambda_S^g = \sum_{\substack{S \in \mathcal{S}_{g(i)}: \\ i \in S}} \lambda_S^{g(i)} & \forall i \in I, g \in G(i) \setminus \{g(i)\} \quad & \text{(C.19)} \\
& \sum_{S \in \mathcal{S}_g} \lambda_S^g = 1 & \forall g \in [G] \quad & \text{(C.20)} \\
& \lambda_S^g \in \{0,1\} & \forall g \in [G], S \in \mathcal{S}_g &
\end{aligned}
$$

with $\mathcal{S}_g = \left\{ S \subseteq I(g) \mid \sum_{i \in I(n) \cap S} w_i \leq C \ \forall n \in N(g) \right\}$.

The set $\mathcal{S}_g$ contains all subsets of active items of group g that do not violate the capacity constraints corresponding to the periods of this group. Constraints (C.20) ensure that for each group, exactly one of these sets is chosen, and constraints (C.19) enforce that items are either chosen in each group in which they are active or in none of them.

For this type of problems, we used the test set described in [13], which contains seven classes, each of which contains ten instances. In the same way as it is done there, we consider grouping 32 and 64 periods and defined the corresponding test sets RAP32 and RAP64, respectively. In order to reduce the computational effort, we also defined small test sets RAP32S and RAP64S, that contain both the same 21 instances, three instances of each class. As far as possible, we chose instances that could be solved within a time limit of one hour. The complete test set is presented in Table C.2, the instances printed in bold face are the ones contained in the smaller test sets. They were used for most computations except for the comparison with `SCIP` where we used the complete test set. For each instance, we list the number of items (column "$|I|$") and the number of periods (column "N").

name	$\|I\|$	N	name	$\|I\|$	N
new1_1	2071	768	new4_6	3774	1408
new1_2	2422	896	new4_7	4164	1536
new1_3	2756	1024	new4_8	4488	1664
new1_4	3104	1152	new4_9	4786	1792
new1_5	3433	1280	**new4_10**	5142	1920
new1_6	3789	1408	new5_1	2916	768
new1_7	4154	1536	**new5_2**	3424	896
new1_8	4476	1664	new5_3	3832	1024
new1_9	4797	1792	new5_4	4316	1152
new1_10	5129	1920	new5_5	4771	1280
new2_1	2071	768	new5_6	5403	1408
new2_2	2422	896	**new5_7**	5793	1536
new2_3	2756	1024	new5_8	6167	1664
new2_4	3104	1152	new5_9	6800	1792
new2_5	3433	1280	**new5_10**	7241	1920
new2_6	3789	1408	**new6_1**	5210	768
new2_7	4154	1536	**new6_2**	6057	896
new2_8	4476	1664	new6_3	6901	1024
new2_9	4797	1792	new6_4	7737	1152
new2_10	5129	1920	**new6_5**	8656	1280
new3_1	4948	768	new6_6	9370	1408
new3_2	5769	896	new6_7	10271	1536
new3_3	6721	1024	new6_8	11057	1664
new3_4	7382	1152	new6_9	11992	1792
new3_5	8266	1280	new6_10	13025	1920
new3_6	9002	1408	**new7_1**	3117	768
new3_7	9865	1536	new7_2	3594	896
new3_8	10661	1664	new7_3	4176	1024
new3_9	11448	1792	**new7_4**	4671	1152
new3_10	12498	1920	new7_5	5209	1280
new4_1	2071	768	new7_6	5628	1408
new4_2	2422	896	new7_7	6215	1536
new4_3	2763	1024	new7_8	6730	1664
new4_4	3103	1152	**new7_9**	7172	1792
new4_5	3434	1280	new7_10	7709	1920

Table C.2: Test set of RAP instances. Instances printed in **bold face** are the ones contained in the small test sets RAP32S and RAP64S, which were used for all computations except for the comparison to SCIP

Appendix D

Tables

In this chapter, we present the results of our test runs in detail. The tables all organized as follows. Let us clarify that all time measurements are given in seconds.

Tables D.1 to D.10 provide information about the Farkas pricing process at the root node, Tables D.11 to D.20 list details about the reduced cost pricing process at the root node. Each table pictures the results for one test set, the rows belong to different settings that are described in the corresponding section. The columns list the number of pricing rounds, the number of pricing problems that were solved, the number of variables that were created, the pricing time, the LP solving time, the total time and the number of time outs. Except for the number of time outs, all values are shifted geometric means of the values of the individual instances of the test set. In the tables corresponding to reduced cost pricing, the first five columns list just the values corresponding to reduced cost pricing, i.e., we substracted for each instance the values belonging to Farkas pricing. The total time listed, however, is the total solving time of the master problem.

Tables D.21 to D.30 provide details about the branch-and-price process for the single test sets. They are constructed in the same way as the previous tables, except for two additional columns that list the number of branch-and-bound nodes and the final gap.

Tables D.31 to D.40 present detailed results about the dual bound obtained at the root node. These results are summarized in Section 6.4. We list for each instance the optimal objective function value, if it is not known, we list the objective function value of the best known solution and indicate this with a star (*) behind the value. In the next columns, the dual bound, the gap to the optimal or best known solution and the solving time of the root node are listed for GCG as well as for SCIP with two different settings. Setting "SCIP all cuts" corresponds to SCIP with default separation settings, setting "SCIP no base" to SCIP without using cuts that need information about the current LP basis. The dual bound and the gap obtained by SCIP is colored red for an instance if GCG computed the tighter dual bound. If SCIP obtained the tighter dual bound, it is colored blue. For both SCIP

settings, we disabled strong branching and used most infeasible branching. Strong branching increased the solving time of the root node and can lead to stronger dual bounds which would distort the results since we just want to compare the dual bounds after the relaxation is solved.

Tables D.41 to D.46 provide details about the computational experiments concerning reduced cost pricing that are summarized in Section 7.1. Each Table corresponds to one test set, the rows correspond to the single instances of the test set. For each of the settings listed in the head row, we present the number of variables created during the solving process, the final gap, the number of branch-and-bound nodes and the total solving time in seconds. A solving time of 3.600 denotes that the time limit of one hour was hit.

Tables D.47 to D.50 present details about the computational experiments summarized in Sections 7.2 and 7.3. Each Table corresponds to one test set, the rows correspond to the single instances of the test set. For each of the settings listed in the head row, we present the final gap, the number of branch-and-bound nodes and the total solving time in seconds. A solving time of 3.600 denotes that the time limit of one hour was hit.

Finally, Tables D.51 to D.56 present detailed comparisons of `GCG` and `SCIP` for the complete vertex coloring and CPMP test sets as well as for the complete RAP test set, with both variants described in Section C.4, assigning either 32 or 64 constraints to one block. These results are summarized and interpreted in Section 7.4.

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	238.4	238.4	238.4	0.4	0.1	0.8	0/20
all vars	95.2	4756.7	12776.9	7.2	2.1	9.7	0/20
no sorting	2056.4	4858.7	2056.4	21.1	4.5	26.9	0/20
two vars	178.9	2079.9	319.6	1.2	0.1	1.5	0/20
2%	161.4	161.4	384.3	0.2	0.1	0.4	0/20
one prob	161.4	161.4	384.3	0.2	0.1	0.4	0/20

Table D.1: Computational results for Farkas pricing at the root node on test set CPMP50S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	571.3	571.3	571.3	2.6	1.0	4.4	0/20
all vars	189.1	18904.4	64103.2	53.2	121.5	181.2	6/20
no sorting	6503.7	20301.2	6503.7	128.7	121.8	288.7	14/20
two vars	402.3	7512.5	731.9	6.2	0.7	7.8	0/20
2%	293.3	586.5	1526.1	1.1	1.0	2.8	0/20
one prob	337.9	337.9	1064.4	0.9	0.6	2.2	0/20

Table D.2: Computational results for Farkas pricing at the root node on test set CPMP100S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	1011.4	1011.4	1011.4	9.7	5.4	17.2	0/20
all vars	237.4	35685.3	103094.7	92.9	217.6	300.0	20/20
no sorting	5303.5	24424.6	5303.5	110.5	174.0	299.6	20/20
two vars	660.5	16719.2	1211.0	19.9	3.3	25.2	0/20
2%	436.7	1309.8	3663.7	3.9	6.1	12.6	0/20
one prob	525.4	525.4	1841.7	2.7	3.0	7.7	0/20

Table D.3: Computational results for Farkas pricing at the root node on test set CPMP150S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	1457.7	1457.7	1457.7	30.4	17.6	57.6	0/12
all vars	266.7	53404.0	127254.4	134.8	158.7	299.9	12/12
no sorting	3301.2	28155.6	3301.2	156.9	70.1	299.3	12/12
two vars	898.4	28869.0	1649.2	51.5	8.7	68.2	1/12
2%	582.7	2330.5	6791.0	8.9	25.2	44.7	0/12
one prob	700.8	700.8	2676.9	5.4	8.4	20.1	0/12

Table D.4: Computational results for Farkas pricing at the root node on test set CPMP200S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	241.1	241.1	241.1	3.8	0.1	6.8	0/28
all vars	145.4	3256.9	4507.9	38.4	0.6	40.5	0/28
no sorting	2240.2	3723.0	2240.2	168.9	14.7	243.8	11/28
two vars	216.7	634.6	411.3	8.3	0.1	10.6	0/28
2%	222.9	263.1	424.6	4.2	0.1	7.0	0/28
one prob	232.4	232.4	394.2	3.7	0.1	6.7	0/28
disc all	172.3	172.3	256.0	2.8	0.1	6.2	0/28
disc best	176.0	176.0	176.0	3.0	0.0	6.4	0/28

Table D.5: Computational results for Farkas pricing at the root node on test set COLORING

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	37.7	37.7	37.7	0.1	0.0	0.4	0/21
all vars	37.7	1336.9	37.7	2.0	0.0	2.3	0/21
no sorting	37.7	695.7	37.7	1.0	0.0	1.4	0/21
two vars	37.7	1336.9	37.7	2.0	0.0	2.3	0/21
2%	37.7	46.3	37.7	0.1	0.0	0.4	0/21
one prob	37.7	37.7	37.7	0.1	0.0	0.4	0/21

Table D.6: Computational results for Farkas pricing at the root node on test set RAP32S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	19.0	19.0	19.0	0.0	-0.0	0.2	0/21
all vars	19.0	345.5	19.0	0.8	-0.0	1.0	0/21
no sorting	19.0	186.9	19.0	0.4	-0.0	0.6	0/21
two vars	19.0	345.5	19.0	0.8	-0.0	1.0	0/21
2%	19.0	19.0	19.0	0.0	-0.0	0.2	0/21
one prob	19.0	19.0	19.0	0.0	-0.0	0.2	0/21

Table D.7: Computational results for Farkas pricing at the root node on test set RAP64S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	227.3	227.3	227.3	0.2	0.0	0.4	0/18
all vars	60.4	2287.7	2725.0	1.6	0.1	1.9	0/18
no sorting	1979.9	13984.2	1979.9	107.4	1.2	109.0	2/18
two vars	172.1	952.8	324.9	0.6	0.0	0.8	0/18
2%	201.8	220.0	311.9	0.2	0.0	0.4	0/18
one prob	219.1	219.1	312.2	0.2	0.0	0.4	0/18
disc all	61.0	61.0	71.4	0.0	0.0	0.3	0/18
disc best	63.2	63.2	63.2	0.0	0.0	0.3	0/18

Table D.8: Computational results for Farkas pricing at the root node on test set BINDATA1-N1S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	545.6	545.6	545.6	1.0	0.1	2.1	0/18
all vars	120.0	8825.0	12047.5	9.8	0.5	11.4	0/18
no sorting	4278.9	29857.4	4278.9	294.8	3.6	299.9	18/18
two vars	415.3	3302.7	790.6	2.9	0.1	4.0	0/18
2%	344.8	695.2	968.6	0.8	0.1	1.9	0/18
one prob	462.1	462.1	775.6	0.7	0.1	1.8	0/18
disc all	122.0	122.0	166.0	0.1	0.0	1.1	0/18
disc best	131.4	131.4	131.4	0.2	0.0	1.2	0/18

Table D.9: Computational results for Farkas pricing at the root node on test set BINDATA1-N2S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	1463.0	1463.0	1463.0	7.9	0.6	15.8	0/18
all vars	229.1	34721.0	54843.6	60.7	6.7	77.0	0/18
no sorting	4927.5	25133.5	4927.5	278.9	8.4	299.4	18/18
two vars	1118.9	13299.3	2152.0	19.3	0.7	27.2	0/18
2%	644.2	2301.5	3278.6	5.5	0.4	13.3	0/18
one prob	1352.8	1352.8	2539.6	7.7	0.9	15.8	0/18
disc all	230.8	230.8	362.5	0.4	0.0	8.0	0/18
disc best	256.6	256.6	256.6	0.7	0.0	8.4	0/18

Table D.10: Computational results for Farkas pricing at the root node on test set BINDATA1-N3S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	21.7	753.3	1502.5	1.7	0.1	2.6	0/20
only best	307.4	15185.6	307.4	30.2	0.5	31.4	0/20
all vars	18.4	906.1	2414.9	2.2	0.1	3.1	0/20
100 best	22.2	1073.2	1554.0	2.7	0.1	3.5	0/20
one prob	508.1	3258.9	1110.1	9.6	1.3	11.6	0/20

Table D.11: Computational results for reduced cost pricing at the root node on test set CPMP50S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	35.8	1539.0	2869.8	4.0	0.6	8.9	0/20
only best	665.7	66392.4	665.7	163.0	3.7	171.3	5/20
all vars	23.7	2320.0	7556.0	7.3	1.1	12.8	0/20
100 best	37.6	3628.3	3144.5	11.1	0.7	16.2	0/20
one prob	1070.0	12503.0	2533.4	34.6	6.7	47.0	1/20

Table D.12: Computational results for reduced cost pricing at the root node on test set CPMP100S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	51.3	2515.1	4339.5	7.7	1.7	27.2	0/20
only best	391.2	58091.2	391.2	276.0	5.2	299.6	20/20
all vars	27.9	4078.4	14925.7	16.4	3.6	38.3	0/20
100 best	53.8	7835.2	4766.0	29.5	2.2	49.9	0/20
one prob	1579.9	19389.8	4009.0	64.3	17.1	107.7	5/20

Table D.13: Computational results for reduced cost pricing at the root node on test set CPMP150S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	69.6	3534.6	6073.8	12.6	4.2	73.4	0/12
only best	151.5	29303.2	151.5	221.9	4.3	299.2	12/12
all vars	31.1	6114.7	25224.5	29.6	8.2	95.2	1/12
100 best	67.1	13175.2	6234.3	58.2	4.9	120.7	1/12
one prob	1571.9	17727.2	4818.3	79.2	28.4	185.5	3/12

Table D.14: Computational results for reduced cost pricing at the root node on test set CPMP200S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	40.5	487.3	1695.5	33.6	0.5	40.9	0/28
only best	45.4	834.7	45.4	51.3	0.2	58.3	0/28
all vars	31.8	603.2	2487.1	41.3	0.5	47.8	0/28
100 best	31.7	604.4	1397.9	40.7	0.3	47.1	0/28
one prob	49.0	82.8	231.1	5.8	0.2	12.2	0/28
disc all	70.2	71.3	292.0	6.2	0.2	11.4	0/28
disc best	79.0	80.0	79.0	7.2	0.2	12.4	0/28

Table D.15: Computational results for reduced cost pricing at the root node on test set COLORING

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	48.4	792.2	3646.1	177.5	0.6	178.8	0/21
only best	209.1	7376.6	208.7	1395.9	0.3	1396.9	13/21
all vars	17.9	680.8	4343.5	149.3	0.4	150.3	0/21
100 best	23.5	884.3	1443.9	185.1	0.2	185.9	0/21
one prob	340.0	3877.7	4121.1	934.6	4.4	940.2	8/21

Table D.16: Computational results for reduced cost pricing at the root node on test set RAP32S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	35.3	321.9	2539.0	289.6	0.2	290.2	0/21
only best	102.3	1779.2	102.2	1280.7	0.1	1281.2	14/21
all vars	15.7	302.6	2998.4	260.8	0.1	261.3	0/21
100 best	19.2	367.1	1142.0	309.8	0.1	310.1	0/21
one prob	149.6	832.4	2675.6	861.8	0.7	863.2	9/21

Table D.17: Computational results for reduced cost pricing at the root node on test set RAP64S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	34.2	937.4	1812.6	6.7	0.1	7.3	0/18
only best	53.1	1656.9	53.1	17.0	0.1	17.6	0/18
all vars	35.9	1146.4	2582.2	8.4	0.2	9.0	0/18
100 best	36.6	1175.9	1933.1	8.4	0.1	9.0	0/18
one prob	36.7	77.1	165.8	0.6	0.0	1.1	0/18
disc all	73.9	74.9	168.9	0.4	0.0	0.7	0/18
disc best	97.1	98.1	97.1	0.6	0.0	0.9	0/18

Table D.18: Computational results for reduced cost pricing at the root node on test set BINDATA1-N1S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	79.1	3370.1	6618.5	50.3	0.8	54.3	0/18
only best	129.8	8978.7	129.8	148.0	0.5	152.0	5/18
all vars	77.2	5225.8	15971.5	82.3	1.9	88.0	2/18
100 best	93.4	6354.1	7846.7	92.1	1.3	97.1	2/18
one prob	74.3	152.3	480.5	5.8	0.3	8.6	0/18
disc all	150.2	151.3	508.7	2.1	0.1	3.5	0/18
disc best	222.8	223.8	222.8	3.9	0.2	5.4	0/18

Table D.19: Computational results for reduced cost pricing at the root node on test set BINDATA1-N2S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	total time	time outs
default	132.6	6902.7	12848.5	101.8	4.2	126.6	2/18
only best	108.0	15728.1	108.0	276.5	1.4	295.2	16/18
all vars	98.6	13664.9	52565.7	236.1	13.5	274.2	14/18
100 best	113.7	16725.2	11885.3	237.1	4.4	261.5	11/18
one prob	114.0	276.5	839.3	9.1	1.9	31.3	0/18
disc all	283.4	284.4	1159.4	5.9	0.8	15.6	0/18
disc best	469.5	470.5	469.5	11.7	0.9	22.0	0/18

Table D.20: Computational results for reduced cost pricing at the root node on test set BINDATA1-N3S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	398.6	4771.1	1970.3	11.9	0.4	44.9	0.0	12.8	0/20
most frac.	564.7	7375.3	2078.1	19.4	1.0	82.5	0.0	20.7	0/20
master	395.3	4684.3	1965.2	11.6	0.5	44.6	0.0	12.5	0/20

Table D.21: Computational results for the branch-and-price process on test set CPMP50S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	2052.9	81141.9	4700.2	170.1	6.0	587.1	0.1	184.7	1/20
most frac.	4646.5	218552.0	4878.4	433.3	12.5	1962.6	0.8	469.5	6/20
master	2001.8	79623.0	4641.4	166.4	6.5	596.2	0.2	181.7	1/20

Table D.22: Computational results for the branch-and-price process on test set CPMP100S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	3519.4	168804.7	7417.9	428.8	21.0	847.6	0.7	493.5	5/20
most frac.	5861.6	346729.2	7210.4	802.9	28.9	2211.7	1.6	920.3	10/20
master	3516.4	168278.9	7559.2	429.1	24.2	830.4	1.0	501.0	5/20

Table D.23: Computational results for the branch-and-price process on test set CPMP150S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	4545.1	405458.0	10307.3	1101.6	46.0	1753.3	0.2	1243.9	3/12
most frac.	9761.0	1023719.3	10925.2	2605.2	78.9	5577.7	2.2	2978.0	10/12
master	4737.6	427010.4	10021.6	1177.9	49.8	1926.6	0.3	1343.6	3/12

Table D.24: Computational results for the branch-and-price process on test set CPMP200S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	139.0	2361.6	3752.8	547.0	1.2	26.8	0.0	549.0	0/21
most frac.	155.3	2643.1	3768.7	613.6	1.3	32.9	0.0	615.8	1/21
master	142.1	2445.9	3764.6	567.8	1.3	27.3	0.0	569.8	0/21

Table D.25: Computational results for the branch-and-price process on test set RAP32S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
default	74.6	617.8	2592.5	556.9	0.3	9.5	0.0	557.5	1/21
most frac.	73.5	607.2	2586.6	547.1	0.2	9.4	0.0	547.7	1/21
master	72.7	604.3	2590.9	547.6	0.2	8.8	0.0	548.1	1/21

Table D.26: Computational results for the branch-and-price process on test set RAP64S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
conv. def.	299.1	1590.7	2931.5	13.9	0.3	8.6	0.0	14.4	0/18
conv.	315.6	366.7	522.0	1.8	0.1	11.0	0.0	2.1	0/18
disc.	143.4	142.0	243.5	0.9	0.0	2.2	0.0	1.1	0/18

Table D.27: Computational results for the branch-and-price process on test set BINDATA1-N1S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
conv. def.	744.1	4947.8	9649.5	98.2	2.4	24.7	0.0	102.3	0/18
conv.	771.6	873.5	1415.0	13.2	1.0	20.8	0.0	15.1	0/18
disc.	308.6	294.5	660.4	4.7	0.3	10.9	0.0	5.6	0/18

Table D.28: Computational results for the branch-and-price process on test set BINDATA1-N2S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
conv. def.	2095.1	13438.4	22867.7	311.9	17.0	30.8	0.0	343.1	0/18
conv.	2245.7	2989.2	3932.0	48.3	7.8	31.4	0.0	61.2	0/18
disc.	595.7	581.9	1492.4	12.6	1.6	8.9	0.0	17.3	0/18

Table D.29: Computational results for the branch-and-price process on test set BINDATA1-N3S

setting	pricing rounds	problems solved	variables created	pricing time	lp time	B&B nodes	gap	total time	time outs
conv. def.	584.5	2047.4	4578.7	129.7	6.5	84.4	1.6	151.2	1/28
conv.	1084.3	1397.7	1500.7	41.7	7.5	78.2	1.6	60.6	1/28
disc.	645.2	596.2	1791.9	57.0	3.6	52.3	0.0	65.9	0/28
disc. no aggr.	1038.0	1411.2	1451.1	39.6	6.4	69.3	1.6	56.0	1/28

Table D.30: Computational results for the branch-and-price process on test set COLORING

		GCG			SCIP all cuts			SCIP no base		
instance	opt	db	gap	time	db	gap	time	db	gap	time
p550-1	713	705.00	1.13	3.70	702.55	1.49	1.50	703.33	1.38	0.40
p550-2	740	740.00	0.00	3.50	740.00	0.00	0.90	740.00	0.00	0.30
p550-3	751	749.00	0.27	5.20	748.88	0.28	1.60	748.88	0.28	0.40
p550-4	651	651.00	0.00	5.70	651.00	0.00	0.90	651.00	0.00	0.30
p550-5	664	664.00	0.00	6.60	664.00	0.00	1.40	664.00	0.00	0.40
p550-6	778	778.00	0.00	3.70	778.00	0.00	1.50	778.00	0.00	0.40
p550-7	787	778.25	1.12	4.10	775.38	1.50	1.50	775.61	1.47	0.40
p550-8	820	771.67	6.26	4.30	769.30	6.59	1.60	769.30	6.59	0.40
p550-9	715	712.40	0.36	3.60	710.93	0.57	2.50	710.93	0.57	0.60
p550-10	829	817.87	1.36	4.80	805.32	2.94	2.60	805.41	2.93	0.60
p1250-1	383	373.33	2.59	1.90	373.65	2.50	1.00	373.25	2.61	0.30
p1250-2	412	408.21	0.93	1.90	409.52	0.60	1.60	407.72	1.05	0.30
p1250-3	405	398.25	1.69	1.90	398.93	1.52	1.90	397.40	1.91	0.40
p1250-4	384	364.93	5.23	2.10	362.57	5.91	1.60	362.19	6.02	0.40
p1250-5	429	418.80	2.44	2.40	418.34	2.55	1.90	417.55	2.74	0.50
p1250-6	482	466.21	3.39	2.40	463.27	4.04	2.10	462.69	4.17	0.70
p1250-7	445	424.33	4.87	2.20	421.82	5.50	1.60	422.51	5.32	0.60
p1250-8	403	392.73	2.61	2.40	390.20	3.28	2.00	389.28	3.52	0.60
p1250-9	436	422.46	3.20	2.60	420.18	3.77	2.50	417.04	4.55	0.50
p1250-10	461	442.17	4.26	2.70	436.02	5.73	2.10	435.52	5.85	0.50
p1650-1	298	296.00	0.68	1.80	296.88	0.38	1.20	295.50	0.85	0.30
p1650-2	336	328.00	2.44	1.70	333.13	0.86	2.20	326.69	2.85	0.40
p1650-3	314	309.00	1.62	1.80	313.45	0.18	1.20	308.59	1.75	0.30
p1650-4	303	296.07	2.34	1.90	297.63	1.81	1.50	295.67	2.48	0.40
p1650-5	351	346.00	1.45	1.90	345.12	1.70	1.30	345.43	1.61	0.40
p1650-6	390	386.50	0.91	2.00	383.46	1.71	1.90	383.24	1.76	0.60
p1650-7	361	357.00	1.12	2.00	353.85	2.02	2.00	353.85	2.02	0.50
p1650-8	353	322.64	9.41	2.00	328.03	7.61	1.80	318.33	10.89	0.40
p1650-9	373	366.00	1.91	2.00	365.40	2.08	1.90	363.22	2.69	0.60
p1650-10	390	375.21	3.94	2.50	369.52	5.54	1.90	369.94	5.42	0.50
p2050-1	266	258.70	2.82	1.70	260.55	2.09	1.20	258.50	2.90	0.40
p2050-2	298	292.50	1.88	1.70	293.52	1.53	1.50	291.43	2.25	0.50
p2050-3	311	307.00	1.30	2.00	307.54	1.12	2.20	304.41	2.16	0.50
p2050-4	277	275.50	0.54	1.90	275.60	0.51	1.60	275.50	0.54	0.50
p2050-5	356	354.50	0.42	2.20	349.93	1.74	2.00	346.32	2.79	0.60
p2050-6	370	367.00	0.82	2.10	366.69	0.90	1.70	366.63	0.92	0.60
p2050-7	358	357.00	0.28	2.20	353.22	1.35	1.70	354.11	1.10	0.80
p2050-8	312	297.07	5.03	2.00	294.41	5.97	1.80	291.34	7.09	0.50
p2050-9	412	403.40	2.13	2.60	394.57	4.42	2.40	391.34	5.28	0.90
p2050-10	458	441.17	3.82	2.30	418.96	9.32	2.20	418.63	9.40	0.60
mean		441.49	2.15	2.65	439.80	2.51	1.73	438.35	2.91	0.48

Table D.31: Comparison of the dual bound obtained by `SCIP` and `GCG` on test set CPMP50

instance	opt	GCG db	GCG gap	GCG time	SCIP all cuts db	SCIP all cuts gap	SCIP all cuts time	SCIP no base db	SCIP no base gap	SCIP no base time
p10100-11	1006	1001.07	0.49	30.10	998.98	0.70	3.90	998.98	0.70	2.70
p10100-12	966	958.49	0.78	12.30	955.72	1.08	4.40	955.82	1.07	2.70
p10100-13	1026	1021.55	0.44	13.70	1020.02	0.59	3.40	1020.00	0.59	2.00
p10100-14	982	971.75	1.05	15.10	968.10	1.44	3.90	968.71	1.37	2.70
p10100-15	1091	1080.41	0.98	15.40	1076.40	1.36	4.80	1076.40	1.36	3.10
p10100-16	954	951.33	0.28	18.90	948.19	0.61	4.20	948.49	0.58	2.80
p10100-17	1034	1025.28	0.85	13.30	1020.96	1.28	4.30	1022.51	1.12	3.30
p10100-18	1043	1031.90	1.08	13.40	1029.33	1.33	3.90	1029.38	1.32	2.50
p10100-19	1031	1026.26	0.46	13.80	1022.22	0.86	4.90	1022.19	0.86	3.30
p10100-20	1005	973.65	3.22	14.40	965.43	4.10	6.30	967.90	3.83	2.90
p25100-11	544	528.92	2.85	8.60	526.15	3.39	6.30	525.92	3.44	2.70
p25100-12	504	496.00	1.61	7.90	495.34	1.75	6.30	494.50	1.92	2.20
p25100-13	555	534.04	3.92	6.70	535.51	3.64	6.70	532.92	4.14	2.50
p25100-14	544	529.50	2.74	8.60	531.14	2.42	7.50	528.23	2.99	2.40
p25100-15	583	572.68	1.80	8.30	570.58	2.18	6.20	568.22	2.60	2.40
p25100-16	534	520.11	2.67	8.00	516.19	3.45	7.20	514.91	3.71	3.30
p25100-17	542	535.84	1.15	9.00	534.92	1.32	8.10	533.53	1.59	2.50
p25100-18	508	501.00	1.40	7.90	500.29	1.54	5.00	500.20	1.56	2.10
p25100-19	551	530.67	3.83	7.20	531.03	3.76	5.40	530.45	3.87	2.40
p25100-20	653*	522.63	24.95	10.00	510.24	27.98	8.90	510.24	27.98	2.80
p33100-11	414	405.83	2.01	7.60	406.66	1.80	5.80	404.43	2.37	2.50
p33100-12	391	374.75	4.34	6.80	375.05	4.25	3.90	373.43	4.71	2.00
p33100-13	446	440.50	1.25	6.10	441.89	0.93	5.40	439.26	1.53	2.10
p33100-14	447	434.00	3.00	7.20	431.94	3.49	6.40	431.80	3.52	2.60
p33100-15	474	469.38	0.99	7.30	471.47	0.54	6.60	468.96	1.07	2.50
p33100-16	447	430.43	3.85	7.30	432.25	3.41	7.50	428.96	4.20	2.80
p33100-17	431	423.33	1.81	7.20	423.66	1.73	3.80	423.33	1.81	2.10
p33100-18	456	430.69	5.88	8.00	438.64	3.96	9.80	429.14	6.26	2.40
p33100-19	445	430.67	3.33	7.50	431.77	3.06	5.20	429.26	3.67	2.30
p33100-20	460	450.96	2.00	8.40	448.69	2.52	10.80	442.70	3.91	3.20
p40100-11	415	405.83	2.26	7.90	404.90	2.49	8.00	397.27	4.46	2.30
p40100-12	377	364.25	3.50	7.00	363.00	3.86	6.60	360.94	4.45	2.90
p40100-13	412	404.83	1.77	6.50	406.67	1.31	6.30	404.35	1.89	2.70
p40100-14	421	412.69	2.01	6.40	411.53	2.30	6.50	410.64	2.52	2.90
p40100-15	496	488.83	1.47	7.70	487.38	1.77	10.50	481.40	3.03	3.70
p40100-16	428	424.30	0.87	7.00	421.04	1.65	9.20	420.30	1.83	3.20
p40100-17	440	430.00	2.33	8.40	428.31	2.73	8.80	428.08	2.78	3.50
p40100-18	450	433.40	3.83	10.10	431.82	4.21	8.30	431.33	4.33	2.90
p40100-19	450	439.05	2.49	6.80	439.37	2.42	8.30	434.99	3.45	2.90
p40100-20	486	475.33	2.24	8.10	460.52	5.53	9.20	466.17	4.25	3.30
mean		573.86	2.63	9.42	572.34	2.89	6.35	570.67	3.24	2.70

Table D.32: Comparison of the dual bound obtained by SCIP and GCG on test set CPMP100

instance	opt	GCG db	GCG gap	GCG time	SCIP all cuts db	SCIP all cuts gap	SCIP all cuts time	SCIP no base db	SCIP no base gap	SCIP no base time
p15150-21	1288	1281.83	0.48	35.90	1279.79	0.64	9.80	1279.36	0.68	7.00
p15150-22	1256	1247.81	0.66	52.00	1244.08	0.96	9.10	1243.73	0.99	7.50
p15150-23	1279	1277.86	0.09	38.00	1274.54	0.35	9.30	1274.62	0.34	6.00
p15150-24	1220	1218.38	0.13	40.10	1214.99	0.41	8.60	1215.04	0.41	5.90
p15150-25	1193	1188.86	0.35	31.70	1187.02	0.50	8.80	1187.02	0.50	5.80
p15150-26	1264	1258.83	0.41	53.00	1256.61	0.59	7.90	1256.11	0.63	6.00
p15150-27	1323	1311.81	0.85	74.90	1304.63	1.41	13.70	1303.16	1.52	9.00
p15150-28	1233	1230.50	0.20	35.50	1228.35	0.38	7.90	1228.35	0.38	5.20
p15150-29	1219	1219.00	0.00	36.50	1218.44	0.05	7.50	1218.44	0.05	4.90
p15150-30	1201	1200.33	0.06	35.00	1198.29	0.23	11.00	1198.60	0.20	6.70
p37150-21	681	674.32	0.99	23.20	672.89	1.21	17.70	672.50	1.26	6.90
p37150-22	660	645.27	2.28	29.10	643.65	2.54	19.10	641.36	2.91	6.60
p37150-23	704*	642.07	9.65	26.20	633.80	11.08	19.60	634.23	11.00	7.40
p37150-24	594	586.83	1.22	21.60	584.55	1.62	15.70	584.02	1.71	7.20
p37150-25	629	627.17	0.29	25.40	626.85	0.34	16.90	626.16	0.45	6.00
p37150-26	653	646.06	1.07	24.30	645.60	1.15	16.50	644.16	1.37	7.80
p37150-27	736	713.36	3.17	23.90	703.63	4.60	19.40	702.99	4.70	8.10
p37150-28	644	632.12	1.88	25.80	634.03	1.57	18.80	631.24	2.02	7.40
p37150-29	649	640.64	1.31	23.40	640.38	1.35	14.90	639.29	1.52	6.40
p37150-30	630	619.00	1.78	24.10	618.89	1.80	19.80	617.71	1.99	7.50
p50150-21	599	580.61	3.17	19.60	585.63	2.28	25.80	576.98	3.82	8.30
p50150-22	561	539.85	3.92	19.80	542.90	3.33	28.10	536.92	4.48	8.90
p50150-23	564	550.86	2.38	19.90	542.79	3.91	19.60	544.17	3.64	7.80
p50150-24	505	495.17	1.99	23.30	496.01	1.81	18.60	493.33	2.37	7.30
p50150-25	488	485.17	0.58	20.40	485.91	0.43	11.80	484.71	0.68	7.10
p50150-26	540	525.67	2.73	24.10	532.31	1.45	14.20	525.09	2.84	6.50
p50150-27	579	567.35	2.05	21.30	563.69	2.72	19.20	564.11	2.64	8.80
p50150-28	503	499.00	0.80	24.40	498.67	0.87	16.50	498.65	0.87	7.80
p50150-29	545	529.31	2.96	21.40	532.63	2.32	24.60	527.13	3.39	8.60
p50150-30	502	487.33	3.01	20.60	488.30	2.81	18.40	486.17	3.26	6.90
p60150-21	552	544.92	1.30	23.70	542.30	1.79	23.10	541.01	2.03	11.20
p60150-22	601	587.73	2.26	29.30	577.47	4.08	32.80	570.57	5.33	13.40
p60150-23	555	539.64	2.85	28.00	528.40	5.04	27.90	523.82	5.95	9.20
p60150-24	487	474.10	2.72	22.30	472.31	3.11	24.40	471.14	3.37	9.90
p60150-25	436	427.71	1.94	24.10	432.66	0.77	21.20	427.23	2.05	8.00
p60150-26	512	505.50	1.29	23.30	508.31	0.73	21.70	503.29	1.73	9.30
p60150-27	757	714.50	5.95	28.50	689.86	9.73	30.90	679.43	11.42	11.80
p60150-28	471	462.40	1.86	25.60	461.35	2.09	15.30	460.46	2.29	8.60
p60150-29	494	488.00	1.23	26.70	488.77	1.07	17.40	487.67	1.30	9.20
p60150-30	444	434.83	2.11	21.00	437.34	1.52	18.10	434.03	2.30	8.00
mean		699.81	1.83	27.57	697.95	2.09	16.79	695.57	2.48	7.72

Table D.33: Comparison of the dual bound obtained by SCIP and GCG on test set CPMP150

instance	opt	GCG db	GCG gap	GCG time	SCIP all cuts db	SCIP all cuts gap	SCIP all cuts time	SCIP no base db	SCIP no base gap	SCIP no base time
p20200-31	1378	1372.09	0.43	89.30	1369.89	0.59	21.10	1369.94	0.59	11.30
p20200-32	1424	1409.92	1.00	260.20	1403.15	1.49	22.50	1403.88	1.43	15.50
p20200-33	1367	1361.85	0.38	73.60	1357.70	0.68	23.10	1358.93	0.59	12.30
p20200-34	1385	1374.55	0.76	114.00	1372.29	0.93	28.80	1371.99	0.95	10.90
p20200-35	1437	1430.73	0.44	72.90	1427.44	0.67	25.10	1427.81	0.64	12.70
p20200-36	1382	1378.33	0.27	183.90	1376.21	0.42	25.20	1376.38	0.41	11.20
p20200-37	1458	1454.25	0.26	83.80	1453.09	0.34	23.60	1452.69	0.37	12.90
p20200-38	1382	1372.31	0.71	106.10	1368.66	0.97	21.70	1368.66	0.97	11.90
p20200-39	1374	1369.69	0.31	72.90	1367.85	0.45	24.30	1367.70	0.46	10.90
p20200-40	1416	1413.19	0.20	84.30	1411.54	0.32	28.10	1412.29	0.26	13.10
p50200-31	728*	711.75	2.28	62.50	711.44	2.33	46.70	707.85	2.85	15.10
p50200-32	878*	799.28	9.85	52.20	777.23	12.96	51.50	772.09	13.72	12.10
p50200-33	760*	693.65	9.57	52.40	695.31	9.30	46.20	688.19	10.43	10.60
p50200-34	858*	778.73	10.18	64.00	758.73	13.08	43.80	759.95	12.90	17.30
p50200-35	752*	727.11	3.42	66.00	724.07	3.86	50.00	718.48	4.67	12.10
p50200-36	701	692.22	1.27	64.50	693.09	1.14	53.30	687.40	1.98	15.20
p50200-37	753	743.40	1.29	46.30	744.28	1.17	28.30	740.92	1.63	12.10
p50200-38	749*	729.96	2.61	67.10	724.57	3.37	42.60	723.89	3.47	14.30
p50200-39	722	712.62	1.32	67.90	715.60	0.89	33.20	711.60	1.46	12.80
p50200-40	779*	737.54	5.62	52.70	731.76	6.46	59.80	726.92	7.16	12.40
p66200-31	575	571.08	0.69	59.50	571.12	0.68	35.50	569.49	0.97	17.00
p66200-32	860*	699.15	23.01	41.80	682.61	25.99	77.60	673.05	27.78	18.70
p66200-33	648*	599.12	8.16	50.10	599.07	8.17	74.40	593.67	9.15	18.60
p66200-34	729*	683.50	6.66	54.90	672.28	8.44	46.10	664.06	9.78	16.30
p66200-35	615*	592.43	3.81	38.60	593.21	3.67	33.20	590.51	4.15	17.50
p66200-36	580	569.51	1.84	47.60	569.21	1.90	52.30	565.48	2.57	15.30
p66200-37	636*	616.50	3.16	46.60	616.49	3.16	34.10	614.93	3.43	15.20
p66200-38	608	595.35	2.12	44.00	592.81	2.56	50.20	591.62	2.77	21.50
p66200-39	589*	573.57	2.69	60.10	576.89	2.10	37.50	572.68	2.85	15.70
p66200-40	632*	607.90	3.96	50.20	613.99	2.93	62.20	603.89	4.66	20.70
p80200-31	541*	527.83	2.49	58.70	529.51	2.17	50.50	526.55	2.74	22.90
p80200-32	802*	780.48	2.76	50.60	725.41	10.56	73.60	714.55	12.24	23.30
p80200-33	557	547.83	1.67	46.70	546.28	1.96	62.30	543.62	2.46	25.90
p80200-34	845	834.14	1.30	63.00	789.29	7.06	75.00	779.52	8.40	24.70
p80200-35	552	542.00	1.85	44.00	544.98	1.29	57.30	537.68	2.66	18.80
p80200-36	551	537.92	2.43	55.00	537.79	2.46	61.50	536.06	2.79	25.30
p80200-37	594*	579.67	2.47	49.10	582.67	1.94	51.20	576.92	2.96	23.90
p80200-38	592	586.97	0.86	56.40	573.15	3.29	62.80	573.99	3.14	27.40
p80200-39	541*	527.94	2.47	44.00	530.07	2.06	47.30	524.19	3.21	15.30
p80200-40	595*	570.95	4.21	47.90	563.59	5.57	69.20	558.05	6.62	19.40
mean		798.87	3.19	63.09	793.63	3.88	42.72	789.77	4.44	16.11

Table D.34: Comparison of the dual bound obtained by SCIP and GCG on test set CPMP200

instance	opt	GCG db	GCG gap	GCG time	SCIP all cuts db	SCIP all cuts gap	SCIP all cuts time	SCIP no base db	SCIP no base gap	SCIP no base time
new1_1	30866	30883.00	0.06	51.60	31354.39	1.56	9.60	31788.39	2.90	5.20
new1_2	35771	35805.25	0.10	113.30	36446.63	1.85	11.60	36929.10	3.14	7.60
new1_3	40934	40963.00	0.07	140.50	41624.31	1.66	13.70	42055.83	2.67	8.40
new1_4	46180	46223.24	0.09	136.40	46972.04	1.69	13.70	47429.83	2.64	9.00
new1_5	50324	50340.17	0.03	184.70	51239.34	1.79	18.30	51905.05	3.05	11.80
new1_6	55495	55519.17	0.04	141.00	56361.14	1.54	18.80	56880.96	2.44	13.30
new1_7	59255	59274.00	0.03	161.90	60072.09	1.36	21.60	60663.66	2.32	13.60
new1_8	65465	65480.00	0.02	256.70	66470.83	1.51	25.30	67331.98	2.77	16.30
new1_9	69530	69573.62	0.06	262.50	70734.82	1.70	26.20	71481.88	2.73	19.40
new1_10	75756	75773.50	0.02	529.80	77119.39	1.77	33.10	77905.96	2.76	20.80
new2_1	3109	3110.50	0.05	65.00	3172.34	2.00	9.60	3210.66	3.17	5.30
new2_2	3725	3730.28	0.14	77.90	3789.92	1.71	11.60	3833.86	2.84	7.00
new2_3	4202	4206.50	0.11	118.80	4258.75	1.33	13.10	4291.74	2.09	8.40
new2_4	4703	4703.00	0.00	115.30	4770.61	1.42	16.90	4814.15	2.31	10.00
new2_5	5206	5210.83	0.09	187.60	5309.34	1.95	17.20	5365.89	2.98	11.30
new2_6	5616	5618.17	0.04	179.60	5715.27	1.74	21.30	5776.56	2.78	12.50
new2_7	6148	6151.25	0.05	176.60	6239.62	1.47	19.30	6291.32	2.28	15.80
new2_8	6630	6632.50	0.04	238.90	6745.41	1.71	25.70	6816.80	2.74	14.50
new2_9	7039	7042.75	0.05	265.40	7157.21	1.65	26.20	7227.81	2.61	16.90
new2_10	7811	7817.00	0.08	222.40	7931.14	1.51	30.20	8008.00	2.46	18.60
new3_1	71998	72017.17	0.03	245.40	72766.23	1.06	20.40	73297.00	1.77	12.30
new3_2	81898	81931.00	0.04	351.60	82718.49	0.99	22.50	83466.28	1.88	16.20
new3_3	97056	97069.83	0.01	626.60	98045.55	1.01	29.70	98714.67	1.68	20.90
new3_4	107491	107530.00	0.04	534.90	108824.10	1.23	39.80	109677.30	1.99	25.90
new3_5	120504	120519.00	0.01	605.40	122018.76	1.24	49.40	122734.65	1.82	35.00
new3_6	129039*	129145.33	0.08	864.00	130759.99	1.32	58.20	131653.05	1.99	37.30
new3_7	142486	142524.90	0.03	800.30	144297.33	1.26	68.30	145289.18	1.93	46.70
new3_8	151489	151517.50	0.02	784.60	153598.92	1.37	75.30	154891.20	2.20	58.60
new3_9	165038	165156.50	0.07	1034.30	167484.83	1.46	82.70	168855.03	2.26	60.50
new3_10	182813	182850.67	0.02	1416.30	185167.80	1.27	100.90	186651.68	2.06	75.90
new4_1	22044	22044.00	0.00	25.50	22086.15	0.19	3.30	22115.35	0.32	2.50
new4_2	26115	26115.00	0.00	17.60	26133.52	0.07	3.40	26163.61	0.19	2.90
new4_3	29110	29110.00	0.00	24.40	29141.13	0.11	4.40	29209.22	0.34	3.40
new4_4	32692	32692.00	0.00	71.60	32729.63	0.11	5.70	32785.22	0.28	4.10
new4_5	37016	37021.50	0.01	35.10	37071.88	0.15	6.10	37130.54	0.31	4.80
new4_6	39593	39593.00	0.00	57.30	39664.98	0.18	7.40	39720.35	0.32	5.80
new4_7	44735	44735.00	0.00	56.10	44781.16	0.10	7.90	44866.21	0.29	5.30
new4_8	48182	48184.00	0.00	72.30	48242.90	0.13	8.40	48289.78	0.22	6.20
new4_9	50559	50562.00	0.01	80.00	50642.44	0.16	10.90	50724.41	0.33	7.90
new4_10	54842	54851.00	0.02	52.10	54931.44	0.16	11.20	55020.05	0.32	8.50
new5_1	40982	40994.33	0.03	90.30	41388.98	0.98	10.60	41758.97	1.86	6.70
new5_2	47914	47921.50	0.02	116.60	48537.58	1.28	13.10	48941.31	2.10	8.60
new5_3	52447	52451.50	0.01	122.10	53022.17	1.08	15.40	53432.66	1.84	9.80
new5_4	59790	59818.00	0.05	195.40	60506.58	1.18	18.10	60992.72	1.97	10.80
new5_5	66179	66217.83	0.06	225.00	66989.14	1.21	19.90	67641.67	2.16	13.60
new5_6	75070	75130.67	0.08	367.50	75860.52	1.04	24.60	76421.70	1.77	15.20
new5_7	81982	82033.00	0.06	266.30	82842.52	1.04	29.20	83579.19	1.91	16.30
new5_8	85314	85338.00	0.03	252.20	86414.34	1.27	29.00	87034.59	1.98	18.90
new5_9	95037	95098.50	0.06	295.60	96102.56	1.11	33.20	96944.04	1.97	21.20
new5_10	100031	100059.83	0.03	352.70	101306.61	1.26	36.40	102112.50	2.04	23.60
new6_1	71426	71443.17	0.02	178.70	72153.55	1.01	20.50	72701.87	1.75	12.80
new6_2	82942	82947.50	0.01	325.70	83768.28	0.99	31.10	84265.22	1.57	15.70
new6_3	96115	96129.00	0.01	280.00	96887.04	0.80	26.90	97420.90	1.34	18.30
new6_4	110102	110116.00	0.01	531.10	111028.74	0.83	32.80	111747.42	1.47	20.60
new6_5	119233	119260.50	0.02	528.00	120206.51	0.81	40.30	120956.35	1.42	26.80
new6_6	128178	128186.17	0.01	599.40	129492.28	1.01	50.00	130364.96	1.68	30.80
new6_7	142056	142089.33	0.02	1252.90	143439.19	0.96	51.00	144392.78	1.62	35.70
new6_8	154745*	154840.75	0.06	759.60	156347.21	1.02	64.00	157514.09	1.76	44.60
new6_9	167916	167962.00	0.03	1001.40	169757.71	1.08	66.40	170749.89	1.66	52.80
new6_10	176884*	176929.20	0.03	1252.70	178998.94	1.18	95.20	180321.30	1.91	70.90
new7_1	42685	42706.67	0.05	97.50	43260.62	1.33	11.50	43809.70	2.57	7.90
new7_2	46526	46534.50	0.02	130.20	47176.94	1.38	13.80	47512.70	2.08	8.70
new7_3	54437	54452.50	0.03	180.30	55129.12	1.26	15.70	55509.58	1.93	13.60
new7_4	60719	60729.75	0.02	236.30	61713.70	1.61	19.40	62392.09	2.68	14.00
new7_5	68432	68449.50	0.03	291.00	69271.84	1.21	25.90	69794.23	1.95	16.00
new7_6	72337	72371.30	0.05	421.40	73353.34	1.39	28.10	73892.10	2.10	16.70
new7_7	80122	80177.33	0.07	370.50	81250.31	1.39	30.40	81853.27	2.12	19.50
new7_8	88460	88509.42	0.06	545.00	89899.36	1.60	35.50	90550.46	2.31	24.30
new7_9	92380	92389.50	0.01	428.60	93723.58	1.43	38.60	94561.84	2.31	26.00
new7_10	100915	100980.83	0.07	647.00	102611.43	1.65	44.60	103454.43	2.45	28.70
mean		49281.99	0.04	224.89	49839.63	1.17	23.54	50216.98	1.92	16.04

Table D.35: Comparison of the dual bound obtained by SCIP and GCG (32 constraints per block) on test set RAP

instance	opt	GCG db	GCG gap	GCG time	SCIP all cuts db	SCIP all cuts gap	SCIP all cuts time	SCIP no base db	SCIP no base gap	SCIP no base time
new1_1	30866	30869.00	0.01	85.80	31354.39	1.56	9.60	31788.39	2.90	5.20
new1_2	35771	35781.50	0.03	155.70	36446.63	1.85	11.60	36929.10	3.14	7.60
new1_3	40934	40942.83	0.02	267.10	41624.31	1.66	13.70	42055.83	2.67	8.40
new1_4	46180	46190.67	0.02	228.80	46972.04	1.69	13.70	47429.83	2.64	9.00
new1_5	50324	50334.50	0.02	234.10	51239.34	1.79	18.30	51905.05	3.05	11.80
new1_6	55495	55519.17	0.04	216.40	56361.14	1.54	18.80	56880.96	2.44	13.30
new1_7	59255	59255.00	0.00	463.10	60072.09	1.36	21.60	60663.66	2.32	13.60
new1_8	65465	65469.00	0.01	300.20	66470.83	1.51	25.30	67331.98	2.77	16.30
new1_9	69530	69530.00	0.00	384.90	70734.82	1.70	26.20	71481.88	2.73	19.40
new1_10	75756	75766.00	0.01	539.70	77119.39	1.77	33.10	77905.96	2.76	20.80
new2_1	3109	3110.50	0.05	96.30	3172.34	2.00	9.60	3210.66	3.17	5.30
new2_2	3725	3728.83	0.10	133.10	3789.92	1.71	11.60	3833.86	2.84	7.00
new2_3	4202	4202.50	0.01	130.60	4258.75	1.33	13.10	4291.74	2.09	8.40
new2_4	4703	4703.00	0.00	353.90	4770.61	1.42	16.90	4814.15	2.31	10.00
new2_5	5206	5210.83	0.09	396.90	5309.34	1.95	17.20	5365.89	2.98	11.30
new2_6	5616	5617.50	0.03	209.30	5715.27	1.74	21.30	5776.56	2.78	12.50
new2_7	6148	6149.50	0.02	225.30	6239.62	1.47	19.30	6291.32	2.28	15.80
new2_8	6630	6631.00	0.02	218.10	6745.41	1.71	25.70	6816.80	2.74	14.50
new2_9	7039	7039.00	0.00	283.10	7157.21	1.65	26.20	7227.81	2.61	16.90
new2_10	7811	7815.00	0.05	310.20	7931.14	1.51	30.20	8008.00	2.46	18.60
new3_1	71998	71998.00	0.00	1176.20	72766.23	1.06	20.40	73297.00	1.77	12.30
new3_2	81898	81923.00	0.03	1133.30	82718.49	0.99	22.50	83466.28	1.88	16.20
new3_3	97056	97060.00	0.00	3646.00	98045.55	1.01	29.70	98714.67	1.68	20.90
new3_4	107491	107506.50	0.01	816.00	108824.10	1.23	39.80	109677.30	1.99	25.90
new3_5	120504	120513.50	0.01	896.70	122018.76	1.24	49.40	122734.65	1.82	35.00
new3_6	129039*	129092.00	0.04	2489.50	130759.99	1.32	58.20	131653.05	1.99	37.30
new3_7	142486	142514.90	0.02	993.50	144297.33	1.26	68.30	145289.18	1.93	46.70
new3_8	151489	151502.33	0.01	1830.90	153598.92	1.37	75.30	154891.20	2.20	58.60
new3_9	165038	165097.50	0.04	6269.80	167484.83	1.46	82.70	168855.03	2.26	60.50
new3_10	182813	182815.83	0.00	6279.60	185167.80	1.27	100.90	186651.68	2.06	75.90
new4_1	22044	22044.00	0.00	15.30	22086.15	0.19	3.30	22115.35	0.32	2.50
new4_2	26115	26115.00	0.00	16.80	26133.52	0.07	3.40	26163.61	0.19	2.90
new4_3	29110	29110.00	0.00	27.90	29141.13	0.11	4.40	29209.22	0.34	3.40
new4_4	32692	32692.00	0.00	43.40	32729.63	0.11	5.70	32785.22	0.28	4.10
new4_5	37016	37021.50	0.01	36.10	37071.88	0.15	6.10	37130.54	0.31	4.80
new4_6	39593	39593.00	0.00	64.60	39664.98	0.18	7.40	39720.35	0.32	5.80
new4_7	44735	44735.00	0.00	52.80	44781.16	0.10	7.90	44866.21	0.29	5.30
new4_8	48182	48183.50	0.00	37.20	48242.90	0.13	8.40	48289.78	0.22	6.20
new4_9	50559	50562.00	0.01	77.50	50642.44	0.16	10.90	50724.41	0.33	7.90
new4_10	54842	54851.00	0.02	124.20	54931.44	0.16	11.20	55020.05	0.32	8.50
new5_1	40982	40982.00	0.00	125.50	41388.98	0.98	10.60	41758.97	1.86	6.70
new5_2	47914	47914.00	0.00	222.80	48537.58	1.28	13.10	48941.31	2.10	8.60
new5_3	52447	52451.50	0.01	219.70	53022.17	1.08	15.40	53432.66	1.84	9.80
new5_4	59790	59794.00	0.01	187.50	60506.58	1.18	18.10	60992.72	1.97	10.80
new5_5	66179	66209.33	0.05	278.40	66989.14	1.21	19.90	67641.67	2.16	13.60
new5_6	75070	75110.17	0.05	463.40	75860.52	1.04	24.60	76421.70	1.77	15.20
new5_7	81982	82000.00	0.02	306.40	82842.52	1.04	29.20	83579.19	1.91	16.30
new5_8	85314	85314.00	0.00	545.30	86414.34	1.27	29.00	87034.59	1.98	18.90
new5_9	95037	95073.00	0.04	740.50	96102.56	1.11	33.20	96944.04	1.97	21.20
new5_10	100031	100053.00	0.02	412.80	101306.61	1.26	36.40	102112.50	2.04	23.60
new6_1	71426	71438.67	0.02	267.00	72153.55	1.01	20.50	72701.87	1.75	12.80
new6_2	82942	82946.50	0.01	506.10	83768.28	0.99	31.10	84265.22	1.57	15.70
new6_3	96115	96120.00	0.01	258.10	96887.04	0.80	26.90	97420.90	1.34	18.30
new6_4	110102	110102.00	0.00	613.00	111028.74	0.83	32.80	111747.42	1.47	20.60
new6_5	119233	119247.50	0.01	704.20	120206.51	0.81	40.30	120956.35	1.42	26.80
new6_6	128178	128182.67	0.00	1104.00	129492.28	1.01	50.00	130364.96	1.68	30.80
new6_7	142056	142074.33	0.01	793.60	143439.19	0.96	51.00	144392.78	1.62	35.70
new6_8	154745*	154793.25	0.03	1539.90	156347.21	1.02	64.00	157514.09	1.76	44.60
new6_9	167916	167930.50	0.01	1715.00	169757.71	1.08	66.40	170749.89	1.66	52.80
new6_10	176884*	176916.20	0.02	3062.60	178998.94	1.18	95.20	180321.30	1.91	70.90
new7_1	42685	42686.50	0.00	178.10	43260.62	1.33	11.50	43809.70	2.57	7.90
new7_2	46526	46533.50	0.02	196.80	47176.94	1.38	13.80	47512.70	2.08	8.70
new7_3	54437	54445.00	0.01	323.90	55129.12	1.26	15.70	55509.58	1.93	13.60
new7_4	60719	60729.75	0.02	1783.70	61713.70	1.61	19.40	62392.09	2.68	14.00
new7_5	68432	68449.50	0.03	448.80	69271.84	1.21	25.90	69794.23	1.95	16.00
new7_6	72337	72346.50	0.01	994.10	73353.34	1.39	28.10	73892.10	2.10	16.70
new7_7	80122	80154.33	0.04	551.70	81250.31	1.39	30.40	81853.27	2.12	19.50
new7_8	88460	88478.00	0.02	1052.30	89899.36	1.60	35.50	90550.46	2.31	24.30
new7_9	92380	92385.50	0.01	1171.00	93723.58	1.43	38.60	94561.84	2.31	26.00
new7_10	100915	100969.50	0.05	2459.90	102611.43	1.65	44.60	103454.43	2.45	28.70
mean		49272.67	0.02	366.01	49839.63	1.17	23.54	50216.98	1.92	16.04

Table D.36: Comparison of the dual bound obtained by SCIP and GCG (64 constraints per block) on test set RAP

		GCG			SCIP all cuts			SCIP no base		
instance	opt	db	gap	time	db	gap	time	db	gap	time
1-FullIns_3	4	4.00	0.00	1.00	3.00	33.33	0.20	3.00	33.33	0.10
2-FullIns_3	5	4.25	17.65	2.00	3.00	66.67	0.50	3.00	66.67	0.20
3-FullIns_3	6	5.20	15.38	1.60	3.00	100.00	1.70	3.00	100.00	0.60
4-FullIns_3	7	6.17	13.51	3.50	6.00	16.67	9.60	5.00	40.00	1.40
5-FullIns_3	8	7.14	12.00	4.50	3.00	166.67	5.50	3.00	166.67	1.80
anna	11	11.00	0.00	1.10	11.00	0.00	2.40	11.00	0.00	1.60
david	11	11.00	0.00	0.60	11.00	0.00	0.20	11.00	0.00	0.20
fpsol2.i.1	65	65.00	0.00	182.80	51.00	27.45	3600.00	51.00	27.45	3600.00
games120	9	9.00	0.00	0.90	9.00	0.00	1.60	9.00	0.00	1.00
homer	13	13.00	0.00	10.10	10.00	30.00	18.00	10.00	30.00	15.80
huck	11	11.00	0.00	0.40	8.00	37.50	0.90	11.00	0.00	0.50
jean	10	10.00	0.00	0.30	4.00	150.00	2.90	4.00	150.00	0.60
miles1000	42	42.00	0.00	17.40	37.00	13.51	115.70	37.00	13.51	80.90
miles1500	73	73.00	0.00	46.20	69.00	5.80	724.90	69.00	5.80	363.80
miles250	8	8.00	0.00	0.80	6.00	33.33	2.70	6.00	33.33	1.10
miles500	20	20.00	0.00	2.20	19.00	5.26	10.30	19.00	5.26	5.70
miles750	31	31.00	0.00	7.30	24.00	29.17	55.10	24.00	29.17	41.20
mulsol.i.1	49	49.00	0.00	28.70	47.00	4.26	121.20	47.00	4.26	64.40
mulsol.i.2	31	31.00	0.00	36.60	30.00	3.33	69.90	30.00	3.33	54.20
mulsol.i.3	31	31.00	0.00	37.70	30.00	3.33	234.70	30.00	3.33	177.00
mulsol.i.4	31	31.00	0.00	43.70	30.00	3.33	320.20	30.00	3.33	267.40
mulsol.i.5	31	31.00	0.00	39.30	30.00	3.33	325.10	30.00	3.33	286.00
myciel4	5	3.24	54.09	2.10	3.00	66.67	0.20	2.29	118.75	0.10
queen6_6	7	7.00	0.00	3.50	6.00	16.67	1.20	6.00	16.67	0.50
queen7_7	7	7.00	0.00	4.50	7.00	0.00	3.10	7.00	0.00	1.40
zeroin.i.1	49	49.00	0.00	28.70	45.00	8.89	196.50	45.00	8.89	180.90
zeroin.i.2	30	30.00	0.00	28.60	28.00	7.14	156.50	28.00	7.14	144.10
zeroin.i.3	30	30.00	0.00	28.40	28.00	7.14	153.30	28.00	7.14	147.60
mean		22.17	3.55	11.38	19.75	24.81	35.66	19.79	25.42	28.50

Table D.37: Comparison of the dual bound obtained by `SCIP` and `GCG` on test set COLORING

		GCG			SCIP all cuts			SCIP no base		
instance	opt	db	gap	time	db	gap	time	db	gap	time
N1C1W1_A	25	25.00	0.00	0.70	24.36	2.63	0.70	24.36	2.63	0.30
N1C1W1_K	26	26.00	0.00	0.40	23.97	8.47	0.70	23.97	8.47	0.40
N1C1W2_B	30	30.00	0.00	0.70	29.49	1.73	0.50	29.49	1.73	0.30
N1C1W2_L	31	31.00	0.00	0.60	30.44	1.84	0.80	30.44	1.84	0.50
N1C1W4_C	36	36.00	0.00	0.60	35.07	2.65	0.70	36.00	0.00	0.50
N1C1W4_M	41	41.00	0.00	0.60	36.09	13.60	0.90	36.09	13.60	0.60
N1C2W1_D	21	21.00	0.00	0.50	20.52	2.36	0.50	20.52	2.36	0.30
N1C2W1_N	21	21.00	0.00	1.10	19.93	5.40	0.60	19.93	5.40	0.30
N1C2W2_E	33	33.00	0.00	0.50	28.07	17.54	0.80	28.07	17.54	0.50
N1C2W2_O	29	29.00	0.00	0.40	25.80	12.40	0.90	25.80	12.40	0.40
N1C2W4_F	32	32.00	0.00	0.50	28.67	11.63	0.70	28.67	11.63	0.40
N1C2W4_P	28	28.00	0.00	0.70	27.07	3.42	0.80	27.07	3.42	0.40
N1C3W1_G	15	14.71	1.99	0.80	14.71	1.99	0.30	14.71	1.99	0.20
N1C3W1_Q	20	20.00	0.00	0.90	19.65	1.76	0.70	19.65	1.76	0.20
N1C3W2_H	23	23.00	0.00	0.90	21.86	5.22	0.60	21.86	5.22	0.30
N1C3W2_R	19	18.82	0.94	1.20	18.81	0.99	0.60	18.81	0.99	0.20
N1C3W4_I	23	22.17	3.76	0.60	21.78	5.60	0.90	21.78	5.60	0.30
N1C3W4_S	22	22.00	0.00	0.60	21.29	3.32	0.70	21.29	3.32	0.30
mean		26.28	0.37	0.68	24.84	5.59	0.69	24.89	5.44	0.35

Table D.38: Comparison of the dual bound obtained by `SCIP` and `GCG` on test set BINDATA1-N1S

		GCG			SCIP all cuts			SCIP no base		
instance	opt	db	gap	time	db	gap	time	db	gap	time
N2C1W1_A	48	47.33	1.41	2.90	46.72	2.74	2.80	46.72	2.74	2.10
N2C1W1_K	55	55.00	0.00	2.20	52.79	4.19	4.10	52.79	4.19	2.80
N2C1W2_B	61	61.00	0.00	2.70	59.82	1.97	3.50	59.82	1.97	2.50
N2C1W2_L	62	62.00	0.00	2.60	60.13	3.11	4.20	60.13	3.11	2.50
N2C1W4_C	77	77.00	0.00	2.40	73.54	4.70	3.30	73.54	4.70	2.70
N2C1W4_M	72	72.00	0.00	2.40	69.97	2.90	3.50	69.97	2.90	2.70
N2C2W1_D	42	42.00	0.00	7.30	41.55	1.08	2.50	41.55	1.08	1.80
N2C2W1_N	43	43.00	0.00	4.20	42.00	2.38	2.50	42.00	2.38	1.70
N2C2W2_E	54	54.00	0.00	2.10	50.01	7.98	3.90	50.01	7.98	2.70
N2C2W2_O	50	49.75	0.50	3.20	48.41	3.29	3.30	48.41	3.29	2.50
N2C2W4_F	57	57.00	0.00	3.70	56.49	0.90	3.50	56.49	0.90	2.50
N2C2W4_P	60	60.00	0.00	3.00	56.64	5.93	3.60	56.64	5.93	2.50
N2C3W1_G	33	32.51	1.52	2.80	32.51	1.52	1.70	32.51	1.52	1.20
N2C3W1_Q	34	33.67	0.99	2.80	33.67	0.99	1.80	33.67	0.99	1.20
N2C3W2_H	38	37.11	2.39	3.30	37.11	2.39	2.60	37.11	2.39	1.20
N2C3W2_R	40	40.00	0.00	3.30	39.06	2.41	2.20	39.06	2.41	1.80
N2C3W4_I	44	43.39	1.41	4.80	43.19	1.87	3.00	43.19	1.87	1.80
N2C3W4_S	42	41.15	2.06	10.10	41.08	2.24	2.60	41.08	2.24	2.10
mean		50.30	0.57	3.54	49.03	2.91	3.01	49.03	2.91	2.12

Table D.39: Comparison of the dual bound obtained by `SCIP` and `GCG` on test set BINDATA1-N2S

		GCG			SCIP all cuts			SCIP no base		
instance	opt	db	gap	time	db	gap	time	db	gap	time
N3C1W1_A	105	105.00	0.00	12.40	100.88	4.08	22.50	100.88	4.08	18.10
N3C1W1_K	102	102.00	0.00	13.10	99.51	2.50	20.20	99.51	2.50	17.10
N3C1W2_B	126	126.00	0.00	13.20	118.37	6.45	29.00	118.37	6.45	24.50
N3C1W2_L	136	136.00	0.00	15.40	131.23	3.63	30.90	131.23	3.63	26.50
N3C1W4_C	146	146.00	0.00	18.10	142.66	2.34	31.40	142.66	2.34	25.10
N3C1W4_M	149	149.00	0.00	17.90	144.27	3.28	31.40	144.27	3.28	26.20
N3C2W1_D	85	85.00	0.00	9.60	83.19	2.17	14.90	83.19	2.17	12.10
N3C2W1_N	91	91.00	0.00	10.40	86.70	4.96	19.60	86.70	4.96	16.00
N3C2W2_E	116	116.00	0.00	13.50	107.22	8.18	26.90	107.22	8.18	21.20
N3C2W2_O	107	107.00	0.00	13.30	101.40	5.52	23.90	101.40	5.52	20.60
N3C2W4_F	115	115.00	0.00	20.90	110.63	3.95	25.30	110.63	3.95	20.60
N3C2W4_P	122	122.00	0.00	18.50	115.76	5.39	27.60	115.76	5.39	22.70
N3C3W1_G	65	64.16	1.31	10.30	64.16	1.31	9.70	64.16	1.31	8.40
N3C3W1_Q	73	72.61	0.54	11.80	72.61	0.54	13.20	72.61	0.54	10.30
N3C3W2_H	82	81.85	0.19	33.80	81.82	0.22	17.90	81.82	0.22	12.50
N3C3W2_R	79	78.51	0.63	14.20	78.51	0.63	14.60	78.51	0.63	12.40
N3C3W4_I	92	91.25	0.82	14.00	90.09	2.12	20.60	90.09	2.12	16.90
N3C3W4_S	84	83.08	1.11	32.70	83.03	1.16	16.50	83.03	1.16	13.00
mean		103.48	0.25	15.59	100.25	3.22	21.31	100.25	3.22	17.44

Table D.40: Comparison of the dual bound obtained by `SCIP` and `GCG` on test set BINDATA1-N3S

instance	default				20 vars				10%				no limit			
	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
p550-1	3491	0.0	32	11.9	4009	0.0	24	13.0	3274	0.0	21	14.6	3117	0.0	23	10.8
p550-3	3526	0.0	3	6.5	3526	0.0	3	6.5	3531	0.0	3	8.8	4030	0.0	3	6.6
p550-5	3876	0.0	1	7.2	3876	0.0	1	7.2	3876	0.0	1	7.1	3639	0.0	1	5.6
p550-7	3735	0.0	15	13.2	3737	0.0	16	14.0	3844	0.0	21	30.6	3933	0.0	15	13.7
p550-9	2925	0.0	9	6.1	2925	0.0	9	6.1	2929	0.0	5	6.6	3118	0.0	7	6.7
p1250-1	2168	0.0	173	18.8	2090	0.0	157	17.0	1956	0.0	141	14.2	2044	0.0	178	17.7
p1250-3	1825	0.0	155	18.7	1824	0.0	155	18.8	1810	0.0	137	17.2	1802	0.0	69	10.2
p1250-5	2204	0.0	68	11.7	2199	0.0	68	11.5	2324	0.0	134	21.2	2340	0.0	87	15.8
p1250-7	3019	0.0	1514	174.3	2714	0.0	1332	167.7	2677	0.0	1676	188.4	4041	0.0	1921	215.6
p1250-9	2085	0.0	138	36.9	2030	0.0	143	31.4	1982	0.0	123	29.1	2092	0.0	97	24.0
p1650-1	1484	0.0	8	2.6	1487	0.0	8	2.7	1469	0.0	8	2.6	1526	0.0	5	2.3
p1650-3	1418	0.0	13	2.8	1418	0.0	13	2.8	1434	0.0	10	3.3	1403	0.0	9	2.6
p1650-5	1752	0.0	59	8.5	1761	0.0	59	8.7	1705	0.0	78	9.8	1766	0.0	99	14.4
p1650-7	1325	0.0	75	9.8	1325	0.0	75	9.8	1334	0.0	81	10.2	1390	0.0	53	7.8
p1650-9	1567	0.0	198	26.7	1567	0.0	198	26.7	1552	0.0	195	25.9	1847	0.0	150	22.1
p2050-1	1285	0.0	38	4.5	1285	0.0	38	4.5	1272	0.0	36	4.6	1231	0.0	57	5.3
p2050-3	1282	0.0	11	3.5	1282	0.0	11	3.5	1288	0.0	11	3.6	1221	0.0	12	3.2
p2050-5	1262	0.0	5	2.9	1262	0.0	5	2.9	1262	0.0	5	2.8	1259	0.0	6	3.4
p2050-7	1166	0.0	5	3.0	1166	0.0	5	3.0	1166	0.0	5	2.9	1181	0.0	7	3.3
p2050-9	1356	0.0	343	46.3	1356	0.0	343	46.3	1320	0.0	331	44.8	1364	0.0	355	46.7
total	42k	0.0	2863	415.9	42k	0.0	2663	404.1	42k	0.0	3022	448.3	44k	0.0	3154	437.8
timeouts				0/20				0/20				0/20				0/20
mean	2016.4	0.0	44.9	12.8	2014.9	0.0	43.9	12.7	1983.0	0.0	45.0	13.9	2075.0	0.0	42.7	12.4

Table D.41. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set CPMP50S)

	default				20 vars				10%				no limit			
instance	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
p10100-11	7892	0.0	35	62.6	9079	0.0	52	105.9	7452	0.0	15	74.5	10527	0.0	44	86.8
p10100-13	6771	0.0	7	19.5	7272	0.0	9	25.6	7691	0.0	12	83.8	8766	0.0	11	37.8
p10100-15	11050	0.0	235	244.3	10960	0.0	239	286.7	11490	0.0	228	527.0	11460	0.0	212	257.5
p10100-17	9977	0.0	134	144.8	8218	0.0	96	114.0	8796	0.0	141	214.5	10052	0.0	96	117.9
p10100-19	6958	0.0	19	32.7	7126	0.0	25	42.6	7009	0.0	26	86.6	8160	0.0	31	48.7
p25100-11	5465	0.0	5202	862.4	5558	0.0	4785	831.8	5440	0.0	3771	649.2	5054	0.0	4017	656.2
p25100-13	6879	1.4	26451	3600.0	6671	0.9	27786	3600.0	5659	0.3	29760	3600.0	7375	0.7	26999	3600.0
p25100-15	4333	0.0	778	201.9	4349	0.0	754	193.9	4065	0.0	717	189.6	3833	0.0	407	107.1
p25100-17	4155	0.0	47	28.3	3953	0.0	44	26.9	3971	0.0	55	41.7	3731	0.0	55	27.6
p25100-19	6413	0.0	8501	1623.1	6302	0.0	9621	1830.1	6124	0.0	7772	1352.5	6203	0.0	8194	1551.9
p33100-11	3072	0.0	558	83.6	3151	0.0	521	84.4	2890	0.0	566	82.2	3168	0.0	659	100.5
p33100-13	3028	0.0	185	36.8	2899	0.0	130	30.4	2846	0.0	249	47.0	3112	0.0	157	35.7
p33100-15	3604	0.0	250	56.2	3646	0.0	392	72.6	3395	0.0	298	55.2	3687	0.0	179	41.2
p33100-17	3064	0.0	218	47.6	3054	0.0	196	46.6	3023	0.0	152	39.7	3195	0.0	331	64.5
p33100-19	4066	0.0	4076	603.2	4130	0.0	4940	738.9	4237	0.0	4478	670.0	4480	0.0	5026	787.7
p40100-11	3257	0.0	7957	1114.1	3260	0.0	6973	981.8	2989	0.0	1848	271.2	3250	0.0	3159	452.8
p40100-13	2706	0.0	623	82.3	2688	0.0	623	82.2	2652	0.0	749	88.6	2786	0.0	815	98.5
p40100-15	3132	0.0	828	153.2	3132	0.0	828	153.1	2948	0.0	878	143.4	3102	0.0	607	113.9
p40100-17	3807	0.0	2218	422.1	3712	0.0	1799	357.9	3525	0.0	2079	377.6	3459	0.0	1179	249.1
p40100-19	3424	0.0	3173	531.6	3373	0.0	2543	422.4	3353	0.0	2552	438.1	3467	0.0	2515	448.0
total	103k	1.4	61k	9950.3	102k	0.9	62k	10027.8	99k	0.3	56k	9032.4	108k	0.7	54k	8883.4
timeouts				1/20				1/20				1/20				1/20
mean	4762.0	0.1	587.1	184.7	4742.5	0.0	584.1	190.1	4584.1	0.0	540.8	199.5	4928.7	0.0	531.2	172.8

Table D.42. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set CPMP100S)

instance	default				20 vars				10%				no limit			
	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
p15150-21	12150	0.0	84	162.6	12057	0.0	89	165.2	10958	0.0	94	290.7	14470	0.0	127	175.0
p15150-23	9818	0.0	3	43.0	9818	0.0	3	42.8	9817	0.0	3	42.9	9502	0.0	3	41.7
p15150-25	11392	0.0	116	125.3	12690	0.0	153	169.7	13536	0.0	184	459.1	13578	0.0	138	167.2
p15150-27	28300	0.0	1688	2251.4	27620	0.2	2980	3600.0	28151	0.5	1765	3600.0	26332	0.0	906	1545.2
p15150-29	9734	0.0	1	39.2	9734	0.0	1	39.3	9734	0.0	1	39.2	9913	0.0	1	38.6
p37150-21	6005	0.0	207	105.1	6005	0.0	218	108.0	5934	0.0	210	118.6	6838	0.0	206	122.8
p37150-23	12307	3.0	8303	3600.0	13501	16.0	7422	3600.0	11371	6.3	9274	3600.0	11647	2.5	8574	3600.0
p37150-25	5426	0.0	55	41.5	5457	0.0	65	41.0	5349	0.0	58	43.6	5599	0.0	45	45.3
p37150-27	10412	0.9	7682	3600.0	10127	1.9	7642	3600.0	9978	1.6	8206	3600.0	12273	1.7	6991	3600.0
p37150-29	5876	0.0	650	215.9	5762	0.0	702	225.2	5701	0.0	754	241.2	6535	0.0	702	223.1
p50150-21	6543	1.9	12817	3600.0	7427	9.5	13163	3600.0	6416	1.6	13643	3600.0	6859	3.6	13234	3600.0
p50150-23	5779	0.0	3408	1154.2	6195	0.0	3692	1340.1	5231	0.0	3639	1157.2	6067	0.0	4129	1266.4
p50150-25	4418	0.0	56	37.5	4418	0.0	56	37.9	4450	0.0	100	52.3	4699	0.0	37	34.1
p50150-27	6510	0.0	3284	1314.7	6099	0.0	1789	729.2	5569	0.0	1739	671.5	7115	0.0	2717	1164.2
p50150-29	6386	1.0	15713	3600.0	6198	1.8	15253	3600.0	5936	0.9	16493	3600.0	6433	1.3	15821	3600.0
p60150-21	4273	0.0	463	175.3	4268	0.0	463	175.4	3989	0.0	315	118.2	4349	0.0	290	121.7
p60150-23	5635	0.0	5148	1648.0	5364	0.0	5852	1999.3	5036	0.0	6541	2040.3	5260	0.0	6791	2285.0
p60150-25	5475	0.0	18501	2985.0	5409	0.0	14488	2329.6	5202	0.4	23953	3600.0	5173	0.0	1371	326.2
p60150-27	6688	6.5	7646	3600.0	7307	12.2	7068	3600.0	6220	6.6	7225	3600.0	7061	12.7	7104	3600.0
p60150-29	4418	0.0	512	133.9	4430	0.0	523	138.6	4395	0.0	504	137.6	4718	0.0	544	139.2
total	167k	13.3	86k	28432.6	169k	41.6	81k	29141.3	162k	17.9	94k	30612.4	174k	21.8	69k	25695.7
timeouts				5/20				6/20				7/20				5/20
mean	7489.2	0.7	847.6	493.5	7588.6	2.0	861.6	502.9	7216.7	0.9	894.4	551.7	7835.5	1.1	716.3	443.4

Table D.43. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set CPMP150S)

	default				20 vars				10%				no limit			
instance	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
p20200-32	30084	0.0	889	2482.8	22349	0.0	831	1905.4	30470	0.6	480	3600.0	26660	0.0	794	1925.0
p20200-34	22962	0.0	508	1187.2	23010	0.0	502	1326.1	24716	0.0	547	2789.8	25236	0.0	551	1216.5
p20200-36	14833	0.0	62	293.0	14318	0.0	30	256.0	15643	0.0	84	589.8	15986	0.0	38	288.3
p50200-36	9841	0.0	2488	1234.3	10003	0.0	1988	1054.2	9408	0.0	2824	1376.9	8851	0.0	2356	1012.0
p50200-37	9149	0.0	2650	1067.9	9218	0.0	2868	1169.6	8746	0.0	3026	1178.9	9460	0.0	3644	1314.9
p50200-39	8610	0.0	1176	589.7	8445	0.0	1253	561.6	8056	0.0	973	532.4	8710	0.0	1087	547.3
p66200-31	6769	0.0	1148	377.0	6841	0.0	1024	358.1	6543	0.0	1098	358.6	7575	0.0	3135	750.7
p66200-36	8097	0.0	4533	1606.2	8252	0.0	5009	1877.2	7580	0.0	3858	1364.9	8327	0.0	5169	1833.5
p66200-38	9096	0.6	8346	3600.0	8828	2.5	8622	3600.0	8894	3.8	8594	3600.0	9295	1.9	8587	3600.0
p80200-33	6706	0.8	10218	3600.0	6301	0.7	11252	3600.0	6361	1.0	10938	3600.0	6664	0.3	11043	3600.0
p80200-34	8063	0.9	4554	3600.0	7645	0.7	4988	3600.0	7697	1.7	4671	3600.0	7530	0.8	4660	3600.0
p80200-38	6642	0.0	1656	695.6	6813	0.0	2187	950.0	6449	0.0	1438	682.1	6502	0.0	1308	608.8
total	140k	2.3	38k	20333.7	132k	3.9	40k	20258.2	140k	7.1	38k	23273.4	140k	3.0	42k	20297.0
timeouts				3/12				3/12				4/12				3/12
mean	10386.2	0.2	1753.3	1243.9	10019.6	0.3	1714.7	1245.4	10171.4	0.6	1692.7	1444.2	10431.5	0.2	1870.5	1282.9

Table D.44. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set CPMP200S)

instance	default				20 vars				10%				no limit			
	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
new1_2_32	3073	0.0	119	979.6	3073	0.0	119	975.8	3094	0.0	119	838.8	3766	0.0	119	973.6
new1_6_32	3690	0.0	17	468.5	3690	0.0	17	467.3	3693	0.0	21	424.3	4570	0.0	17	410.0
new1_10_32	6344	0.0	35	1085.1	6344	0.0	35	1079.5	6342	0.0	35	1025.9	7437	0.0	39	912.5
new2_1_32	2102	0.0	7	109.5	2102	0.0	7	109.2	2104	0.0	7	105.7	2370	0.0	7	81.5
new2_3_32	2365	0.0	23	299.5	2365	0.0	23	298.4	2375	0.0	23	261.8	2706	0.0	23	243.9
new2_7_32	3576	0.0	36	642.2	3576	0.0	36	640.8	3552	0.0	21	445.3	4149	0.0	97	1415.3
new3_1_32	3875	0.0	31	1205.1	3747	0.0	27	1163.5	3890	0.0	27	996.7	4151	0.0	35	1447.3
new3_4_32	5235	0.0	39	2596.2	5235	0.0	39	2586.8	5237	0.0	37	2487.1	5915	0.0	37	2229.7
new3_5_32	6261	0.0	33	1937.4	6257	0.0	33	1946.3	6262	0.0	33	1747.4	7315	0.0	29	1852.4
new4_1_32	833	0.0	1	26.1	833	0.0	1	26.0	833	0.0	1	26.1	952	0.0	1	26.1
new4_5_32	1502	0.0	13	86.4	1502	0.0	13	85.6	1502	0.0	13	74.7	1662	0.0	13	90.2
new4_10_32	2474	0.0	3	61.0	2474	0.0	3	60.5	2474	0.0	3	60.7	2696	0.0	5	92.9
new5_2_32	3182	0.0	11	248.6	3182	0.0	11	247.6	3179	0.0	11	225.1	3906	0.0	11	290.2
new5_7_32	5601	0.0	133	2008.2	5601	0.0	133	1998.6	5638	0.0	135	2027.1	6865	0.0	193	2916.1
new5_10_32	6982	0.0	85	2109.2	6982	0.0	85	2103.0	6989	0.0	74	1682.2	8466	0.0	37	1207.5
new6_1_32	3604	0.0	25	692.2	3604	0.0	25	689.6	3595	0.0	25	576.7	4046	0.0	21	630.1
new6_2_32	4983	0.0	7	518.3	4983	0.0	7	516.9	4991	0.0	7	482.1	5996	0.0	7	569.9
new6_5_32	6962	0.0	29	1546.7	6957	0.0	29	1541.1	6956	0.0	29	1405.6	8368	0.0	47	2377.4
new7_1_32	3300	0.0	58	614.3	3300	0.0	58	613.7	3312	0.0	55	519.4	4124	0.0	52	632.6
new7_4_32	5582	0.0	25	665.2	5582	0.0	25	663.3	5582	0.0	25	614.0	6702	0.0	62	1235.0
new7_9_32	7228	0.0	31	1051.1	7228	0.0	31	1050.4	7229	0.0	31	1048.2	8776	0.0	19	1004.3
total	88k	0.0	761	18950.4	88k	0.0	757	18863.9	88k	0.0	732	17074.9	104k	0.0	871	20638.5
timeouts				0/21				0/21				0/21				0/21
mean	3862.2	0.0	26.8	551.5	3855.8	0.0	26.6	548.9	3865.6	0.0	25.8	495.7	4520.7	0.0	28.9	590.0

Table D.45. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set RAP32S)

	default				20 vars				10%				no limit			
instance	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time	vars	gap	nodes	time
new1_2_64	2723	0.0	13	350.5	2723	0.0	13	350.4	2724	0.0	13	307.7	3137	0.0	15	332.6
new1_6_64	3197	0.0	39	1143.4	3171	0.0	33	1002.8	3200	0.0	38	913.0	3592	0.0	29	1027.6
new1_10_64	4750	0.0	7	896.1	4748	0.0	7	893.3	4743	0.0	7	892.3	5569	0.0	7	874.7
new2_1_64	1401	0.0	3	140.0	1401	0.0	3	139.9	1401	0.0	3	121.8	1628	0.0	3	149.0
new2_3_64	1804	0.0	2	135.9	1804	0.0	2	135.1	1804	0.0	2	136.1	2160	0.0	3	141.9
new2_7_64	2803	0.0	13	740.3	2803	0.0	13	738.6	2788	0.0	11	570.3	3493	0.0	11	669.4
new3_1_64	2521	0.0	1	1205.8	2521	0.0	1	1204.3	2521	0.0	1	1207.3	2798	0.0	1	1004.3
new3_4_64	3154	0.0	9	1688.5	3154	0.0	9	1686.2	3150	0.0	9	1590.8	3435	0.0	9	2089.1
new3_5_64	3600	0.0	9	3039.5	3595	0.0	9	3002.1	3606	0.0	7	2354.9	3971	0.0	15	3420.5
new4_1_64	565	0.0	1	15.6	565	0.0	1	15.6	565	0.0	1	15.7	685	0.0	1	19.0
new4_5_64	1166	0.0	13	106.3	1166	0.0	13	105.8	1166	0.0	13	96.0	1421	0.0	13	108.7
new4_10_64	2247	0.0	5	154.4	2247	0.0	5	153.9	2247	0.0	5	152.9	2505	0.0	3	129.6
new5_2_64	2143	0.0	1	230.7	2143	0.0	1	230.6	2143	0.0	1	230.5	2359	0.0	1	219.2
new5_7_64	3455	0.0	13	730.5	3455	0.0	13	728.9	3455	0.0	13	662.0	3966	0.0	13	934.7
new5_10_64	4843	0.0	25	1799.1	4804	0.0	25	1786.2	4826	0.0	25	1303.5	5643	0.0	37	2819.9
new6_1_64	1920	0.0	7	484.2	1920	0.0	7	483.1	1919	0.0	7	451.3	2017	0.0	9	590.1
new6_2_64	2846	0.0	3	686.3	2846	0.0	3	685.3	2846	0.0	3	620.0	3174	0.0	3	946.0
new6_5_64	3296	0.0	21	1826.8	3287	0.0	21	1797.5	3296	0.0	21	1656.3	3578	0.0	13	1896.4
new7_1_64	2117	0.0	7	276.9	2117	0.0	7	276.3	2117	0.0	7	270.9	2422	0.0	7	286.2
new7_4_64	4190	0.0	37	3600.0	4190	0.0	41	3600.0	4178	0.0	44	3600.0	4443	0.0	57	3600.0
new7_9_64	6002	0.0	11	1800.4	6002	0.0	11	1796.5	6020	0.0	11	1665.1	7025	0.0	9	1750.9
total	60k	0.0	240	21051.2	60k	0.0	238	20812.4	60k	0.0	242	18818.4	69k	0.0	259	23009.8
timeouts				1/21				1/21				1/21				1/21
mean	2679.6	0.0	9.4	559.3	2676.7	0.0	9.3	554.0	2678.4	0.0	9.3	509.3	3036.5	0.0	9.7	591.2

Table D.46. Impact of the reduced cost pricing setting at subsequent nodes on the performance of GCG (test set RAP64S)

instance	default			no pseudocosts			enforce in master			no proper variables			no early termination			convexification			all off			knapsack		
	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time
p550-1	0.0	32	11.9	0.0	33	15.7	0.0	23	10.1	0.0	25	11.6	0.0	29	13.1	0.0	29	15.9	0.0	33	15.6	0.0	26	3.9
p550-3	0.0	3	6.5	0.0	3	6.5	0.0	3	6.8	0.0	3	6.9	0.0	3	6.5	0.0	3	6.5	0.0	3	6.8	0.0	3	2.3
p550-5	0.0	1	7.2	0.0	1	7.1	0.0	1	7.1	0.0	1	7.2	0.0	1	7.1	0.0	1	7.2	0.0	1	7.2	0.0	1	2.3
p550-7	0.0	15	13.2	0.0	19	14.2	0.0	15	11.2	0.0	21	14.1	0.0	15	14.1	0.0	15	12.0	0.0	31	19.6	0.0	15	1.4
p550-9	0.0	9	6.1	0.0	5	5.2	0.0	5	5.9	0.0	5	5.6	0.0	7	6.0	0.0	5	5.8	0.0	5	5.6	0.0	5	1.2
p1250-1	0.0	173	18.8	0.0	81	8.6	0.0	113	12.3	0.0	108	11.6	0.0	117	14.4	0.0	115	13.6	0.0	82	9.0	0.0	138	1.9
p1250-3	0.0	155	18.7	0.0	155	16.0	0.0	125	15.8	0.0	139	15.7	0.0	150	20.1	0.0	164	19.6	0.0	185	21.2	0.0	93	1.6
p1250-5	0.0	68	11.7	0.0	535	60.5	0.0	150	19.1	0.0	85	14.8	0.0	68	12.5	0.0	68	11.8	0.0	482	62.0	0.0	80	1.8
p1250-7	0.0	1514	174.3	0.0	14286	1093.0	0.0	1616	192.0	0.0	1447	186.5	0.0	1559	210.3	0.0	2283	263.1	0.0	14071	1400.7	0.0	1261	8.3
p1250-9	0.0	138	36.9	0.0	442	82.8	0.0	162	33.4	0.0	129	31.6	0.0	144	37.9	0.0	116	23.9	0.0	433	89.1	0.0	141	1.9
p1650-1	0.0	8	2.6	0.0	14	2.9	0.0	8	2.5	0.0	8	2.6	0.0	8	2.7	0.0	8	2.6	0.0	14	3.1	0.0	5	0.9
p1650-3	0.0	13	2.8	0.0	68	6.8	0.0	13	2.8	0.0	13	2.8	0.0	13	2.9	0.0	13	2.8	0.0	90	8.7	0.0	11	1.0
p1650-5	0.0	59	8.5	0.0	144	14.6	0.0	60	8.3	0.0	136	15.9	0.0	66	10.1	0.0	75	10.2	0.0	148	16.6	0.0	54	1.1
p1650-7	0.0	75	9.8	0.0	109	11.6	0.0	81	10.4	0.0	79	9.9	0.0	75	11.5	0.0	75	9.8	0.0	109	14.1	0.0	82	1.2
p1650-9	0.0	198	26.7	0.0	354	32.8	0.0	228	31.3	0.0	174	24.4	0.0	186	29.6	0.0	198	26.8	0.0	455	60.6	0.0	249	1.9
p2050-1	0.0	38	4.5	0.0	45	5.8	0.0	38	4.5	0.0	38	4.5	0.0	38	4.6	0.0	42	4.7	0.0	47	5.8	0.0	40	1.0
p2050-3	0.0	11	3.5	0.0	11	3.5	0.0	11	3.4	0.0	11	3.5	0.0	11	3.8	0.0	11	3.5	0.0	11	3.7	0.0	11	1.0
p2050-5	0.0	5	2.9	0.0	5	2.9	0.0	5	3.0	0.0	5	3.0	0.0	5	3.0	0.0	5	2.9	0.0	5	3.2	0.0	6	0.8
p2050-7	0.0	5	3.0	0.0	3	2.7	0.0	5	3.0	0.0	5	3.1	0.0	5	3.1	0.0	5	2.9	0.0	3	2.8	0.0	7	0.9
p2050-9	0.0	343	46.3	0.0	5222	332.6	0.0	288	39.7	0.0	380	50.6	0.0	258	43.3	0.0	249	34.4	0.0	4924	488.4	0.0	303	2.1
total	0.0	2863.0	415.9	0.0	21k	1725.8	0.0	2950.0	422.6	0.0	2812.0	425.9	0.0	2758.0	456.6	0.0	3480.0	480.0	0.0	21k	2243.8	0.0	2531.0	38.5
timeouts			0/20			0/20			0/20			0/20			0/20			0/20			0/20			0/20
mean	0.0	44.9	12.8	0.0	82.5	20.7	0.0	44.6	12.5	0.0	45.2	12.9	0.0	42.9	13.3	0.0	43.8	12.6	0.0	86.6	23.9	0.0	42.1	1.8

Table D.47. Impact of selected advanced features and problem specific pricing solvers on the performance of GCG (test set CPMP50)

instance	default			no pseudocosts			enforce in master			no proper variables			no early termination			convexification			all off			knapsack		
	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time
p10100-11	0.0	35	62.6	0.0	88	115.2	0.0	39	66.5	0.0	25	64.6	0.0	25	59.5	0.0	52	95.5	0.0	96	117.3	0.0	20	37.3
p10100-13	0.0	7	19.5	0.0	11	30.1	0.0	5	19.6	0.0	5	29.1	0.0	7	19.8	0.0	14	37.9	0.0	12	39.7	0.0	23	35.2
p10100-15	0.0	235	244.3	0.0	220	296.9	0.0	192	193.2	0.0	187	214.6	0.0	232	252.5	0.0	228	250.0	0.0	300	383.5	0.0	189	55.7
p10100-17	0.0	134	144.8	0.0	331	196.7	0.0	79	89.2	0.0	126	109.7	0.0	155	139.0	0.0	84	85.0	0.0	365	259.4	0.0	69	40.9
p10100-19	0.0	19	32.7	0.0	41	44.3	0.0	36	45.5	0.0	29	45.3	0.0	17	33.2	0.0	32	56.4	0.0	59	69.5	0.0	25	31.2
p25100-11	0.0	5202	862.4	1.0	27299	3600.0	0.0	4516	766.3	0.0	4479	797.5	0.0	5614	1155.6	0.0	4080	741.7	3.4	17045	3600.0	0.0	3181	73.0
p25100-13	1.4	26451	3600.0	8.7	33898	3600.0	3.8	23613	3600.0	0.8	28804	3600.0	2.8	21395	3600.0	1.0	27666	3600.0	7.6	24157	3600.0	0.0	55676	981.9
p25100-15	0.0	778	201.9	0.0	2138	405.2	0.0	598	144.6	0.0	622	169.6	0.0	566	185.1	0.0	490	142.3	0.0	2067	514.2	0.0	493	23.7
p25100-17	0.0	47	28.3	0.0	141	47.5	0.0	52	27.9	0.0	56	31.7	0.0	53	30.7	0.0	34	19.7	0.0	129	45.2	0.0	48	12.4
p25100-19	0.0	8501	1623.1	1.3	25488	3600.0	0.0	10819	2099.1	0.0	11996	2343.6	0.0	8647	1880.1	0.0	7963	1592.1	3.4	18371	3600.0	0.0	9272	183.1
p33100-11	0.0	558	83.6	0.0	1140	145.8	0.0	530	79.8	0.0	765	100.9	0.0	435	82.7	0.0	596	90.0	0.0	2309	367.3	0.0	568	18.9
p33100-13	0.0	185	36.8	0.0	959	119.8	0.0	131	29.8	0.0	131	27.2	0.0	145	38.7	0.0	118	26.0	0.0	1031	180.0	0.0	120	6.5
p33100-15	0.0	250	56.2	0.0	8578	945.4	0.0	450	87.8	0.0	306	61.5	0.0	516	116.8	0.0	306	73.3	0.0	4114	679.4	0.0	300	14.6
p33100-17	0.0	218	47.6	0.0	231	47.0	0.0	428	70.2	0.0	482	81.5	0.0	188	48.8	0.0	284	55.4	0.0	704	127.6	0.0	149	11.0
p33100-19	0.0	4076	603.2	0.6	28604	3600.0	0.0	4323	654.8	0.0	6147	957.6	0.0	4109	851.5	0.0	3041	471.1	0.3	19800	3600.0	0.0	2743	49.7
p40100-11	0.0	7957	1114.1	0.0	19350	2245.3	0.0	4629	646.2	0.0	7129	1044.8	0.0	4486	824.8	0.0	5175	698.5	0.0	16204	2870.0	0.0	4691	65.8
p40100-13	0.0	623	82.3	0.0	3751	373.5	0.0	1050	111.3	0.0	302	42.4	0.0	593	103.3	0.0	1027	120.9	0.0	3720	518.5	0.0	824	16.5
p40100-15	0.0	828	153.2	0.0	3812	464.0	0.0	789	141.0	0.0	1143	179.6	0.0	705	170.6	0.0	869	162.9	0.0	4147	815.5	0.0	622	18.7
p40100-17	0.0	2218	422.1	3.6	26740	3600.0	0.0	2221	429.8	0.0	1416	282.3	0.0	1554	380.4	0.0	1633	313.3	3.5	17055	3600.0	0.0	671	17.1
p40100-19	0.0	3173	531.6	1.2	25681	3600.0	0.0	2891	511.1	0.0	1605	295.3	0.0	2694	612.1	0.0	3602	619.0	4.9	16913	3600.0	0.0	2243	36.8
total	1.4	61k	9950.3	16.4	208k	27076.7	3.8	57k	9813.7	0.8	65k	10478.8	2.8	52k	10585.2	1.0	57k	9251.0	23.1	148k	28587.1	0.0	81k	1730.0
timeouts			1/20			6/20			1/20			1/20			1/20			1/20			6/20			0/20
mean	0.1	587.1	184.7	0.8	1962.6	469.5	0.2	596.2	181.7	0.0	574.9	183.5	0.1	537.6	199.5	0.0	567.7	182.8	1.1	1900.2	583.3	0.0	491.6	37.8

Table D.48. Impact of selected advanced features and problem specific pricing solvers on the performance of GCG (test set CPMP100)

instance	default			no pseudocosts			enforce in master			no proper variables			no early termination			convexification			all off			knapsack		
	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time
p15150-21	0.0	84	162.6	0.0	186	278.7	0.0	103	187.3	0.0	119	159.4	0.0	89	154.6	0.0	78	171.9	0.0	91	172.5	0.0	74	140.9
p15150-23	0.0	3	43.0	0.0	7	46.2	0.0	3	42.6	0.0	3	43.3	0.0	3	42.9	0.0	3	42.0	0.0	7	45.1	0.0	11	182.9
p15150-25	0.0	116	125.3	0.0	231	196.6	0.0	85	144.2	0.0	167	242.3	0.0	162	184.6	0.0	155	174.7	0.0	233	249.1	0.0	119	130.9
p15150-27	0.0	1688	2251.4	1.3	2799	3600.0	0.0	2931	3325.0	0.3	2613	3600.0	0.1	2925	3600.0	0.0	2436	3288.2	1.6	2512	3600.0	0.0	2707	660.8
p15150-29	0.0	1	39.2	0.0	1	39.2	0.0	1	39.2	0.0	1	39.2	0.0	1	38.9	0.0	1	39.2	0.0	1	39.1	0.0	1	171.9
p37150-21	0.0	207	105.1	0.0	4078	908.6	0.0	159	87.0	0.0	234	118.6	0.0	129	87.9	0.0	155	95.1	0.0	3603	1239.5	0.0	130	62.8
p37150-23	3.0	8303	3600.0	5.5	11748	3600.0	3.1	8417	3600.0	2.7	8489	3600.0	4.5	7012	3600.0	1.8	8447	3600.0	11.4	8210	3600.0	0.0	76642	3173.1
p37150-25	0.0	55	41.5	0.0	869	152.7	0.0	40	36.8	0.0	49	39.1	0.0	97	63.1	0.0	55	41.3	0.0	305	117.4	0.0	105	67.0
p37150-27	0.9	7682	3600.0	–	10485	3600.0	1.5	6777	3600.0	4.4	6417	3600.0	2.0	6159	3600.0	1.2	7655	3600.0	7.4	7569	3600.0	0.0	44139	1673.5
p37150-29	0.0	650	215.9	0.0	2704	606.8	0.0	648	236.8	0.0	626	202.1	0.0	576	242.0	0.0	785	277.5	0.0	2921	917.0	0.0	582	79.9
p50150-21	1.9	12817	3600.0	6.4	15829	3600.0	2.9	12332	3600.0	12.1	13460	3600.0	3.2	10688	3600.0	2.9	12799	3600.0	8.5	11189	3600.0	0.0	40201	1456.1
p50150-23	0.0	3408	1154.2	1.8	14700	3600.0	0.0	3437	1097.8	0.0	2881	988.6	0.0	3799	1663.7	0.0	3408	1152.3	1.8	9721	3600.0	0.0	4322	182.0
p50150-25	0.0	56	37.5	0.0	63	40.3	0.0	81	44.5	0.0	79	43.2	0.0	45	40.4	0.0	56	37.8	0.0	69	48.6	0.0	42	49.5
p50150-27	0.0	3284	1314.7	1.3	11633	3600.0	0.0	2536	988.2	0.0	3427	1386.7	0.0	1592	790.7	0.0	3632	1345.3	1.7	7966	3600.0	0.0	3326	154.3
p50150-29	1.0	15713	3600.0	6.0	20546	3600.0	2.2	15385	3600.0	1.7	15801	3600.0	4.1	12337	3600.0	1.0	16039	3600.0	5.6	13573	3600.0	0.0	75091	2366.3
p60150-21	0.0	463	175.3	0.0	7172	1354.4	0.0	389	149.3	0.0	434	166.2	0.0	373	165.0	0.0	378	138.2	0.0	7107	2226.8	0.0	452	46.5
p60150-23	0.0	5148	1648.0	10.0	13466	3600.0	0.0	9157	3076.7	0.0	6365	2032.1	0.0	6469	2660.5	0.0	5547	1813.5	10.0	9637	3600.0	0.0	6551	242.5
p60150-25	0.0	18501	2985.0	1.0	21964	3600.0	0.0	11484	1880.5	0.3	23367	3600.0	0.6	13489	3600.0	0.5	22426	3600.0	1.0	13682	3600.0	0.0	1831	91.6
p60150-27	6.5	7646	3600.0	–	10614	3600.0	11.1	6814	3600.0	6.8	7173	3600.0	10.8	5986	3600.0	6.5	7632	3600.0	–	7090	3600.0	3.5	92880	3600.0
p60150-29	0.0	512	133.9	0.0	3117	591.4	0.0	484	135.6	0.0	470	127.0	0.0	517	166.0	0.0	512	134.2	0.0	4101	1030.7	0.0	270	47.6
total	13.3	86k	28432.6	33.3	152k	40214.9	20.8	81k	29471.5	28.3	92k	30787.8	25.3	72k	31500.3	13.9	92k	30351.2	49.0	109k	42085.8	3.5	349k	14580.1
timeouts			5/20			10/20			5/20			7/20			7/20			6/20			10/20			1/20
mean	0.7	847.6	493.5	1.8	2211.7	920.3	1.0	830.4	501.0	1.4	908.9	529.6	1.2	793.8	540.9	0.7	877.3	518.2	2.5	1725.7	988.4	0.2	1228.6	253.8

Table D.49. Impact of selected advanced features and problem specific pricing solvers on the performance of GCG (test set CPMP150)

instance	default			no pseudocosts			enforce in master			no proper variables			no early termination			convexification			all off			knapsack		
	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time	gap	nodes	time
p20200-32	0.0	889	2482.8	4.2	973	3600.0	0.0	944	2110.2	0.0	738	1884.9	0.0	863	2241.8	0.0	789	1971.8	–	902	3600.0	0.0	1168	772.4
p20200-34	0.0	508	1187.2	2.4	1800	3600.0	0.0	684	1367.3	0.0	388	1024.9	0.0	524	1427.0	0.0	492	1296.8	0.9	1505	3600.0	0.0	495	918.1
p20200-36	0.0	62	293.0	0.0	173	465.6	0.0	36	274.9	0.0	52	328.0	0.0	39	279.6	0.0	33	269.1	0.0	273	756.8	0.0	48	650.7
p50200-36	0.0	2488	1234.3	3.3	10835	3600.0	0.0	1927	1085.1	0.0	2943	1263.1	0.0	2014	1287.7	0.0	889	532.6	3.3	6576	3600.0	0.0	1741	282.7
p50200-37	0.0	2650	1067.9	3.6	11594	3600.0	0.0	3133	1237.4	0.0	2438	999.8	0.0	2284	1126.1	0.0	5811	1950.5	2.5	7897	3600.0	0.0	4203	478.4
p50200-39	0.0	1176	589.7	1.2	11374	3600.0	0.0	2301	972.2	0.0	1029	500.9	0.0	1122	669.1	0.0	1850	822.6	0.9	7386	3600.0	0.0	1324	284.0
p66200-31	0.0	1148	377.0	0.0	14290	2823.5	0.0	2120	563.1	0.0	1133	345.3	0.0	677	336.1	0.0	1524	451.4	0.2	9874	3600.0	0.0	1753	230.7
p66200-36	0.0	4533	1606.2	2.9	13653	3600.0	0.0	5149	1886.4	0.0	5243	1902.5	0.0	6404	3151.3	0.0	5171	1878.0	2.0	7947	3600.0	0.0	6510	500.8
p66200-38	0.6	8346	3600.0	3.2	12556	3600.0	1.8	8663	3600.0	2.0	8677	3600.0	1.5	6709	3600.0	4.4	9066	3600.0	2.7	7156	3600.0	0.0	22744	1403.8
p80200-33	0.8	10218	3600.0	5.4	11499	3600.0	1.4	10176	3600.0	0.9	10834	3600.0	0.8	7802	3600.0	1.3	10768	3600.0	5.4	7875	3600.0	0.0	15458	788.7
p80200-34	0.9	4554	3600.0	–	6542	3600.0	1.0	4530	3600.0	2.7	4614	3600.0	0.7	3602	3600.0	0.7	4861	3600.0	–	4116	3600.0	0.0	15096	891.9
p80200-38	0.0	1656	695.6	0.6	10717	3600.0	0.0	1464	652.7	0.0	1345	643.3	0.0	1246	839.1	0.0	1656	693.4	0.8	6747	3600.0	0.0	1343	181.3
total	2.3	38k	20333.7	26.8	106k	39289.1	4.2	41k	20949.3	5.6	39k	19692.7	3.0	33k	22157.8	6.4	42k	20666.2	18.7	68k	40356.8	0.0	71k	7383.5
timeouts			3/12			10/12			3/12			3/12			3/12			3/12			11/12			0/12
mean	0.2	1753.3	1243.9	2.4	5577.7	2978.0	0.3	1926.6	1343.6	0.5	1666.0	1192.0	0.2	1486.3	1352.7	0.5	1776.8	1265.0	1.7	4025.6	3162.8	0.0	2338.0	519.3

Table D.50. Impact of selected advanced features and problem specific pricing solvers on the performance of GCG (test set CPMP200)

	SCIP			GCG		
instance	gap	nodes	time	gap	nodes	time
1-FullIns_3	0.0	36	2.3	0.0	1	1.9
1-FullIns_4	0.0	177173	1366.2	–	220	3600.0
2-FullIns_3	0.0	2296	18.9	0.0	27	9.0
2-Insertions_3	0.0	39374	62.2	65.1	1384	3600.0
3-FullIns_3	0.0	42827	378.7	0.0	41	15.4
4-FullIns_3	16.7	265396	3600.0	0.0	74	20.3
4-FullIns_4	166.7	1109	3600.0	–	15	3600.0
5-FullIns_3	43.3	135921	3600.0	0.0	111	57.3
anna	0.0	79	14.9	0.0	52	8.2
david	0.0	1	0.4	0.0	26	3.3
DSJC125.9	–	4	3600.0	0.0	1	3573.1
fpsol2.i.1	–	1	3600.0	0.0	250	2589.1
games120	0.0	51	16.1	0.0	68	25.7
homer	30.0	42722	3600.0	0.0	458	1487.7
huck	0.0	242852	1589.1	0.0	25	2.0
jean	42.9	774653	3600.0	0.0	37	2.6
le450_25a	118.8	156	3600.0	0.0	405	3592.4
le450_25b	44.0	507	3600.0	0.0	439	3594.5
miles1000	16.2	1525	3600.0	0.0	16	41.4
miles1500	13.0	104	3600.0	0.0	1	77.1
miles250	14.3	562291	3600.0	0.0	92	22.6
miles500	5.3	80400	3600.0	0.0	55	15.3
miles750	37.5	5626	3600.0	0.0	18	22.1
mulsol.i.1	4.3	4793	3600.0	0.0	40	97.8
mulsol.i.2	3.3	2914	3600.0	0.0	93	354.6
mulsol.i.3	3.3	1236	3600.0	0.0	92	384.9
mulsol.i.4	6.7	532	3600.0	0.0	84	368.9
mulsol.i.5	10.0	202	3600.0	0.0	105	416.2
myciel3	0.0	174	0.5	0.0	7	0.3
myciel4	0.0	147961	255.8	0.0	617	183.0
queen6_6	16.7	368387	3600.0	0.0	5	16.7
queen7_7	0.0	763	38.4	0.0	1	8.7
queen8_8	12.5	58121	3600.0	0.0	105	692.1
queen9_9	22.2	18615	3600.0	22.2	541	3600.0
queen10_10	30.0	2365	3600.0	59.7	262	3600.0
will199GPIA	33.3	615	3600.0	–	1	3600.0
zeroin.i.1	8.9	1811	3600.0	0.0	66	237.8
zeroin.i.2	7.1	2138	3600.0	0.0	82	314.5
zeroin.i.3	7.1	3161	3600.0	0.0	100	311.4
total	914.1	2988k	100943.5	447.0	6017.0	40147.9
timeouts			27/39			6/39
mean	19.5	3256.9	1073.3	8.7	64.3	176.1

Table D.51: Comparison of SCIP and GCG on test set COLORING-ALL

	SCIP			GCG	
instance	gap	nodes	time	gap	nodes	time
p550-1	0.0	163	8.0	0.0	32	12.2
p550-2	0.0	1	1.0	0.0	1	3.8
p550-3	0.0	2	1.9	0.0	3	6.6
p550-4	0.0	2	1.1	0.0	1	6.2
p550-5	0.0	1	1.6	0.0	1	7.3
p550-6	0.0	1	1.7	0.0	1	4.0
p550-7	0.0	154	8.7	0.0	15	13.4
p550-8	0.0	9043	33.6	0.0	1561	599.0
p550-9	0.0	10	3.6	0.0	9	6.2
p550-10	0.0	3294	19.6	0.0	43	40.8
p1250-1	0.0	114	2.9	0.0	173	19.3
p1250-2	0.0	4	2.0	0.0	17	4.3
p1250-3	0.0	1193	8.4	0.0	155	19.2
p1250-4	0.0	168567	192.0	0.0	2406	275.4
p1250-5	0.0	1526	10.4	0.0	68	11.9
p1250-6	0.0	47464	100.5	0.0	472	57.1
p1250-7	0.0	170655	328.1	0.0	1514	177.7
p1250-8	0.0	20866	46.4	0.0	117	17.9
p1250-9	0.0	88223	119.6	0.0	138	37.2
p1250-10	0.0	219478	455.9	0.0	1175	179.9
p1650-1	0.0	160	2.5	0.0	8	2.6
p1650-2	0.0	4985	9.3	0.0	78	9.2
p1650-3	0.0	32	2.5	0.0	13	2.9
p1650-4	0.0	8313	12.8	0.0	53	8.3
p1650-5	0.0	1625	5.0	0.0	59	8.7
p1650-6	0.0	879	9.6	0.0	13	3.6
p1650-7	0.0	2132	14.4	0.0	75	10.1
p1650-8	0.0	293989	484.0	0.0	12679	1256.0
p1650-9	0.0	4616	20.6	0.0	198	27.2
p1650-10	0.0	64647	136.7	0.0	2367	294.9
p2050-1	0.0	5157	6.8	0.0	38	4.7
p2050-2	0.0	1138	3.8	0.0	86	9.9
p2050-3	0.0	598	9.3	0.0	11	3.5
p2050-4	0.0	57	2.8	0.0	20	3.2
p2050-5	0.0	1969	14.2	0.0	5	2.9
p2050-6	0.0	2	2.0	0.0	4	2.8
p2050-7	0.0	15742	17.7	0.0	5	3.0
p2050-8	0.0	209862	368.0	0.0	630	79.5
p2050-9	0.0	47674	113.6	0.0	343	47.3
p2050-10	4.8	1612769	3600.0	0.0	692	110.5
total	4.8	3007k	6182.6	0.0	25k	3390.2
timeouts			1/40			0/40
mean	0.1	1652.9	25.8	0.0	80.9	23.9

Table D.52: Comparison of SCIP and GCG on test set CPMP50

	SCIP			GCG		
instance	gap	nodes	time	gap	nodes	time
p10100-11	0.0	3746	36.9	0.0	35	63.1
p10100-12	0.0	2156	73.4	0.0	45	78.0
p10100-13	0.0	840	40.6	0.0	7	20.1
p10100-14	0.0	12118	194.0	0.0	279	261.6
p10100-15	0.0	11975	164.6	0.0	235	245.6
p10100-16	0.0	478	30.3	0.0	19	45.7
p10100-17	0.0	3163	77.8	0.0	134	147.1
p10100-18	0.0	11269	173.1	0.0	325	259.1
p10100-19	0.0	2102	59.4	0.0	19	33.4
p10100-20	1.4	305944	3600.0	0.8	4165	3600.0
p25100-11	0.0	161357	1492.2	0.0	5202	881.0
p25100-12	0.0	5428	90.9	0.0	2060	315.5
p25100-13	1.3	428360	3600.0	1.4	25947	3600.0
p25100-14	0.0	69642	701.5	0.0	3252	660.1
p25100-15	0.0	28588	301.5	0.0	778	205.9
p25100-16	2.1	345276	3600.0	0.0	11592	2180.9
p25100-17	0.0	1765	64.5	0.0	47	29.2
p25100-18	0.0	12876	247.6	0.0	149	72.7
p25100-19	1.9	408594	3600.0	0.0	8501	1644.4
p25100-20	16.1	130000	3600.0	17.2	9342	3600.0
p33100-11	0.0	6679	97.7	0.0	558	86.1
p33100-12	1.1	461239	3600.0	0.5	30731	3600.0
p33100-13	0.0	602	33.9	0.0	185	37.6
p33100-14	1.2	438288	3600.0	0.0	4447	643.6
p33100-15	0.0	3056	72.4	0.0	250	56.6
p33100-16	0.0	307493	2689.4	0.0	12470	2079.5
p33100-17	0.0	29666	341.5	0.0	218	48.4
p33100-18	0.0	68763	754.7	1.3	23614	3600.0
p33100-19	0.0	43911	421.4	0.0	4076	619.2
p33100-20	1.2	354820	3600.0	0.0	959	264.3
p40100-11	0.0	24484	255.0	0.0	7957	1140.7
p40100-12	1.3	435786	3600.0	0.0	16184	2035.2
p40100-13	0.0	2038	54.8	0.0	623	83.6
p40100-14	0.0	30323	352.9	0.0	329	82.0
p40100-15	0.0	14780	183.4	0.0	828	156.2
p40100-16	0.0	6621	121.1	0.0	283	57.6
p40100-17	1.1	429574	3600.0	0.0	2218	426.8
p40100-18	0.0	291995	2584.2	0.0	1732	351.4
p40100-19	0.0	61675	726.2	0.0	3173	540.5
p40100-20	3.7	386116	3600.0	0.0	297	90.8
total	32.4	5343k	52036.9	21.2	183k	33943.5
timeouts			11/40			5/40
mean	0.8	28557.8	436.9	0.5	929.5	298.0

Table D.53: Comparison of SCIP and GCG on test set CPMP100

	SCIP			GCG		
instance	gap	nodes	time	gap	nodes	time
p15150-21	0.0	3313	165.4	0.0	84	163.0
p15150-22	0.7	103476	3600.0	0.0	363	503.8
p15150-23	0.0	1155	122.7	0.0	3	43.2
p15150-24	0.0	52924	687.3	0.0	10	68.9
p15150-25	0.0	12174	401.1	0.0	116	125.8
p15150-26	0.0	3358	175.2	0.0	47	130.3
p15150-27	0.0	63122	1993.3	0.0	1688	2278.4
p15150-28	0.0	434	22.3	0.0	11	49.9
p15150-29	0.0	11	12.1	0.0	1	39.5
p15150-30	0.0	984	90.6	0.0	3	39.3
p37150-21	0.0	17054	587.8	0.0	207	107.2
p37150-22	2.0	148746	3600.0	0.0	6685	2261.6
p37150-23	14.4	65070	3600.0	3.0	8220	3600.0
p37150-24	0.0	89401	2242.1	0.0	2332	584.6
p37150-25	0.0	2364	160.9	0.0	55	42.0
p37150-26	0.0	10991	326.5	0.0	156	79.4
p37150-27	12.7	57700	3600.0	0.9	7613	3600.0
p37150-28	0.0	33572	767.7	0.0	7164	1600.0
p37150-29	0.0	15015	392.0	0.0	650	218.4
p37150-30	0.6	167437	3600.0	0.0	6819	1621.8
p50150-21	1.4	156359	3600.0	1.9	12666	3600.0
p50150-22	10.0	68778	3600.0	9.0	11479	3600.0
p50150-23	3.9	98421	3600.0	0.0	3408	1165.9
p50150-24	1.5	128420	3600.0	0.0	3487	886.8
p50150-25	0.0	32871	91.5	0.0	56	38.6
p50150-26	0.0	11194	379.3	0.0	4071	927.0
p50150-27	1.1	129987	3600.0	0.0	3284	1325.9
p50150-28	0.0	2604	138.8	0.0	435	115.4
p50150-29	1.4	148682	3600.0	1.0	15627	3600.0
p50150-30	2.1	151174	3600.0	0.5	16284	3600.0
p60150-21	1.1	153650	3600.0	0.0	463	177.3
p60150-22	6.5	84866	3600.0	0.0	5143	1816.3
p60150-23	14.7	40000	3600.0	0.0	5148	1670.7
p60150-24	2.6	121739	3600.0	0.4	15024	3600.0
p60150-25	0.0	987	88.0	0.0	18501	3048.5
p60150-26	4.4	551447	3600.0	0.0	1637	337.3
p60150-27	25.7	52747	3600.0	6.5	7593	3600.0
p60150-28	1.0	177083	3600.0	0.0	6364	1160.3
p60150-29	0.0	874	78.1	0.0	512	135.8
p60150-30	0.0	7525	231.8	0.0	1007	221.4
total	107.8	2967k	77554.5	23.2	174k	51784.3
timeouts			19/40			8/40
mean	2.6	22250.3	846.8	0.6	979.1	513.9

Table D.54: Comparison of SCIP and GCG on test set CPMP150

	SCIP			GCG		
instance	gap	nodes	time	gap	nodes	time
p20200-31	0.0	20534	1554.8	0.0	181	369.4
p20200-32	2.0	53487	3600.0	0.0	889	2493.0
p20200-33	0.0	36409	2330.0	0.0	203	504.7
p20200-34	0.5	67579	3600.0	0.0	508	1191.7
p20200-35	0.0	16443	1283.1	0.0	302	563.9
p20200-36	0.0	2457	313.1	0.0	62	294.0
p20200-37	0.0	2843	297.9	0.0	55	256.4
p20200-38	0.4	69953	3600.0	0.0	344	697.3
p20200-39	0.0	1511	287.0	0.0	25	128.4
p20200-40	0.0	1355	345.7	0.0	82	222.5
p50200-31	2.6	60667	3600.0	2.1	8282	3600.0
p50200-32	41.6	33990	3600.0	–	2737	3600.0
p50200-33	6.1	33156	3600.0	12.1	6335	3600.0
p50200-34	35.6	37520	3600.0	–	3305	3600.0
p50200-35	4.1	41552	3600.0	7.2	7008	3600.0
p50200-36	0.0	27590	1812.3	0.0	2488	1241.7
p50200-37	0.0	24845	1549.2	0.0	2650	1093.8
p50200-38	17.5	33824	3600.0	–	5253	3600.0
p50200-39	0.0	37494	1907.9	0.0	1176	592.1
p50200-40	13.5	34586	3600.0	12.6	4880	3600.0
p66200-31	0.0	1548	219.4	0.0	1148	380.2
p66200-32	72.3	38044	3600.0	–	3660	3600.0
p66200-33	12.6	33080	3600.0	–	6788	3600.0
p66200-34	21.1	33302	3600.0	7.0	5297	3600.0
p66200-35	6.9	41225	3600.0	5.5	8129	3600.0
p66200-36	2.6	64980	3600.0	0.0	4533	1615.8
p66200-37	3.2	54983	3600.0	10.3	10271	3600.0
p66200-38	2.4	62625	3600.0	0.6	8250	3600.0
p66200-39	2.1	66666	3600.0	2.2	10485	3600.0
p66200-40	5.4	44502	3600.0	9.4	8142	3600.0
p80200-31	1.5	83648	3600.0	3.8	11408	3600.0
p80200-32	56.2	26806	3600.0	3.9	3669	3600.0
p80200-33	2.8	63911	3600.0	0.8	10146	3600.0
p80200-34	21.6	22430	3600.0	0.9	4555	3600.0
p80200-35	0.0	68676	3283.8	0.0	7078	2585.3
p80200-36	3.6	52193	3600.0	1.1	8944	3600.0
p80200-37	1.6	70721	3600.0	1.3	10038	3600.0
p80200-38	3.9	51824	3600.0	0.0	1656	698.1
p80200-39	1.7	79323	3600.0	1.1	10841	3600.0
p80200-40	13.7	19330	3600.0	5.9	7104	3600.0
total	359.1	1617k	115984.2	587.8	188k	97728.3
timeouts			28/40			23/40
mean	8.0	28550.0	2350.7	11.4	2248.2	1728.1

Table D.55: Comparison of SCIP and GCG on test set CPMP200

instance	SCIP gap	SCIP nodes	SCIP time	GCG (32) gap	GCG (32) nodes	GCG (32) time	GCG (64) gap	GCG (64) nodes	GCG (64) time
new1_1	1.3	757186	3600.0	0.0	29	201.2	0.0	3	111.4
new1_2	1.7	589833	3600.0	0.0	119	979.6	0.0	13	350.5
new1_3	1.6	408635	3600.0	0.0	23	359.7	0.0	9	529.5
new1_4	1.7	296254	3600.0	0.0	187	1888.9	0.0	7	473.3
new1_5	2.0	262849	3600.0	0.0	41	828.6	0.0	11	562.3
new1_6	1.4	194120	3600.0	0.0	17	468.5	0.0	39	1143.4
new1_7	1.5	163145	3600.0	0.0	27	460.6	0.0	1	478.7
new1_8	1.7	161800	3600.0	0.0	13	583.6	0.0	3	350.6
new1_9	1.8	108307	3600.0	0.0	87	1687.5	0.0	2	453.4
new1_10	2.2	84190	3600.0	0.0	35	1085.1	0.0	7	896.1
new2_1	2.0	750775	3600.0	0.0	7	109.5	0.0	3	140.0
new2_2	1.4	580746	3600.0	0.0	136	898.0	0.0	44	744.7
new2_3	1.1	415357	3600.0	0.0	23	299.5	0.0	2	135.9
new2_4	1.2	428950	3600.0	0.0	1	118.9	0.0	1	367.1
new2_5	1.9	207493	3600.0	0.0	161	1622.6	0.1	154	3600.0
new2_6	2.0	198288	3600.0	0.0	11	425.6	0.0	4	326.9
new2_7	1.6	188555	3600.0	0.0	36	642.2	0.0	13	740.3
new2_8	1.7	142970	3600.0	0.0	13	399.6	0.0	14	782.2
new2_9	1.8	75900	3600.0	0.0	31	702.4	0.0	1	293.5
new2_10	1.9	131284	3600.0	0.0	151	1489.0	0.0	41	1274.4
new3_1	1.3	233708	3600.0	0.0	31	1205.1	0.0	1	1205.8
new3_2	1.2	89037	3600.0	0.0	119	3600.0	0.0	49	3600.0
new3_3	1.3	75770	3600.0	0.0	13	1227.5	0.0	7	3599.9
new3_4	1.4	84830	3600.0	0.0	39	2596.2	0.0	9	1688.5
new3_5	1.7	19150	3600.0	0.0	33	1937.4	0.0	9	3039.5

continue next page

instance	SCIP gap	SCIP nodes	SCIP time	GCG (32) gap	GCG (32) nodes	GCG (32) time	GCG (64) gap	GCG (64) nodes	GCG (64) time
new3_6	2.2	8632	3600.0	0.1	55	3600.0	0.0	39	3600.0
new3_7	1.8	35455	3600.0	0.0	88	3600.0	0.0	62	3600.0
new3_8	1.8	30130	3600.0	0.0	53	3535.1	0.0	5	3172.4
new3_9	2.4	25500	3600.0	0.1	40	3600.0	73.8	1	3600.0
new3_10	2.6	864	3600.0	0.0	45	3600.0	0.2	27	3600.0
new4_1	0.0	4907	16.1	0.0	1	26.1	0.0	1	15.6
new4_2	0.0	187	4.8	0.0	1	17.9	0.0	1	17.1
new4_3	0.0	132	6.0	0.0	1	24.6	0.0	1	28.4
new4_4	0.1	1571331	3600.0	0.0	1	73.0	0.0	1	44.5
new4_5	0.0	70140	342.6	0.0	13	86.4	0.0	13	106.3
new4_6	0.2	792713	3600.0	0.0	1	58.8	0.0	1	66.1
new4_7	0.0	670545	3406.9	0.0	1	57.0	0.0	1	54.1
new4_8	0.2	535592	3600.0	0.0	4	99.3	0.0	3	59.0
new4_9	0.2	546140	3600.0	0.0	3	92.6	0.0	3	93.6
new4_10	0.2	507737	3600.0	0.0	3	61.0	0.0	5	154.4
new5_1	0.7	640409	3600.0	0.0	23	254.4	0.0	1	130.1
new5_2	1.3	317663	3600.0	0.0	11	248.6	0.0	1	230.7
new5_3	0.9	318430	3600.0	0.0	3	177.1	0.0	3	282.4
new5_4	1.4	247945	3600.0	0.0	75	1445.6	0.0	3	265.3
new5_5	1.4	120474	3600.0	0.0	42	871.3	0.0	17	887.4
new5_6	1.3	147990	3600.0	0.0	161	3570.1	0.0	129	3600.0
new5_7	1.0	135050	3600.0	0.0	133	2008.2	0.0	13	730.5
new5_8	1.3	114213	3600.0	0.0	15	554.7	0.0	1	564.4
new5_9	1.2	78548	3600.0	0.0	124	3600.0	0.0	87	3600.0
new5_10	1.7	22990	3600.0	0.0	85	2109.2	0.0	25	1799.1

continue next page

instance	SCIP gap	SCIP nodes	SCIP time	GCG (32) gap	GCG (32) nodes	GCG (32) time	GCG (64) gap	GCG (64) nodes	GCG (64) time
new6_1	1.0	215663	3600.0	0.0	25	692.2	0.0	7	484.2
new6_2	1.2	159970	3600.0	0.0	7	518.3	0.0	3	686.3
new6_3	1.0	92896	3600.0	0.0	27	975.0	0.0	11	559.0
new6_4	1.0	93850	3600.0	0.0	17	1091.2	0.0	1	628.5
new6_5	1.1	69570	3600.0	0.0	29	1546.7	0.0	21	1826.8
new6_6	1.8	14540	3600.0	0.0	13	1094.0	0.0	7	1714.0
new6_7	2.3	10384	3600.0	0.0	51	3600.0	0.0	3	1368.9
new6_8	1.6	6209	3600.0	0.1	77	3600.0	0.0	38	3600.0
new6_9	1.7	38634	3600.0	0.0	37	3600.0	0.0	3	2030.4
new6_10	3.0	772	3600.0	0.0	55	3600.0	0.0	35	3600.0
new7_1	1.4	537230	3600.0	0.0	58	614.3	0.0	7	276.9
new7_2	1.5	373513	3600.0	0.0	31	624.5	0.0	41	1263.9
new7_3	1.4	230756	3600.0	0.0	46	823.1	0.0	11	545.2
new7_4	1.8	164797	3600.0	0.0	25	665.2	0.0	37	3600.0
new7_5	1.3	147376	3600.0	0.0	9	486.2	0.0	9	791.8
new7_6	1.7	158523	3600.0	0.0	186	3600.0	0.0	34	3032.1
new7_7	1.7	95711	3600.0	0.0	140	3600.0	0.1	74	3600.0
new7_8	2.2	96525	3600.0	0.1	148	3600.0	0.0	19	3205.4
new7_9	1.7	80942	3600.0	0.0	31	1051.1	0.0	11	1800.4
new7_10	2.4	78398	3600.0	0.0	107	3600.0	0.1	42	3600.0
total	97.1	16259k	237776.4	0.4	3484.0	98169.6	74.3	1305.0	95403.1
timeouts			65/70			14/70			14/70
mean	1.4	97564.0	2772.4	0.0	32.2	727.4	0.8	11.5	670.7

Table D.56. Comparison of SCIP and GCG on test set RAP

List of Figures

List of Tables

Bibliography

[1] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[2] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] T. Achterberg, T. Berthold, S. Heinz, T. Koch, A. Martin, M. Pfetsch, S. Vigerske, and K. Wolter. SCIP (Solving Constraint Integer Programs), documentation. `http://scip.zib.de/doc`.

[4] T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint Integer Programming: A new approach to integrate CP and MIP. *Lecture Notes in Computer Science*, 5015(2):6–20, 2008.

[5] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006. `http://miplib.zib.de`.

[6] T. Achterberg and C. Raack. The MCF-separator – detecting and exploiting multi-commodity flows in MIPs. ZIB-Report 09-38, submitted to Mathematical Programming C, Zuse Institute Berlin, 2009.

[7] C. Alves and J. M. V. de Carvalho. A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research*, 35(4):1315–1328, 2008.

[8] L. H. Applegren. A column generation algorithm for a ship scheduling problem. *Transportation Science*, 3:71–78, 1969.

[9] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[10] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1996.

[11] T. Berthold. Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin, 2006.

[12] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. Mixed-integer programming: A progress report. In M. Grötschel, editor, *The Sharpest Cut: The Impact of Manfred Padberg and His Work*. SIAM, 2004.

[13] A. Caprara, F. Furini, and E. Malaguti. Exact algorithms for the temporal knapsack problem. Technical report OR-10-7, DEIS, University of Bologna, 2010.

[14] A. Ceselli. Capacitated p-median instances and corresponding format, used in [15] and [17]. `http://www.dti.unimi.it/~ceselli/datasets.html`.

[15] A. Ceselli. Two exact algorithms for the capacitated p-median problem. *4OR*, 1(4):319–340, 2003.

[16] A. Ceselli, M. Gatto, M. E. Lübbecke, M. Nunkesser, and H. Schilling. Optimizing the cargo express service of swiss federal railways. *Transportation Science*, 42(4):450–465, 2006.

[17] A. Ceselli and G. Righini. A branch and price algorithm for the capacitated p-median problem. *Networks*, 45(3):125–142, 2005.

[18] V. Chvátal. *Linear programming*. Freeman, 1983.

[19] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing – an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–106. Spinger, New York, 1984.

[20] COIN-OR, 2009. `http://www.coin-or.org`

[21] G. B. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[22] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[23] G. B. Dantzig and P. Wolfe. The decomposition algorithm for linear programs. *Econometrica*, 29(4):767–778, 1961.

[24] Z. Degraeve, W. Gochet, and R. Jans. Improving discrete model representations via symmetry considerations. *Management Science*, 47(10):1396–1407, 2001.

[25] Z. Degraeve, W. Gochet, and R. Jans. Alternative formulations for a layout problem in the fashion industry. *European Journal of Operational Research*, 143(1):80–93, 2002.

[26] G. Desaulniers, J. Desrosiers, and M. M. Solomon. Acceleration strategies in column generation methods for vehicle routing and crew scheduling problems. In C. C. Ribeiro and P. Hansen, editors, *Essays abd Surveys in Metaheuristics*, pages 309–324. Kluwer, Boston, 2001.

[27] G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors. *Column Generation*. GERAD 25th Anniversary Series. Springer, New York, 2005.

[28] J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. Time constrained routing and scheduling. In *Handbooks in Operations Research and Management Science: Network Routing*, pages 35–139. Elsevier Science, 1995.

[29] J. Desrosiers and M. E. Lübbecke. A primer in column generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 1–32. Springer, New York, 2005.

[30] M. Fischetti, A. Lodi, and D. Salvagnin. Just MIP it! In V. Maniezzo, T. Stützle, and S. Voš, editors, *Matheuristics*, pages 39–70. Springer US, 2009.

[31] J. Forrest and R. Lougee-Heimer. COIN branch and cut, user guide, 2005. `http://www.coin-or.org/Cbc`.

[32] J. J. Forrest and D. Goldfarb. Steepest edge simplex algorithms for linear programming. *Mathematical Programming*, 57(1–3):251–292, 1992.

[33] R. Fukasawa, H. Longo, J. Lysgaard, M. P. de Aragão, M. Reis, E. Uchoa, and R. F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106(3):491–511, 2006.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[35] A. M. Geoffrion. Lagrangean relaxation for integer programming. *Mathematical Programming Study*, 2:82–114, 1974.

[36] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 9(6):849–859, 1961.

[37] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem – part II. *Operations Research*, 11(6):863–888, 1963.

[38] GNU. the GNU linear programming kit. `http://www.gnu.org/software/glpk`.

[39] M. Grötschel. Lineare Optimierung, 2004. Skriptum zur Vorlesung im WS 2003/04.

[40] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, second corrected edition, 1993.

[41] O. Günlük, L. Ladányi, and S. de Vries. A branch-and-price algorithm and new test problems for spectrum auctions. *Management Science*, 51(3):391–406, 2005.

[42] Gurobi Optimizer 2.0. Reference Manual, 2009. http://www.gurobi.com/html/doc/refman/.

[43] IBM ILOG CPLEX 12.10. Reference Manual, 2009. http://www-01.ibm.com/software/integration/optimization/cplex/.

[44] R. Jans. Solving lot-sizing problems on parallel identical machines using symmetry-breaking constraints. *INFORMS J. on Computing*, 21(1):123–136, 2009.

[45] M. Jepsen, B. Petersen, S. Spoorendonk, and D. Pisinger. Subset-row inequlities applied to the vehicle routing problem with time windows. *Operations Research*, 56(2):497–511, 2008.

[46] D. Johnson, A. Mehrotra, and M. Trick. Color02/03/04: Graph coloring and its generalizations. http://mat.gsia.cmu.edu/COLOR04/.

[47] E. Johnson. Modelling and strong linear programs for mixed integer programming. In S. W. Wallace, editor, *Algorithms and model formulations in mathematical programming*, pages 1–43. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[48] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Softw. Pract. Exper.*, 30(11):1325–1352, 2000.

[49] V. Kaibel, M. Peinhardt, and M. E. Pfetsch. Orbitopal fixing. *Lecture Notes in Computer Science*, 4513:74–88, 2007.

[50] V. Kaibel and M. E. Pfetsch. Packing and partitioning orbitopes. *Mathematical Programming*, 114(1):1–36, 2008.

[51] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[52] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[53] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[54] A. Klar. Cutting planes in mixed integer programming. Master's thesis, Technische Universität Berlin, 2006.

[55] T. Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004. ZIB-Report 04-58.

[56] N. Kohl, J. Desrosiers, O. B. G. Madsen, M. M. Solomon, and F. Soumis. 2-path cuts for the vehicle routing problem with time windows. *Transportation Science*, 33(1):101–116, 1999.

[57] M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.

[58] E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[59] H. Marchand, A. Martin, R. Weismantel, and L. A. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123:397–446, 2002.

[60] O. Marcotte. The cutting stock problem and integer rounding. *Mathematical Programming*, 33:82–92, 1985.

[61] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94(1):71–90, 2002.

[62] S. Martello and P. Toth. *Knapsack Problems: Algorithms and computer Implementations*. Wiley-Interscience Series in Discrete Mathematics and Optimization, New York, 1990.

[63] A. Mehrotra and M. A. Trick. A column generation approach for graph coloring. *INFORMS JOURNAL ON COMPUTING*, 8(4):344–354, 1996.

[64] O. D. Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized column generation. *Discrete Math*, 194:229–237, 1997.

[65] R. R. Meyer. On the existence of optimal solutions to integer and mixed-integer programming problems. *Mathematical Programming*, 7(1):223–235, 1974.

[66] M. Minoux. A class of combinatorial optimization problems with polynomially solvable large scale set-covering/partitioning relaxations. *RAIRO*, 21:105–136, 1987.

[67] H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software, 2009. `http://plato.asu.edu/bench.html`

[68] G. L. Nemhauser, M. W. P. Savelsbergh, and G. S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Oper. Res. Lett.*, 15:47–58, 1994.

[69] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[70] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smirglio. Orbital branching. *Lecture Notes in Computer Science*, 4513:104–118, 2007.

[71] J. Ostrowski, J. Linderoth, F. Rossi, and S. Smirglio. Constraint orbital branching. *Lecture Notes in Computer Science*, 5035:225–239, 2008.

[72] P. M. Pardalos, T. Mavridou, and J. Xue. The graph coloring problem: A bibiographic survey. In D.-Z. Due and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 2, pages 331–395. Kluwer Academic Publishers, 1998.

[73] M. Parker and J. Ryan. A column generation algorithm for bandwidth packing. *Telecommunication Systems*, 2(1):185–195, 1993.

[74] B. Petersen, D. Pisinger, and S. Spoorendonk. Chvátal-gomory rank-1 cuts used in a dantzig-wolfe decomposition of the vehicle routing problem with time windows. In *The Vehicle Routing Problem: Latest Advances and New Challenges*, pages 397–420. Springer, 2008.

[75] J. Puchinger, P. J. Stuckey, M. G. Wallace, and S. Brand. Dantzig-Wolfe decomposition and branch-and-price solving in G12. *Constraints*, 2010. To appear.

[76] T. K. Ralphs and M. V. Galati. Decomposition in integer linear programming. In J. K. Karlof, editor, *Integer Programming: Theory and Practice*, pages 57–110. CRC Press, 2005.

[77] T. K. Ralphs and M. V. Galati. Decomposition and dynamic cut generation in integer linear programming. *Mathematical Programming*, 106(2):261–285, 2006.

[78] T. K. Ralphs and M. V. Galati. DIP – decomposition for integer programming. `https://projects.coin-or.org/Dip`, 2009.

[79] T. K. Ralphs and M. Güzelsoy. SYMPHONY version 5.2.0 user's manual, 2009. `http://www.coin-or.org/SYMPHONY`.

[80] T. K. Ralphs and L. Ladányi. *COIN/BCP User's Manual*, 2001. `http://www.coin-or.org/Presentations/bcp-man.pdf`.

[81] J. Renegar. A polynomial-time algorithm, based on newton's method, for linear programming. *Mathematical Programming*, 40(1–3):59–93, 1988.

[82] D. M. Ryan and B. A.Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, pages 269–280. North Holland, Amsterdam, 1981.

[83] M. W. P. Savelsbergh. A branch and price algorithm for the generalized assignment problem. *Operations Research*, 45(6):831–841, 1997.

[84] A. Scholl and R. Klein. Bin packing instances: Data set 1. `http://www.wiwi.uni-jena.de/Entscheidung/binpp/`.

[85] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[86] A. Schrijver. *Combinatorial optimization : polyhedra and efficiency*. Springer, 2003.

[87] S. Spoorendonk. *Cut and column generation*. PhD thesis, Technical University of Denmark, 2008.

[88] P. Toth and D. Vigo. *The Vehicle Routing Problem*. Monograph on Discrete Mathematics and Applications. SIAM, 2002.

[89] P. H. Vance. Branch-and-price algorithms for the one-dimensional cutting stock problem. *Computational Optimization and Applications*, 9(3):211–228, 1998.

[90] P. H. Vance, C. Barnhart, E. L. Johnson, and G. L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3(2):111–130, 1994.

[91] F. Vanderbeck. *Decomposition and column generation for integer programs*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1994.

[92] F. Vanderbeck. On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, 48(1):111–128, 2000.

[93] F. Vanderbeck. BaPCod – a generic branch-and-price code. `https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod`, 2005.

[94] F. Vanderbeck. Implementig mixed integer column generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 331–358. Springer, Berlin, 2005.

[95] F. Vanderbeck. Branching in branch-and-price: a generic scheme. Technical report, Institut de Mathématiques de Bordeaux, 2009.

[96] F. Vanderbeck and M. W. P. Savelsbergh. A generic view of dantzig-wolfe decomposition in mixed integer programming. *Operations Research Letters*, 34(3):296–306, 2006.

[97] F. Vanderbeck and L. A. Wolsey. An exact algorithm for IP column generation. *Operations Research Letters*, 19(4):151–159, 1996.

[98] K. Wolter. Implementation of Cutting Plane Separators for Mixed Integer Programs. Master's thesis, Technische Universität Berlin, 2006.

Die selbständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides statt.

Berlin, März 2010

Gerald Gamrath