Component Bound Branching in a Branch-and-Price Framework

Master Thesis in Computer Science
RWTH Aachen University

Til Mohr

til.mohr@rwth-aachen.de
Student ID: 405959

September 7, 2024

1st Examiner
Prof. Dr. Peter Rossmanith
Chair of Theoretical Computer Science
RWTH Aachen University

2nd Examiner
Prof. Dr. Marco Lübbecke
Chair of Operations Research
RWTH Aachen University

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Acknowledgements

I would like to express my sincere gratitude to my Professor and Supervisor, Prof. Lübbecke, for his guidance and support throughout this thesis. His expertise and valuable insights have been instrumental in shaping this work.

I would also like to thank Erik Mühmer for his assistance and support during the implementation phase in `GCG`. His technical expertise has been invaluable.

Furthermore, I would like to extend my thanks to Oliver Gaul for the valuable discussions regarding some theoretical ideas. His input has greatly contributed to the development of this thesis.

Lastly, I would like to acknowledge Marc Ludevid Wulf for providing the inspiration to use unrolled linked lists to further improve the memory layout of the variable synchronization.

I am grateful to all the individuals mentioned above for their contributions and support, which have been crucial in the successful completion of this thesis.

Abstract

This master thesis integrates the component bound branching rule, originally proposed by Vanderbeck et al. [1, 2] and later reformulated by Desrosiers et al. [3], into the branch-price-and-cut solver GCG. This rule, similarly to Vanderbeck's generic branching scheme [4], exclusively operates within the Dantzig-Wolfe reformulated problem, where branching decisions generally have no corresponding actions in the original formulation. The current GCG framework requires modifications for such branching rules, especially within the pricing loop, as seen in Vanderbeck's method implementation. These rules also fail to utilize enhancements like dual value stabilization.

A significant contribution of this thesis is the enhancement of the GCG architecture to facilitate the seamless integration of new branching rules that operate solely on the reformulated problem. This allows these rules to benefit from current and future advancements in the branch-price-and-cut framework, including dual value stabilization, without necessitating alterations to the implementation of the branching rule itself.

The thesis proposes an interface to manage constraints in the master problem that lack counterparts in the original formulation. These constraints require specific modifications to the pricing problems to ensure their validity in the master. The **generic mastercut** interface, tightly integrated into the GCG solver, spans the pricing loop, column generation, and dual value stabilization. Enhancements to the existing branching rule interface complement this integration, enabling effective utilization of the generic mastercuts.

Finally, the component bound branching rule will be implemented using this new interface and evaluated on a set of benchmark instances. Its performance will be benchmarked against the existing Vanderbeck branching rule, offering a practical comparison of both approaches.

Contents

Chapter 1

Introduction

The development of efficient algorithms for solving large-scale mixed-integer programming (*MIP*) problems has been a central focus of operations research for decades. Column generation, a powerful technique for solving large-scale linear programs, has been extended to integer programs through the branch-and-price algorithm. The effectiveness of branch-and-price relies heavily on the branching strategies employed and the ability to integrate various constraints into the reformulated problem during column generation.

In this thesis, we explore advanced branching rules and constraints within the context of the branch-and-price framework, particularly focusing on the implementation and evaluation of the component bound branching rule. This rule, as proposed by Desrosiers et al. [3], offers a simpler alternative to Vanderbeck's generic branching scheme [4] for branching on so-called component bound sequences [2]. As both branching rules operate entirely within the reformulated problem, integrating them into a branch-and-price framework requires careful consideration of the underlying mathematical structure and the implementation details.

The foundational concepts of polyhedron representation and the primal simplex algorithm are introduced in Chapter 2, providing the mathematical and algorithmic background necessary for understanding the core methods discussed later. In Chapter 3, we delve into the specifics of column generation and branch-and-price, detailing the algorithms and their implementation, including the Dantzig-Wolfe reformulation, which serves as the basis for the decomposition approach used in branch-and-price.

Chapter 4 provides a brief overview of the `SCIP` Optimization Suite, with a particular focus on the `GCG` solver, which forms the foundation for the implementation work carried out in this thesis. In Chapter 5, we present the component bound branching rule in detail, including its separation procedure and a theoretical comparison to Vanderbeck's generic branching scheme.

One of the major contributions of this thesis is the introduction of a new

interface within GCG for handling constraints that exist solely within the master problem, termed **generic mastercuts**. These constraints, which do not have a direct counterpart in the original problem, require special handling within the solver, particularly with respect to synchronizing master variables across the search tree and applying dual value stabilization. Chapter 6 begins by presenting the conceptual framework and definition of generic mastercuts, followed by an elaboration on the synchronization mechanisms and the application of dual value stabilization for these constraints, highlighting the technical innovations introduced in this thesis.

Chapter 7 focuses on the implementation aspects, detailing how the generic mastercut interface was integrated into GCG, and how it supports the component bound branching rule.

The effectiveness of the component bound branching rule and the generic mastercut interface is rigorously evaluated in Chapter 8. We compare different separation heuristics and analyze the impact of dual value stabilization on the performance of the branching rule. Additionally, a detailed comparison with Vanderbeck's generic branching scheme provides insights into the conditions under which the component bound branching rule may offer advantages.

Chapter 2

Preliminaries

This chapter introduces essential notation and concepts that will be utilized throughout this thesis. Additionally, it provides an overview of key theorems and algorithms that form the basis of the techniques discussed in subsequent chapters. For a comprehensive introduction to these topics, readers are referred to Chapter 1 of the book *Branch-and-Price* by Desrosiers et al. [3].

2.1 Polyhedron Representation

Definition 2.1. *Given k points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathbb{R}^n$, any $\boldsymbol{x} = \sum_{i=1}^{k} \alpha_i \boldsymbol{x}_i$ is a **conic combination** of the $\boldsymbol{x}_i$, iff $\forall i \in \{1, \ldots, k\}.\alpha_i \geq 0$.*

Definition 2.2. *Given k points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathbb{R}^n$, any $\boldsymbol{x} = \sum_{i=1}^{k} \alpha_i \boldsymbol{x}_i$ is a **convex combination** of the $\boldsymbol{x}_i$, iff $\sum_{i=1}^{k} \alpha_i = 1 \wedge \forall i \in \{1, \ldots, k\}.\alpha_i \geq 0$.*

The set of all convex combinations of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ is therefore defined as:

$$\operatorname{conv}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) \coloneqq \left\{ \sum_{i=1}^{k} \alpha_i \boldsymbol{x}_i \mid \alpha_i \geq 0, i = 1..k, \sum_{i=1}^{k} \alpha_i = 1 \right\}$$

Corollary 2.1. *The intersection of two convex sets is convex.*

Definition 2.3. *Let $\mathcal{P}$ be a convex set. A point $\boldsymbol{p} \in \mathcal{P}$ is an **extreme point** of $\mathcal{P}$ if there is no non-trivial convex combination of any two points in $\mathcal{P}$ expressing $\boldsymbol{p}$, i.e.:*

$$\forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{P}.\forall \alpha \in (0,1)\,.\boldsymbol{x}_1 \neq \boldsymbol{x}_2 \implies \boldsymbol{p} \neq \alpha \boldsymbol{x}_1 + (1-\alpha)\boldsymbol{x}_2$$

Definition 2.4. *Let $\mathcal{P}$ be a convex set. A vector $\boldsymbol{r} \in \mathbb{R}^n_+ \backslash \{0\}$ is a **ray** of $\mathcal{P}$ iff $\forall \boldsymbol{x} \in \mathcal{P}.\forall \beta \in \mathbb{R}_+.\boldsymbol{x} + \beta \boldsymbol{r} \in \mathcal{P}$.*

The cone of rays $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k \in \mathbb{R}^n_+ \backslash \{0\}$ is denoted as:

$$\text{cone}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k) \coloneqq \left\{ \sum_{i=1}^{k} \alpha_i \boldsymbol{r}_i \mid \alpha_i \geq 0, i = 1..k \right\}$$

Definition 2.5. *A ray $\boldsymbol{r}$ of $\mathcal{P}$ is an **extreme ray** of $\mathcal{P}$ if there is no non-trivial conic combination of any two rays in $\mathcal{P}$ expressing $\boldsymbol{r}$, i.e.:*

$$\forall \boldsymbol{r}_1, \boldsymbol{r}_2 \in \mathcal{P}. \forall \alpha_1, \alpha_2, \beta \in \mathbb{R}_+ \backslash \{0\}. \boldsymbol{r}_1 \neq \beta \boldsymbol{r}_2 \implies \boldsymbol{r} \neq \alpha_1 \boldsymbol{r}_1 + \alpha_2 \boldsymbol{r}_2$$

Definition 2.6. *A **hyperplane** $\mathcal{H} \subset \mathbb{R}^n$ of a n-dimensional space is a subspace of dimension $n-1$, and can therefore be described using a vector $\boldsymbol{f} \in \mathbb{R}^n$ and a scalar $f \in \mathbb{R}$ as $\mathcal{H} = \{\boldsymbol{x} \mid \boldsymbol{f}^\intercal \boldsymbol{x} = f\}$.*

Corollary 2.2. *Any hyperplane is a convex set.*

Definition 2.7. *A **halfspace** is the set either above or below a hyperplane. A halfspace is open if the points on the hyperplane are excluded; otherwise, it is closed.*

Corollary 2.3. *Any halfspace is a convex set.*

Definition 2.8. *A **polyhedron** $\mathcal{P} \subseteq \mathbb{R}^n$ is defined by the intersection of a set of closed halfspaces, i.e., $\mathcal{P} \coloneqq \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\}$, with $\boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m$.*

Corollary 2.4. *A polyhedron is a convex set.*

Definition 2.9. *The **Minkowski sum** of two sets P, Q is defined by:*

$$P \oplus Q \coloneqq \{\boldsymbol{p} + \boldsymbol{q} \mid \boldsymbol{p} \in P \wedge \boldsymbol{q} \in Q\}$$

Theorem 2.1 (Minkowski-Weyl [3, 5])**.** *For $\mathcal{P} \subseteq \mathbb{R}^n$ the following statements are equivalent:*

1. *$\mathcal{P}$ is a polyhedron, i.e., there exists some finite matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and some vector $\boldsymbol{b} \in \mathbb{R}^m$ such that $P = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$*
2. *There exist fine vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s \in \mathbb{R}^n$ and finite vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_t \in \mathbb{R}^n_+$, such that $P = \text{conv}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s) \oplus \text{cone}(\boldsymbol{r}_1, \ldots, \boldsymbol{r}_t)$*

In simple terms, the Minkowski-Weyl theorem states that any polyhedron can always be defined in two ways: either by its faces, i.e., closed halfspaces, or by its vertices and rays. Any polyhedron can be represented in this way using only its extreme points and extreme rays. Figure 2.1 illustrates this theorem on two exemplary polyhedra.

The following theorem builds upon the Minkowski-Weyl theorem to describe a polyhedron, which is represented by its extreme points $\{\boldsymbol{x}_p\}_{p \in P}$ and extreme rays $\{\boldsymbol{x}_r\}_{r \in R}$, using hyperplanes. Here, the sets P, R are used to index the extreme points and extreme rays, respectively.

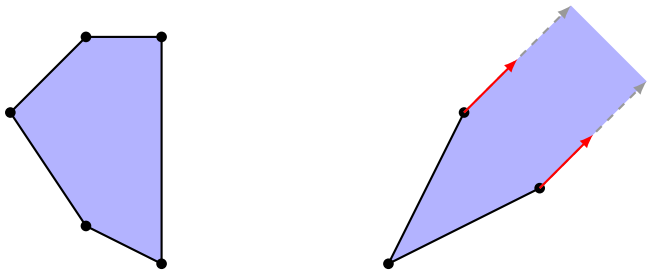


Figure 2.1: Illustration of the Minkowski-Weyl theorem. The left figure shows a polytope represented by its extreme points. Unbounded polyhedra, such as the one on the right, require extreme rays, shown in red, for a complete description.

Theorem 2.2 (Nemhauser-Wolsey [3, 6])**.** *Consider the polyhedron* $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{Q}\boldsymbol{x} \geq \boldsymbol{b}\}$ *with full row rank matrix* $\boldsymbol{Q} \in \mathbb{R}^{m \times n}$*, i.e.,* $\operatorname{rank}(\boldsymbol{Q}) = m \leq n \wedge \mathcal{P} \neq \emptyset$*.*

An equivalent description of $\mathcal{P}$ *using its extreme points* $\{\boldsymbol{x}_p\}_{p \in P}$ *and extreme rays* $\{\boldsymbol{x}_r\}_{r \in R}$ *is:*

$$\mathcal{P} = \left\{ \boldsymbol{x} \in \mathbb{R}^n \;\middle|\; \begin{aligned} \sum_{p \in P} \boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{x}_r \lambda_r &= \boldsymbol{x} \\ \sum_{p \in P} \lambda_p &= 1 \\ \lambda_p &\geq 0 \quad \forall p \in P \\ \lambda_r &\geq 0 \quad \forall r \in R \end{aligned} \right\} \tag{2.1}$$

The conditions of the Minkowski-Weyl theorem are clearly encoded in the Nemhauser-Wolsey theorem: the second and third lines ensure that the convex set of the extreme points is considered in the first line (Definition 2.2), the last pertains to the cone of extreme rays (Definition 2.4), and the first line represents the Minkowski sum of the convex hull of extreme rays and the cone of extreme rays.

By requiring $\boldsymbol{x} \in \mathbb{Z}^n$ in Theorem 2.2, we can also describe the integral polyhedra using possibly fractional extreme points and rays. Alternatively, the Nemhauser-Wolsey theorem has been adapted to describe integral polyhedra using only integral (interior) points and (integer-scaled) extreme rays, as shown in the following theorem.

Theorem 2.3 (Nemhauser-Wolsey [3, 6])**.** *Consider the polyhedron* $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{Q}\boldsymbol{x} \geq \boldsymbol{b}\}$ *with full row rank matrix* $\boldsymbol{Q} \in \mathbb{R}^{m \times n}$*, i.e.,* $\operatorname{rank}(\boldsymbol{Q}) = m \leq n \wedge \mathcal{P} \neq \emptyset$*. Have* $\mathcal{Q} \coloneqq \mathcal{P} \cap \mathbb{Z}^n \neq \emptyset$ *be the integer hull of* $\mathcal{P}$*.*

An equivalent description of $\mathcal{Q}$ *using a finite subset* $\{\boldsymbol{x}_p\}_{p \in \dot{P}}$ *of its integer points and its (integer-scaled) extreme rays* $\{\boldsymbol{x}_r\}_{r \in R}$ *is:*

$$\mathcal{Q} = \left\{ \boldsymbol{x} \in \mathbb{Z}^n \middle| \begin{array}{lll} \sum_{p \in \ddot{P}} \boldsymbol{x}_p \lambda_p + & \sum_{r \in R} \boldsymbol{x}_r \lambda_r = \boldsymbol{x} & \\ \sum_{p \in \ddot{P}} \lambda_p & = 1 & \\ \lambda_p & \in \{0,1\} & \forall p \in \ddot{P} \\ & \lambda_r \in \mathbb{Z}_+ & \forall r \in R \end{array} \right\} \tag{2.2}$$

Note 2.1. A notable difference between the Nemhauser-Wolsey theorem for real polyhedra and integral polyhedra is that for the former it suffices to use extreme points and extreme rays, while for the latter, interior points of $\mathcal{Q}$ might be required to describe the integer hull $\mathcal{Q}$.

2.2 Primal Simplex Algorithm

Consider the following linear program in standard form:

$$\begin{array}{lll} \min & \boldsymbol{c}^\intercal \boldsymbol{x} & \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} & [\boldsymbol{\pi}] \\ & \boldsymbol{x} \geq \boldsymbol{0} & \end{array} \tag{2.3}$$

The primal simplex algorithm [7] finds an optimal solution by moving from one extreme point of the polyhedron to the next, therefore always remaining feasible. A central aspect of this algorithm is the sufficient optimality condition. For a basic solution $\boldsymbol{X} = [\boldsymbol{x}_\mathcal{B}, \boldsymbol{x}_\mathcal{N}]$ at a given extreme point to be optimal, the reduced costs $\bar{c}_j \coloneqq c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j$ for $j \in \mathcal{N}$ must be non-negative.

This sufficient optimality condition leads to the **pricing problem**, which either verifies the optimality of the current basic solution or identifies the non-basic variable x_l, $l \in \mathcal{N}$, with the least reduced cost ($\bar{c}_l < 0$) to be introduced into the basis next, following Dantzig's rule [3, 7, 8]. Formally, this is expressed:

$$l \in \arg\min_{j \in \mathcal{N}} c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j$$

or as the linear program:

$$\bar{c}(\boldsymbol{\pi}) = \min_{j \in \mathcal{N}} c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j \tag{2.4}$$

Solving the pricing problem is integral to the primal simplex algorithm:

Algorithm 2.1: Primal simplex algorithm with Dantzig's rule

Input: *LP* in standard form (2.3); Basic and non-basic index-sets $\mathcal{B}, \mathcal{N}$

Output: Optimal Solution $(\boldsymbol{x}, z)$

loop

  $\boldsymbol{\pi}^\intercal \leftarrow \boldsymbol{c}_{\mathcal{B}}^\intercal \boldsymbol{A}_{\mathcal{B}}^{-1}$; $\bar{\boldsymbol{b}} \leftarrow \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{b}$;

  $\bar{c}_j \leftarrow c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j; \qquad \forall j \in \mathcal{N}$

  $l \leftarrow \underset{j\in\mathcal{N}}{\arg\min}\, \bar{c}_j$; $\bar{c}(\boldsymbol{\pi}) \leftarrow \bar{c}_l$;

  if $\bar{c}(\boldsymbol{\pi}) \geq 0$ **then**

    return $\left(\left[\bar{\boldsymbol{b}}, \boldsymbol{0}\right], \boldsymbol{c}_{\mathcal{B}}^\intercal \boldsymbol{x}_{\mathcal{B}}\right)$ *by optimality*

  end

  $\bar{\boldsymbol{a}}_l \leftarrow \boldsymbol{A}_{\mathcal{B}}^{-1}\boldsymbol{a}_l$;

  if $\bar{\boldsymbol{a}}_l \leq \boldsymbol{0}$ **then**

    return None *by unboundedness*

  end

  $s \leftarrow \underset{i\in\{1,\ldots,m\}}{\arg\min} \frac{\bar{b}_i}{\bar{a}_{il}}$; $x_l \leftarrow \frac{\bar{b}_s}{\bar{a}_{sl}}$; $\mathcal{B} \leftarrow \mathcal{B} \cup \{l\} \subseteq \{s\}$; $\mathcal{N} \leftarrow \mathcal{N} \cup \{s\} \subseteq \{l\}$;

Chapter 3

Column Generation and Branch-and-Price

3.1 Column Generation

Consider the following linear program, referred to as the **master problem** MP, where $c_{\boldsymbol{x}} \in \mathbb{R}, \boldsymbol{a}_{\boldsymbol{x}}, \boldsymbol{b} \in \mathbb{R}^m, \forall \boldsymbol{x} \in \mathcal{X}$:

$$\begin{aligned} z^*_{MP} = \min \quad & \sum_{\boldsymbol{x} \in \mathcal{X}} c_{\boldsymbol{x}} \lambda_{\boldsymbol{x}} \\ \text{s.t.} \quad & \sum_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{a}_{\boldsymbol{x}} \lambda_{\boldsymbol{x}} \geq \boldsymbol{b} \quad [\boldsymbol{\pi}] \\ & \lambda_{\boldsymbol{x}} \geq 0 \qquad \forall \boldsymbol{x} \in \mathcal{X} \end{aligned} \tag{3.1}$$

Assume the number of variables is significantly larger than the number of constraints ($m \ll |\mathcal{X}| < \infty$). Because of this, solving MP directly in a reasonable time frame, or at all, is often infeasible [3].

However, we can utilize a crucial property of the primal simplex algorithm: at any given vertex solution, only few variables are in the basis. Most variables are in the non-basis and therefore have a solution value of 0. Having a solution value of 0 is equivalent to not being in the linear program at all. Therefore, the primal simplex algorithm can operate using a manageable subset of variables $\mathcal{X}' \subseteq \mathcal{X}$, finding a feasible, though possibly non-optimal, solution for MP. This master problem restricted to a subset of variables is called the **restricted master problem** RMP:

$$\begin{aligned} z^*_{RMP} = \min \quad & \sum_{\boldsymbol{x} \in \mathcal{X}'} c_{\boldsymbol{x}} \lambda_{\boldsymbol{x}} \\ \text{s.t.} \quad & \sum_{\boldsymbol{x} \in \mathcal{X}'} \boldsymbol{a}_{\boldsymbol{x}} \lambda_{\boldsymbol{x}} \geq \boldsymbol{b} \quad [\boldsymbol{\pi}] \\ & \lambda_{\boldsymbol{x}} \geq 0 \qquad \forall \boldsymbol{x} \in \mathcal{X}' \end{aligned} \tag{3.2}$$

Assuming *MP* is feasible, two important questions arise for finding an optimal solution to *MP* by solving *RMP*: first, how do we select a subset $\mathcal{X}'$ of variables, such that *RMP* remains feasible? Without this property, no solution for *RMP* can be found, which would contradict the feasibility of *MP*. Secondly, assuming a solution for *RMP* is found, possibly even optimal for the *RMP*, how can we extend this solution to eventually find an optimal solution for *MP*?

The following sections address these questions in detail (Sections 3.1.1 and 3.1.2), leading to the complete column generation algorithm (Section 3.1.3).

3.1.1 Farkas Pricing *FP-SP*

Assume *MP* is feasible, but the current selection of variables $\mathcal{X}' \subset \mathcal{X}$ results in *RMP* being infeasible. The task is to find additional variables such that a new set $\mathcal{X}''$ with $\mathcal{X}' \subset \mathcal{X}'' \subseteq \mathcal{X}$ makes *RMP* feasible. Consider Farkas' lemma:

Theorem 3.1 (Farkas' lemma [3, 9])**.** *Given* $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ *and* $\boldsymbol{b} \in \mathbb{R}^m$*, then exactly one of the following statements holds:*

1. $\exists \boldsymbol{x} \in \mathbb{R}^n_+ . \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$
2. $\exists \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A} \leq \boldsymbol{0} \wedge \boldsymbol{\pi}^\intercal \boldsymbol{b} > 0$

Given that *MP* is feasible, the following must hold for *MP* with $\boldsymbol{A} = \boldsymbol{A}_{|\mathcal{X}}$:

$$\begin{aligned} &\neg \exists \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A} \leq \boldsymbol{0} \wedge \boldsymbol{\pi}^\intercal \boldsymbol{b} > 0 \\ \Leftrightarrow &\forall \boldsymbol{\pi} \in \mathbb{R}^m . \neg \left(\boldsymbol{\pi}^\intercal \boldsymbol{A} \leq \boldsymbol{0} \wedge \boldsymbol{\pi}^\intercal \boldsymbol{b} > 0 \right) \\ \Leftrightarrow &\forall \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A} > \boldsymbol{0} \vee \boldsymbol{\pi}^\intercal \boldsymbol{b} \leq 0 \end{aligned} \tag{3.3}$$

Considering the infeasibility of *RMP* we can further derive the following statement:

$$\begin{aligned} &\left(\forall \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A} > \boldsymbol{0} \vee \boldsymbol{\pi}^\intercal \boldsymbol{b} \leq 0 \right) \wedge \left(\exists \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A}_{|\mathcal{X}'} \leq \boldsymbol{0} \wedge \boldsymbol{\pi}^\intercal \boldsymbol{b} > 0 \right) \\ \Rightarrow &\left(\neg \forall \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{b} \leq 0 \right) \wedge \left(\exists \boldsymbol{\pi} \in \mathbb{R}^m . \boldsymbol{\pi}^\intercal \boldsymbol{A} > \boldsymbol{0} \right) \end{aligned} \tag{3.4}$$

Therefore, there exists some variable $\boldsymbol{x} \in \mathcal{X} \backslash \mathcal{X}'$ such that its column $\boldsymbol{a}_{\boldsymbol{x}} \coloneqq \boldsymbol{A}_{|\{\boldsymbol{x}\}}$ satisfies $\boldsymbol{\pi}^\intercal \boldsymbol{a}_{\boldsymbol{x}} > 0$ for some $\boldsymbol{\pi} \in \mathbb{R}^m$. If none existed, *MP* would not be feasible.

This process of finding corresponding columns $\boldsymbol{a}_{\boldsymbol{x}}$ to add to *RMP* can be formalized as a pricing problem with cost coefficients $c_{\boldsymbol{x}} = 0$ (see Equation (2.4)). Let us denote this subproblem as the *FP-SP*:

$$\mathrm{F}(\boldsymbol{\pi}) = \min_{x \in \mathcal{X}} \; - \boldsymbol{\pi}^\intercal \boldsymbol{a}_x \tag{3.5}$$

We can add all solutions $\boldsymbol{x}$ with a value of $\mathrm{F}(\boldsymbol{\pi}) < 0$ to $\mathcal{X}'' \coloneqq \mathcal{X}' \cup \{\boldsymbol{x}_i\}$, adding the corresponding column $\begin{bmatrix} 0 \\ \boldsymbol{a_x} \end{bmatrix}$ to the problem, thus turning any infeasible *RMP* feasible [3].

3.1.2 Reduced Cost Pricing *RCP-SP*

Assume *RMP* is feasible. Using an appropriate solver, we can construct a solution that is optimal within *RMP*, providing us with the dual values $\boldsymbol{\pi}$. We now need to verify whether this solution is also optimal for *MP*. For this, we can use the familiar pricing problem from the primal simplex algorithm (see Equation (2.4)). Let us denote this subproblem as the *RCP-SP*:

$$\bar{c}(\boldsymbol{\pi}) = \min_{\boldsymbol{x} \in \mathcal{X}} c_{\boldsymbol{x}} - \boldsymbol{\pi}^\intercal \boldsymbol{a_x} \tag{3.6}$$

If $\bar{c}(\boldsymbol{\pi}) \geq 0$, no cost-improving column exists and we have proven the optimality of the current solution for *MP*. Otherwise, if $\bar{c}(\boldsymbol{\pi}) < 0$, there exists some $\boldsymbol{x} \in \mathcal{X} \backslash \mathcal{X}'$ with $\bar{c}(\boldsymbol{\pi}) = c_{\boldsymbol{x}} - \boldsymbol{\pi}^\intercal \boldsymbol{a_x} < 0$. Similar to how the primal simplex algorithm would swap this variable into the basis, during column generation we add the corresponding column $\begin{bmatrix} c_{\boldsymbol{x}} \\ \boldsymbol{a_x} \end{bmatrix}$ to *RMP* [3]. This process ensures that *RMP* remains feasible.

3.1.3 Column Generation Algorithm

The column generation algorithm can be viewed as a variation of the primal simplex algorithm. We start by solving the *MP* with a subset of the original variables, initialized either as an empty set or using some selection heuristics. If this restricted master problem *RMP* is infeasible, we use Farkas pricing to find new variables to add to *RMP*, either until it becomes feasible or until there are no new variables to add, proving the infeasibility of *MP*. Once *RMP* is feasible, we solve it to optimality, using reduced cost pricing to verify whether the solution is also optimal for *MP*. If it is, we have found the optimal solution to *MP*. Otherwise, we add the corresponding column to *RMP* and repeat the process [3].

Note 3.1. All columns in *RMP* are distinct by design. If a subproblem produced a column already present in *RMP*, its reduced cost would be non-negative, and it would not be added to *RMP*.

Algorithm 3.1: Column Generation Algorithm

Input: *RMP* with subset $\mathcal{X}' \subseteq \mathcal{X}$, *RCP-SP*, *FP-SP*
Output: Optimal Solution $(\boldsymbol{\lambda}, z)$ for the *MP*

while IsInfeasible(*RMP*) **do**
  $(\textbf{None}, \boldsymbol{\pi}) \leftarrow \text{Solve}(RMP)$;
  $(\boldsymbol{x}, \text{F}(\boldsymbol{\pi})) \leftarrow \text{Solve}(FP\text{-}SP, \boldsymbol{\pi})$;
  if $\text{F}(\boldsymbol{\pi}) \geq 0$ **then**
    return None *by MP infeasibility*
  end
  $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\boldsymbol{x}\}$;
  $\boldsymbol{A} \leftarrow \begin{bmatrix} \boldsymbol{A} & \boldsymbol{a}_{\boldsymbol{x}} \end{bmatrix}$;
end
loop
  $(\boldsymbol{\lambda}_{RMP}, \boldsymbol{\pi}) \leftarrow \text{Solve}(RMP)$;
  $(\boldsymbol{x}, \bar{c}(\boldsymbol{\pi})) \leftarrow \text{Solve}(RCP\text{-}SP, \boldsymbol{\pi})$;
  if $\bar{c}(\boldsymbol{\pi}) \geq 0$ **then**
    return $(\boldsymbol{\lambda}_{RMP}, \boldsymbol{c}_{\mathcal{B}}^{\intercal} \boldsymbol{x}_{\mathcal{B}})$ *by optimality*
  end
  $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\boldsymbol{x}\}$;
  $\boldsymbol{A} \leftarrow \begin{bmatrix} \boldsymbol{A} & \boldsymbol{a}_{\boldsymbol{x}} \end{bmatrix}$;

3.2 Dantzig-Wolfe Reformulation

The column generation algorithm presented in Section 3.1 is particularly useful when we can directly formulate our optimization problem using a master and a pricing problem. However, many problems are provided in the general form of a *LP*. Using the Dantzig-Wolfe reformulation, we can transform such a *LP* into a master and pricing problem, making it suitable for column generation [3]. This section introduces this technique and demonstrates how it can be applied to solve a *LP*.

$$\begin{aligned} z^*_{LP} = \min \quad & \boldsymbol{c}^\intercal \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \quad [\boldsymbol{\sigma_b}] \\ & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \quad [\boldsymbol{\sigma_d}] \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{3.7}$$

Consider the above *LP*. The solution space defined by its constraints can be viewed as the intersection of the following two polyhedra:

$$\begin{aligned} \mathcal{A} &\coloneqq \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\} \quad \neq \emptyset \\ \mathcal{D} &\coloneqq \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\} \quad \neq \emptyset \end{aligned} \tag{3.8}$$

Applying the Nemhauser-Wolsey Theorem (Theorem 2.2) on polyhedron $\mathcal{D}$, we can reformulate the *LP* using $\mathcal{D}$'s extreme points $\{\boldsymbol{x}_p\}_{p \in P}$ and extreme rays $\{\boldsymbol{x}_r\}_{r \in R}$. To achieve this, we substitute the original variables $\boldsymbol{x}$ with these extreme points and extreme rays using:

$$\begin{aligned} \boldsymbol{x} &= \sum_{p \in P} \boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{x}_r \lambda_r \\ \boldsymbol{c}^\intercal \boldsymbol{x} &= \sum_{p \in P} \boldsymbol{c}^\intercal \boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{c}^\intercal \boldsymbol{x}_r \lambda_r \\ \boldsymbol{A}\boldsymbol{x} &= \sum_{p \in P} \boldsymbol{A}\boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{A}\boldsymbol{x}_r \lambda_r \end{aligned} \tag{3.9}$$

Using the following shorthand notations:

$$\begin{aligned} c_p &\coloneqq \boldsymbol{c}^\intercal \boldsymbol{x}_p \quad c_r \coloneqq \boldsymbol{c}^\intercal \boldsymbol{x}_r \\ \boldsymbol{a}_p &\coloneqq \boldsymbol{A}\boldsymbol{x}_p \quad \boldsymbol{a}_r \coloneqq \boldsymbol{A}\boldsymbol{x}_r \end{aligned} \tag{3.10}$$

We obtain a new *MP* equivalent to the *LP*:

$$
\begin{aligned}
z^*_{MP} = \min \quad & \sum_{p\in P} c_p\lambda_p + \quad \sum_{r\in R} c_r\lambda_r & & \\
\text{s.t.} \quad & \sum_{p\in P} \boldsymbol{a}_p\lambda_p + \quad \sum_{r\in R} \boldsymbol{a}_r\lambda_r \geq \boldsymbol{b} & & [\boldsymbol{\pi_b}] \\
& \sum_{p\in P} \lambda_p = 1 & & [\pi_0] \\
& \lambda_p \geq 0 & & \forall p \in P \\
& \lambda_r \geq 0 & & \forall r \in R \\
& \sum_{p\in P} \boldsymbol{x}_p\lambda_p + \quad \sum_{r\in R} \boldsymbol{x}_r\lambda_r = \boldsymbol{x} \geq \boldsymbol{0} & &
\end{aligned}
\tag{3.11}
$$

In this formulation, the last constraint corresponds to projecting a solution of the *MP* using the λ variables back into a solution of the original *LP*. As this constraint is not otherwise involved in the optimization, it is often omitted during the solving stages and only used afterward to reconstruct a solution using the original $\boldsymbol{x}$ variables [3].

Since the extreme points and extreme rays of $\mathcal{D}$ are often unknown, and their number might be enormous, solving the *MP* directly is typically infeasible [3]. Instead, we can generate these columns iteratively using column generation. For this, we need a subproblem that finds (improving) columns for the *MP*, i.e., extreme points and extreme rays of $\mathcal{D}$. We can formulate this pricing problem as follows:

$$
\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \pi_0 & & \\
\text{s.t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & & [\boldsymbol{\pi_d}] \\
& \boldsymbol{x} \geq \boldsymbol{0} & &
\end{aligned}
\tag{3.12}
$$

We start by solving the *RMP* using a subset of the extreme points $P' \subset P$ and extreme rays $R' \subset R$, yielding the dual values $\boldsymbol{\pi_b}$ and π_0 for the *SP*. Solving this *SP* to optimality then leads to a solution $\boldsymbol{x}^*$ with objective value z^*_{SP}. The value of z^*_{SP} determines whether we add a column to *RMP*, and if so, which column to add:

- If $-\infty < z^*_{SP} < 0$, $\boldsymbol{x}^*$ is an extreme point $\boldsymbol{x}_p, p \in P\backslash P'$, and we add column $\begin{bmatrix} \boldsymbol{c}^\intercal\boldsymbol{x}^* \\ \boldsymbol{A}\boldsymbol{x}^* \\ 1 \end{bmatrix}$ to *RMP*.
- If $z^*_{SP} = -\infty$, $\boldsymbol{x}^*$ is an extreme ray $\boldsymbol{x}_r, r \in R\backslash R'$, and we add column $\begin{bmatrix} \boldsymbol{c}^\intercal\boldsymbol{x}^* \\ \boldsymbol{A}\boldsymbol{x}^* \\ 0 \end{bmatrix}$ to *RMP*.

- If $z^*_{SP} \geq 0$, there exists no improving column for RMP, and the column generation algorithm terminates.

While theoretically, the grouping of constraints in the original LP formulation for the Dantzig-Wolfe reformulation does not change the optimal solution, in practice, the choice of grouping can significantly impact the performance of the column generation algorithm. Since many iterations of the column generation algorithm are often required to find an optimal solution, ideally, one wants SP to be efficiently solvable. Numerous efficient algorithms for specific optimization problems exist, and by grouping constraints in a way that SP corresponds to such structures, one can leverage these algorithms to solve SP efficiently [3]. Although there are ways to find such groupings automatically, this topic is beyond the scope of this thesis.

3.3 Dantzig-Wolfe Reformulation for Integer Programs

Dantzig-Wolfe reformulation can also be applied to integer programs. In this section, we will show how to reformulate an integer program into a master and pricing problem, specifically focusing on the integrality conditions. Later, in Section 3.5, we will explore how to solve such an integer program using column generation.

Consider the following integer program:

$$\begin{aligned} z^*_{IP} = \min \quad & \boldsymbol{c}^\intercal \boldsymbol{x} & \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} & [\boldsymbol{\sigma}_b] \\ & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & [\boldsymbol{\sigma}_d] \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ & \end{aligned} \tag{3.13}$$

Again, we group the constraints into two sets:

$$\begin{aligned} \mathcal{A} &:= \{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\} \quad \neq \emptyset \\ \mathcal{D} &:= \{\boldsymbol{x} \in \mathbb{Z}^n \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\} \quad \neq \emptyset \end{aligned} \tag{3.14}$$

Note that $\mathcal{A}$ and $\mathcal{D}$ are now the integer hulls of the original polyhedra. For simplicity, let us denote the convex hulls defined by both groups of constraints as:

$$\begin{aligned} \mathcal{A}' &:= \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\} \quad \neq \emptyset \\ \mathcal{D}' &:= \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\} \quad \neq \emptyset \end{aligned} \tag{3.15}$$

There are two ways to proceed from here. The straightforward approach, called **Convexification**, follows the method seen in the Dantzig-Wolfe reformulation of

linear programs, retaining the integrality constraints on $\boldsymbol{x}$ in both the master and pricing problem. Alternatively, in **Discretization**, we modify our approach slightly, adding integrality constraints to the master variables to ensure the integrality of the original variables.

3.3.1 Convexification

As seen in Section 3.2, we can reformulate the polyhedron $\mathcal{D}$, which is now the integer hull defined by the constraints $\boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}$, using the Nemhauser-Wolsey Theorem (Theorem 2.2). This results in a master problem where the original variables $\boldsymbol{x}$ are represented as a convex combination of extreme points and extreme rays of $\mathcal{D}$:

$$\begin{aligned}
z^*_{MP} = \min \quad & \sum_{p\in P} c_p\lambda_p + \sum_{r\in R} c_r\lambda_r & & \\
\text{s. t.} \quad & \sum_{p\in P} \boldsymbol{a}_p\lambda_p + \sum_{r\in R} \boldsymbol{a}_r\lambda_r \geq \boldsymbol{b} & [\boldsymbol{\pi_b}] & \\
& \sum_{p\in P} \lambda_p = 1 & [\pi_0] & \\
& \lambda_p \geq 0 & & \forall p \in P \\
& \lambda_r \geq 0 & & \forall r \in R \\
& \sum_{p\in P} \boldsymbol{x}_p\lambda_p + \sum_{r\in R} \boldsymbol{x}_r\lambda_r = \boldsymbol{x} \in \mathbb{Z}^n_+ & &
\end{aligned} \tag{3.16}$$

In contrast to the Dantzig-Wolfe reformulation for linear programs, during convexification the last constraint, which reconstructs an original solution using a solution of the master problem, is crucial during the solving process to ensure the integrality of the original variables and cannot simply be computed after a solution has been found. The master problem has the following pricing subproblem:

$$\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \pi_0 & & \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & [\boldsymbol{\pi_d}] & \\
& \boldsymbol{x} \in \mathbb{Z}^n_+ & &
\end{aligned} \tag{3.17}$$

These two changes marked in blue are the key differences between the Dantzig-Wolfe reformulation of linear programs and integer programs, ensuring that we find integer solutions for our original problem.

The beauty of this approach lies in the fact that the subproblem only generates the extreme points and extreme rays of the integer hull of $\{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\}$, regardless of how well the constraints $\boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}$ approximate this integer hull. Therefore, we implicitly make use of the integer hull of $\mathcal{D}$ without explicitly defining it.

λ solutions to the *MP* might lead to fractional $\boldsymbol{x}$ solutions. In this case, we must branch on those fractional original variables [3]. We will discuss this in more detail in Section 3.5.1.

3.3.2 Discretization

In the discretization approach, we use the adaptation of the Nemhauser-Wolsey Theorem to integer polyhedra (Theorem 2.3) to reformulate the polyhedron $\mathcal{D}$, yielding the following master problem:

$$\begin{aligned}
z^*_{MP} = \min \quad & \sum_{p\in\ddot{P}} c_p\lambda_p + \sum_{r\in R} c_r\lambda_r & & \\
\text{s.t.} \quad & \sum_{p\in\ddot{P}} \boldsymbol{a}_p\lambda_p + \sum_{r\in R} \boldsymbol{a}_r\lambda_r \geq \boldsymbol{b} & & [\boldsymbol{\pi_b}] \\
& \sum_{p\in\ddot{P}} \lambda_p = 1 & & [\pi_0] \\
& \lambda_p \in \{0,1\} & & \forall p \in \ddot{P} \\
& \lambda_r \in \mathbb{Z}_+ & & \forall r \in R \\
& \sum_{p\in\ddot{P}} \boldsymbol{x}_p\lambda_p + \sum_{r\in R} \boldsymbol{x}_r\lambda_r = \boldsymbol{x} \in \mathbb{Z}^n_+ & &
\end{aligned} \tag{3.18}$$

By design, a solution to the master problem is now guaranteed to be transformable into an integer solution of the original problem. Therefore, the last constraint can be omitted during the solving process. Solving the linear relaxation of the *RMP* might lead to fractional λ variables, which we can then branch on [3]. Keeping in mind that $\ddot{P}$ is a subset of integer points of $\mathcal{D}$, i.e., it might include interior points, we must find a pricing problem that can generate not only extreme points (and rays) of $\mathcal{D}$ but also interior points. This, however, is not trivial, since mathematical optimization typically focuses on finding the most optimal solutions, i.e., the extreme points. We can postpone this concern for now and use the same pricing problem as in the convexification approach:

$$\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \pi_0 & & \\
\text{s.t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & & [\boldsymbol{\pi_d}] \\
& \boldsymbol{x} \in \mathbb{Z}^n_+ & &
\end{aligned} \tag{3.19}$$

As we will discuss in Section 3.5.2, the concern of generating interior points is addressed during the branching process, allowing us to generate such points on the fly. Therefore, combined with branching, the discretization approach is also a viable method to solve integer programs using column generation.

3.4 Several and Identical Subproblems

Many applications are composed of different families of variables and constraints, which can be decomposed into several distinct subproblems. Column generation can be adapted to this scenario, where we have a set K of subproblems SP^k generating variables $\boldsymbol{x}^k \in \mathcal{X}^k$ [3]. Our *MP* is then defined as:

$$\begin{aligned} z^*_{MP} = \min \quad & \sum_{k \in K} \sum_{\boldsymbol{x}^k \in \mathcal{X}^k} c_{\boldsymbol{x}^k} \lambda_{\boldsymbol{x}^k} \\ \text{s. t.} \quad & \sum_{k \in K} \sum_{\boldsymbol{x}^k \in \mathcal{X}^k} \boldsymbol{a}_{\boldsymbol{x}^k} \lambda_{\boldsymbol{x}^k} \geq \boldsymbol{b} \quad [\boldsymbol{\pi}] \\ & \lambda_{\boldsymbol{x}} \geq 0 \qquad \forall k \in K . \forall \boldsymbol{x}^k \in \mathcal{X}^k \end{aligned} \tag{3.20}$$

All subproblems SP^k now use the same dual values $\boldsymbol{\pi}$, and the pricing problem for each subproblem SP^k is defined as:

$$z^*_{SP^k} = \min_{\boldsymbol{x}^k \in \mathcal{X}^k} c_{\boldsymbol{x}^k} - \boldsymbol{\pi}^\intercal \boldsymbol{a}_{\boldsymbol{x}^k} \tag{3.21}$$

The column generation algorithm from Section 3.1.3 proceeds as before, with the adaptation that it terminates only when *all* subproblems SP^k produce columns with non-negative reduced costs.

This idea of having several subproblems generating columns for the master problem can also be applied to Dantzig-Wolfe reformulated *LP*s and *IP*s. Recall that we find two groups of constraints:

$$\begin{aligned} \mathcal{A} &:= \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\} \quad \neq \emptyset \\ \mathcal{D} &:= \{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\} \quad \neq \emptyset \end{aligned} \tag{3.22}$$

In many applications, the coefficient matrix $\boldsymbol{D}$ has a block diagonal structure [3]:

$$\boldsymbol{D} = \begin{bmatrix} \boldsymbol{D}^1 & & \\ & \ddots & \\ & & \boldsymbol{D}^{|K|} \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}^1 \\ \vdots \\ \boldsymbol{d}^{|K|} \end{bmatrix} \tag{3.23}$$

Each of these $k \in K$ blocks can be considered its own subproblem independent of others. Therefore, another way of writing the *MP* for Dantzig-Wolfe reformulated *LP*s is (analogous for convexification and discretization of *IP*s):

$$
\begin{aligned}
z^*_{MP} = \min \quad & \sum_{k\in K}\sum_{p\in P^k} c_p^k\lambda_p^k + \quad \sum_{k\in K}\sum_{r\in R^k} c_r^k\lambda_r^k & & \\
\text{s. t.} \quad & \sum_{k\in K}\sum_{p\in P^k} \boldsymbol{a}_p^k\lambda_p^k + \quad \sum_{k\in K}\sum_{r\in R^k} \boldsymbol{a}_r^k\lambda_r^k \geq \boldsymbol{b} & & [\boldsymbol{\pi_b}] \\
& \sum_{p\in P^k} \lambda_p^k & = 1 & \left[\pi_0^k\right] \forall k \in K \\
& \lambda_p^k & \geq 0 & \forall k \in K, \forall p \in P^k \\
& \lambda_r^k & \geq 0 & \forall k \in K, \forall r \in R^k \\
& \sum_{p\in P^k} \boldsymbol{x}_p^k\lambda_p^k + \quad \sum_{r\in R^k} \boldsymbol{x}_r^k\lambda_r^k = \boldsymbol{x}^k \geq \boldsymbol{0} & & \forall k \in K
\end{aligned}
\tag{3.24}
$$

Each subproblem SP^k is given by:

$$
\begin{aligned}
z^*_{SP^k} = \min \quad & \left(\boldsymbol{c}^{k\intercal} - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A}^k\right)\boldsymbol{x}^k - \pi_0^k & & \\
\text{s. t.} \quad & \boldsymbol{D}^k\boldsymbol{x}^k \geq \boldsymbol{d}^k & \left[\boldsymbol{\pi}_{\boldsymbol{d}}^k\right] & \\
& \boldsymbol{x}^k \geq \boldsymbol{0} & &
\end{aligned}
\tag{3.25}
$$

Now, consider the case where all blocks are equal, i.e., $\boldsymbol{D}^1 = \ldots = \boldsymbol{D}^{|K|} = \boldsymbol{D}$ and $\boldsymbol{d}^1 = \ldots = \boldsymbol{d}^{|K|} = \boldsymbol{d}$. In this case, all subproblems SP^k are identical, generating new columns from the same set of extreme points and extreme rays. This implies that in MP, different λ_p^k (λ_r^k) variables for different k correspond to the same extreme point $\boldsymbol{x}_p$ ($\boldsymbol{x}_r$), which is redundant and could slow down the solving process [3]. In a process called **aggregation**, we can improve upon this by eliminating this redundancy:

$$
\lambda_p \coloneqq \sum_{k\in K} \lambda_p^k, \ \forall p \in P \qquad \text{and} \qquad \lambda_r \coloneqq \sum_{k\in K} \lambda_r^k, \ \forall r \in R \tag{3.26}
$$

Substituting these aggregated variables in MP yields:

$$
\begin{aligned}
z^*_{MP} = \min & \sum_{p\in P} c_p\lambda_p + \sum_{r\in R^k} c_r\lambda_r && && && (3.27a)\\
\text{s.t.} & \sum_{p\in P} \boldsymbol{a}_p\lambda_p + \sum_{r\in R^k} \boldsymbol{a}_r\lambda_r \geq \boldsymbol{b} && [\boldsymbol{\pi_b}] && && (3.27b)\\
& \sum_{p\in P} \lambda_p = |K| && [\pi_{agg}] && && (3.27c)\\
& \lambda_p \geq 0 && && \forall p \in P && (3.27d)\\
& \lambda_r \geq 0 && && \forall r \in R && (3.27e)\\
& \sum_{k\in K} \lambda^k_p = \lambda_p && && \forall p \in P && (3.27f)\\
& \sum_{k\in K} \lambda^k_r = \lambda_r && && \forall r \in R && (3.27g)\\
& \sum_{p\in P} \lambda^k_p = 1 && && \forall k \in K && (3.27h)\\
& \lambda^k_p \geq 0 && && \forall k \in K, \forall p \in P && (3.27i)\\
& \lambda^k_r \geq 0 && && \forall k \in K, \forall r \in R && (3.27j)\\
& \sum_{p\in P} \boldsymbol{x}_p\lambda^k_p + \sum_{r\in R} \boldsymbol{x}_r\lambda^k_r = \boldsymbol{x}^k \geq \boldsymbol{0} && && \forall k \in K && (3.27k)
\end{aligned}
$$

Columns for this *MP* are generated by the following subproblem:

$$
\begin{aligned}
z^*_{SP^{agg}} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \pi_{agg} && \\
\text{s.t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} && [\boldsymbol{\pi_d}] \\
& \boldsymbol{x} \geq \boldsymbol{0} &&
\end{aligned}
\tag{3.28}
$$

The constraints (3.27f) to (3.27j) disaggregate a solution for the aggregated variables back into the master variables for each subproblem, which are used to compute a solution to the original formulation using the original variables $\boldsymbol{x}^k$. For this reason, the constraints from (3.27f) onwards may be omitted during the column generation algorithm. This statement also holds for Dantzig-Wolfe reformulated *IP*s using the convexification approach, where the only difference in *MP* are the integrality conditions on $\boldsymbol{x}^k$ in constraint (3.27k). In convexification, however, we can only ensure the integrality of the original solution by branching on the integer original variables with fractional value. Therefore, we constantly need to reintroduce the disaggregated master variables to project a solution of *RMP* to an original solution.

Discretization, however, offers a powerful alternative. Its *MP* for identical subproblems looks as follows:

$$z^*_{MP} = \min \sum_{p \in P} c_p \lambda_p + \sum_{r \in R^k} c_r \lambda_r \tag{3.29a}$$

$$\text{s. t.} \sum_{p \in P} \boldsymbol{a}_p \lambda_p + \sum_{r \in R^k} \boldsymbol{a}_r \lambda_r \geq \boldsymbol{b} \qquad [\boldsymbol{\pi_b}] \tag{3.29b}$$

$$\sum_{p \in P} \lambda_p = |K| \qquad [\pi_{agg}] \tag{3.29c}$$

$$\lambda_p \in \mathbb{Z}_+ \qquad \forall p \in P \tag{3.29d}$$

$$\lambda_r \in \mathbb{Z}_+ \qquad \forall r \in R \tag{3.29e}$$

$$\sum_{k \in K} \lambda^k_p = \lambda_p \qquad \forall p \in P \tag{3.29f}$$

$$\sum_{k \in K} \lambda^k_r = \lambda_r \qquad \forall r \in R \tag{3.29g}$$

$$\sum_{p \in P} \lambda^k_p = 1 \qquad \forall k \in K \tag{3.29h}$$

$$\lambda^k_p \in \mathbb{Z}_+ \qquad \forall k \in K, \forall p \in P \tag{3.29i}$$

$$\lambda^k_r \in \mathbb{Z}_+ \qquad \forall k \in K, \forall r \in R \tag{3.29j}$$

$$\sum_{p \in P} \boldsymbol{x}_p \lambda^k_p + \sum_{r \in R} \boldsymbol{x}_r \lambda^k_r = \boldsymbol{x}^k \in \mathbb{Z}^n_+ \qquad \forall k \in K \tag{3.29k}$$

In Section 3.3.2, we have observed that the integrality constraints on the original variables $\boldsymbol{x}^k$ are already enforced by ensuring the integrality of the disaggregated master variables λ^k_p and λ^k_r. In the case of identical subproblems, we can go a step further and also neglect the integrality constraints on the disaggregated master variables, as those are implied by the integrality of the aggregated variables λ_p and λ_r [3]. Therefore, during the entire solving process, we can omit the constraints (3.29f) to (3.29k) entirely.

On a final note, it is possible to have both identical and differing subproblems in the same *MP*. In this case, we introduce classes C of identical subproblems, use one column generator per class, and aggregate the variables within each class.

3.5 Branch-and-Price

In Section 3.3, we discussed reformulating an integer program into a master and pricing problem with a focus on integrality conditions. In this section, we explore solving an integer master program using column generation. Branching is essential when an optimal solution of the *LP* relaxation has fractional values for the integer variables, making the solution infeasible for the *IP*. To address this, we branch on these fractional variables, creating subproblems that explicitly exclude these

fractional solutions. By recursively solving these subproblems, we eventually find an optimal integer solution, a process known as **branch-and-bound**.

In the context of column generation for integer master programs, we follow a similar approach: first, we relax the integrality constraints of the master problem, allowing us to solve the relaxation using column generation to optimality. Then, we check if the integrality conditions are satisfied. If not, we must cut off the fractional solution by branching, and re-optimize using column generation [3]. This technique of combining branching with column generation is referred to as **branch-and-price**.

We have seen two distinct approaches to reformulating an *IP* into an integer master and pricing problem: convexification (Section 3.3.1) and discretization (Section 3.3.2). Since integrality of the original variables is required in both approaches, we can always branch on fractional solutions of the original variables. However, discretization introduces additional integrality constraints on the master variables, which imply the integrality of the original variables. Therefore, in discretization, we can also branch on the master variables. In the following, we will discuss both approaches in more detail.

3.5.1 Branching on the Original Variables

Assume we have a fractional solution $\boldsymbol{x}^*_{RMP}$ to the relaxed restricted master problem *RMP*, i.e., there at least one integer variable x_j for which $x^*_j \notin \mathbb{Z}$. We now must cut off this fractional solution, for example by creating two subbranches, one where $x_j \leq \lfloor x^*_j \rfloor$ and one where $x_j \geq \lceil x^*_j \rceil$ (**dichotomous branching**). In each of these subtrees, a solution to *RMP* should be guaranteed to only use columns that satisfy the branching decision, and during the solving process, the pricing problems should only be able to generate such columns.

Note 3.2. Branching on the original variables allows the subproblem to generate the interior points required for the correctness of the discretization approach, as discussed in Section 3.3.2.

In the branch-and-price context, there are two ways to enforce this branching decision [3]:

3.5.1.1 Branching in the Master Problem

Recall that the *MP* includes the following constraint:

$$\sum_{p \in P} \boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{x}_r \lambda_r = \boldsymbol{x} \in \mathbb{Z}^n_+ \tag{3.30}$$

This constraint is now violated in the case of variable x_j. We can enforce the branching decision $x_j \leq \lfloor x_j^* \rfloor$ by adding the following constraint to *MP* (analogous for the up-branch):

$$\sum_{p \in P} x_{pj} \lambda_p + \sum_{r \in R} x_{rj} \lambda_r \leq \lfloor x_j^* \rfloor \quad [\alpha_j] \tag{3.31}$$

To continue generating only improving columns after branching, we must consider the dual variable α_j in the pricing problem:

$$\begin{aligned} z_{SP}^* = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \alpha_j x_j - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{Z}_+^n \end{aligned} \tag{3.32}$$

3.5.1.2 Branching in the Pricing Problem

Alternatively, we may add the branching decision directly to the pricing problem:

$$\begin{aligned} z_{SP}^* = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & x_j \leq \lfloor x_j^* \rfloor \\ & \boldsymbol{x} \in \mathbb{Z}_+^n \end{aligned} \tag{3.33}$$

However, *RMP* might already contain generated columns that violate the branching decision. To ensure correctness of this implementation of the branching decision, we must forbid all existing columns with $x_j > \lfloor x_j^* \rfloor$ from being part of the solution in the master. This can be achieved by removing such columns altogether or by adding the following constraint to *MP*:

$$\sum_{p \in P : x_{pj} > \lfloor x_j^* \rfloor} \lambda_p + \sum_{r \in R : x_{rj} > \lfloor x_j^* \rfloor} \lambda_r = 0 \tag{3.34}$$

3.5.2 Branching on the Master Variables

Let $Q := \ddot{P} \cup R$. Assume our master solution $\boldsymbol{\lambda}_{RMP}^*$ is fractional, i.e., $\lambda_q^* \notin \mathbb{Z}$ for at least one $q \in Q$. Unfortunately, dichotomous branching on a such a single master variable λ_q is very weak: Assume our problem consists of a single non-aggregated subproblem, and $\lambda_q^* = 0.5$. We then would create the down-branch $\lambda_q = 0$ and the up-branch $\lambda_q = 1$. The former constraint would cut off almost no solutions, while the latter would forbid most solutions. This would lead to an extremely unbalanced branching tree, which is only little better than enumerating all possible solutions

[3]. Cutting off multiple fractional solutions in each child node would be more desirable, i.e., we must branch on constraints, in general of the following form:

$$\sum_{p \in \ddot{P}} f(p)\lambda_p + \sum_{r \in R} f(r)\lambda_r \leq h \qquad [\gamma]$$

Here, the function f determines the coefficient of column p or r in the constraint, and h is a constant. The subproblem would have to respect the dual value γ of the constraint in the pricing problem:

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \gamma f(\boldsymbol{x}) - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \end{aligned}$$

The question one might ask themselves now is how we can find such a function f and constant h that are suitable for branching. Henceforth, consider the case where $Q = \ddot{P}$, i.e., our problem is bounded ($R = \emptyset$). In this case, Vanderbeck proposes we can find a subset $\emptyset \subset Q' \subset Q$ of variables in RMP, for which the following holds [2]:

$$\sum_{q \in Q'} \lambda^*_q =: K \notin \mathbb{Z} \tag{3.35}$$

It is obvious that such a subset Q' always exists, for example choose $Q' = \{\lambda_q\}$ for dichotomous branching. In the master problem, we could then branch on this integrality condition, e.g., in the down branch using:

$$\sum_{q \in Q'} \lambda_q \leq \lfloor K \rfloor \quad [\gamma] \tag{3.36}$$

The corresponding subproblem must be adapted to ensure the validity of the branching decision in the master. In particular, if and only if the pricing problem generates a new column q' for which $q' \in Q'$, the corresponding master variable $\lambda_{q'}$ must be set to 1. This can be achieved by adding the following constraint to the pricing problem:

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \gamma y - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & y = 1 \Leftrightarrow \boldsymbol{x} \in Q' \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \\ & y \in \{0, 1\} \end{aligned} \tag{3.37}$$

This idea Vanderbeck proposed plays nicely into our more general definition, where we set $f(q) = \mathbb{1}_{q \in Q'}$ and, in this down branch, $f = \lfloor K \rfloor$.

What remains is to find a routine to determine such a subset Q' in the master solution, for which the set inclusion rule to be added to *SP* is also *expressible using a finite set of linear constraints.*

Note 3.3. Adding the variable y to the subproblem allows the column generation algorithm to generate the interior points required for the correctness of the discretization approach, as discussed in Section 3.3.2 [2].

Note 3.4. Aggregation of subproblems (Section 3.4) is not an issue when branching in the master. In such cases, we would simply branch on the aggregated variables within each block of identical subproblems. For readability, we focus only on single non-aggregated blocks.

3.5.2.1 Vanderbeck's Generic Branching Scheme

Vanderbeck proposed an elaborate scheme (`GENERIC`) [1, 2] to find such a subset Q' in the master solution, enabling branching on master variables for any type of bounded *IP*, i.e., which has no extreme rays ($Q = \ddot{P}$). This branching rule is based on bounds applied to the components of a column:

$$B \coloneqq (x_i, \eta, v) \in \{x_i \mid 1 \leq i \leq n\} \times \{\leq, \geq\} \times \mathbb{Z} \tag{3.38}$$

$$\bar{B} \coloneqq (x_i, \bar{\eta}, v), \bar{\eta} \coloneqq \begin{cases} \leq & \text{if } \eta = \geq \\ \geq & \text{if } \eta = \leq \end{cases} \tag{3.39}$$

where η is the type of bound, and v is the value of the bound. Furthermore, $\bar{B}$ describes the inverse component bound of B. We can now define a component bound sequence as:

$$S \coloneqq \{(x_{i,1}, \eta_1, v_1), \ldots, (x_{j,k}, \eta_k, v_k)\} \in 2^{\{x_i \mid 1 \leq i \leq n\} \times \{\leq, \geq\} \times \mathbb{Z}} \tag{3.40}$$

Let us further introduce the following shorthand notation:

$$\eta(a, v) \Leftrightarrow \begin{cases} a \leq v & \text{if } \eta = \leq \\ a \geq v & \text{if } \eta = \geq \end{cases} \tag{3.41}$$

For a given component bound sequence S and a set of columns Q, we can define the restriction of Q to S as:

$$Q(S) \coloneqq \{q \in Q \mid \forall (x_i, \eta, v) \in S . \eta(x_{qi}, v)\} \tag{3.42}$$

Note that $Q(\emptyset) = Q$.

We now reduce the problem of finding a subset Q' to finding a component bound sequence S for which the following holds:

- $\sum_{q \in Q(S)} \lambda_q^* =: K \notin \mathbb{Z}$
- $y = 1 \Leftrightarrow \boldsymbol{x} \in Q(S)$ is expressible using a finite set of linear constraints

Proposition 3.1. *If $\boldsymbol{\lambda}^*_{RMP}$ is a fractional solution to the master problem, then there exists a component bound sequence S for which the first condition holds.*

Proof. Let $Q_{fraq} \coloneqq \{q \in Q \mid \lambda_q^* \notin \mathbb{Z}\} \neq \emptyset$ be the set of columns with currently fractional master variables. Then take $q* \coloneqq \arg\min_{q \in Q_{fraq}} \boldsymbol{x}_q$ as any minimal undominated column in Q_{fraq}. From $q*$, we can now construct a component bound sequence S, which is only satisfied by $q*$ out of all $q \in Q_{fraq}$, as follows:

$$S \coloneqq \{(x_i, \leq, \lfloor x_{q*} \rfloor) \mid x_i \in \{x_j \mid q_j \in Q_{fraq}\}\} \tag{3.43}$$

By construction, $Q(S) = \{q*\}$, and thus $\sum_{q \in Q(S)} \lambda_q^* = \lambda_{q*}^* \notin \mathbb{Z}$. □

Vanderbeck's scheme divides the solution space along the component bounds into multiple sub-polyhedra. In this way, each child branch can only generate points within its own sub-polyhedron, and the master solution will be integral within one of these sub-polyhedra. In fact, this scheme closely resembles the main idea of dichotomous branching in branch-and-bound, where the solution space is divided into two halves. For a given component bound sequence $S = \{B_1, \ldots, B_m\}$, where each variable x_i has at least one upper and one lower component bound, there are up to $2^n - 1$ possible sub-polyhedra. To avoid an exponential increases in nodes, we group some sub-polyhedra together, creating a total of $n + 1$ nodes. Each of the $1 \leq j \leq m + 1$ nodes is now modified as follows: first define the component bound sequence S_j for the j-th node as:

$$S_j \coloneqq \begin{cases} \{B_1, \ldots, B_{j-1}, \bar{B}_j\} & \text{if } j \leq m \\ \{B_1, \ldots, B_m\} & \text{if } j = m + 1 \end{cases} \tag{3.44}$$

Determine the fractional value K_j for the j-th node as:

$$K_j \coloneqq \sum_{q \in Q(S_j)} \lambda_q^* \tag{3.45}$$

Then, to *RMP* of node j, add the following constraint:

$$\sum_{q \in Q(S_j)} \lambda_q \geq \lceil K_j \rceil \quad [\gamma_j] \tag{3.46}$$

Finally, modify the pricing problem as follows:

$$
\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \gamma_j - \pi_0 \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\
& x_i \leq v \qquad \forall\,(x_i, \leq, v) \in S_j \\
& x_i \geq v \qquad \forall\,(x_i, \geq, v) \in S_j \\
& \boldsymbol{x} \in \mathbb{Z}^n_+
\end{aligned}
\tag{3.47}
$$

Note 3.5. The modifications made to the pricing problems during Vanderbeck's generic branching still fit the description stated in Equation (3.37) and can be written formally as:

$$
\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \gamma_j y - \pi_0 \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\
& x_i \leq v \qquad \forall\,(x_i, \leq, v) \in S_j \\
& x_i \geq v \qquad \forall\,(x_i, \geq, v) \in S_j \\
& y = 1 \\
& \boldsymbol{x} \in \mathbb{Z}^n_+ \\
& y \in \{0, 1\}
\end{aligned}
\tag{3.48}
$$

The procedure for finding a component bound sequence S as described in Proof 3.5.2.1 leads to dichotomous branching. As discussed earlier, branching on a single master variable leads to an unbalanced tree. To overcome this, Vanderbeck proposes a sophisticated routine that divides the solution space into multiple branches more evenly [4].

This presentation of Vanderbeck's generic branching scheme just covers the main ideas and concepts. For a more in-depth derivation of this rule, detailed descriptions of the routines, and further improvements such as node pruning, we refer to [1, 2, 4, 10].

3.6 Branch-Price-and-Cut

From solving *IP*s, we know that adding cutting planes, or valid inequalities, can significantly enhance the performance of the branch-and-bound algorithm. These cutting planes can be generated and added to the *LP* relaxation at any stage of the solving process, forming the basis of the branch-and-cut algorithm.

We can extend this concept to branch-and-price. Whenever we have a solution for the *LP* relaxation of the *MP*, we can add additional valid inequalities to the *RMP*, aiming to strengthen the relaxation. This extension transforms the branch-and-price algorithm into a branch-price-and-cut algorithm. In general, separators

generating cuts for the *MP* operate either on the original formulation or within the master problem of the Dantzig-Wolfe reformulation. We will briefly discuss these two types of separators. For more detailed information about cutting planes for column generation and their effectiveness, refer to [3, 11, 12].

3.6.1 Separators using the Original Formulation

Assume we have solved the *LP* relaxation of the *MP* to optimality using column generation to obtain the master solution $\boldsymbol{\lambda}^*$, which can be projected back into a solution $\boldsymbol{x}^*$ of the original formulation. We can then call any separation algorithms that operate on the original formulation to generate cuts of the general form:

$$\boldsymbol{F}^\intercal \boldsymbol{x} \geq \boldsymbol{f} \tag{3.49}$$

We can apply Dantzig-Wolfe reformulation to transform these cuts, for example by adding the following constraints to the *MP*:

$$\sum_{p \in P} \boldsymbol{F}\boldsymbol{x}_p \lambda_p + \sum_{r \in R} \boldsymbol{F}\boldsymbol{x}_r \lambda_r \geq \boldsymbol{f} \quad [\boldsymbol{\alpha}] \tag{3.50}$$

and imposing the constraints in the pricing problem:

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A} - \boldsymbol{\alpha}^\intercal \boldsymbol{F})\, \boldsymbol{x} - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \end{aligned} \tag{3.51}$$

This approach allows existing separators originally intended for use in a branch-and-cut scenario to be reused to generate cutting planes for the Dantzig-Wolfe reformulation. However, some caveats apply. For example, some separators rely on a basis solution. Since the Dantzig-Wolfe reformulation might be stronger than the original formulation [3, 13], an interior point of the polyhedron could be the optimal solution for the relaxed *RMP*. In this case, the basis solution is not available, and such a separator cannot be applied directly.

3.6.2 Separators using the Master Problem

Through discretization, we obtain a *MP* with integral master variables. To strengthen the *LP* relaxation of the *MP*, we aim to cut off some fractional solutions. Unfortunately, applying an ordinary branch-and-cut separator to a solution of the *RMP* is undesirable: such cuts would only be defined for variables currently contained in the *RMP*. Instead, we need cuts over all variables in the *MP* that can also be imposed in the subproblem to limit which columns can be generated. Formally, we seek a function $f : \mathbb{R}^n \to \mathbb{R}$ such that the cut is expressible as:

$$\sum_{p \in \ddot{P}} f(\boldsymbol{x}_p)\lambda_p + \sum_{r \in R} f(\boldsymbol{x}_r)\lambda_r \geq h \quad [\gamma] \tag{3.52}$$

requiring the following modifications to the pricing problem:

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \gamma g_{\boldsymbol{x}} - \pi_0 \\ \text{s.t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & g_{\boldsymbol{x}} = f(\boldsymbol{x}) \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \\ & g_{\boldsymbol{x}} \in \mathbb{R} \end{aligned} \tag{3.53}$$

As with branching on master variables (Section 3.5.2), the challenge is expressing $g_{\boldsymbol{x}} = f(\boldsymbol{x})$ using a finite set of linear constraints.

3.7 Dual Value Stabilization

To understand the necessity and method of dual value stabilization, we first consider Lagrangian relaxation. Revisit the following IP, which includes complicating constraints $\boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}$ and simpler constraints $\boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}$:

$$\begin{aligned} z^*_{IP} = \min \quad & \boldsymbol{c}^\intercal \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \quad [\boldsymbol{\pi}_{\boldsymbol{b}}] \\ & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \quad [\boldsymbol{\pi}_{\boldsymbol{d}}] \\ & \boldsymbol{x} \in \mathbb{Z}^n \end{aligned}$$

Recall that during Dantzig-Wolfe reformulation (Section 3.2), we decomposed this problem by separating the complicating constraints from the simpler ones, resulting in the following pricing problem:

$$\begin{aligned} z^*_{SP} = -\pi_0 + \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \quad [\boldsymbol{\pi}_{\boldsymbol{d}}] \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \end{aligned}$$

A common approach to compute a lower bound on z^*_{IP} is to perform a **Lagrangian relaxation** (LR). In this Lagrangian relaxation, we penalize the violation $(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})$ of the complicating constraints using **Lagrangian multipliers** $\boldsymbol{\pi}_{\boldsymbol{b}}$, yielding the following **Lagrangian subproblem** or **Lagrangian function** [3]:

$$
\begin{aligned}
LR(\boldsymbol{\pi_b}) = \quad & \min \boldsymbol{c}^\intercal \boldsymbol{x} + \boldsymbol{\pi_b}^\intercal (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}) & \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & [\boldsymbol{\pi_d}] \\
& \boldsymbol{x} & \in \mathbb{Z}_+^n \\
= \quad & \boldsymbol{\pi_b}\boldsymbol{b} + \min (\boldsymbol{c} - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\, \boldsymbol{x} & \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} & [\boldsymbol{\pi_d}] \\
& \boldsymbol{x} & \in \mathbb{Z}_+^n
\end{aligned}
$$

Notice that both the Lagrangian subproblem and pricing problem are equivalent, except for a constant offset in their objective functions.

The quality of the lower bound provided by the Lagrangian relaxation depends on the choice of the Lagrangian multipliers ($\forall \boldsymbol{\pi_b} > \boldsymbol{0}$. $LR(\boldsymbol{\pi_b}) \leq z^*_{IP}$ [3]). To find the greatest lower bound, i.e., the optimal Lagrangian multipliers $\boldsymbol{\pi}^*_{\boldsymbol{b}}$ that maximize the Lagrangian subproblem, we solve the **Lagrangian dual problem** (LDP):

$$
z^*_{LDP} = \min_{\boldsymbol{\pi_b} \geq \boldsymbol{0}} LR(\boldsymbol{\pi_b})
$$

If z^*_{IP} is finite, an optimal solution $\boldsymbol{\pi}^*_{\boldsymbol{b}}$ to the LDP provides a bound equal to the optimal objective value of the MP, i.e., $z^*_{IP} = z^*_{MP} = z^*_{LDP}$. Consequently, the optimal Lagrangian multipliers $\boldsymbol{\pi}^*_{\boldsymbol{b}}$ are dual optimal for the MP, and vice versa; optimal dual solutions to the MP are optimal for the LDP [3]. The natural question is whether we can leverage this interplay of primal and dual solutions efficiently. For this, let us consider a simple approach of solving the LDP by approximation, the **subgradient method**.

The Lagrangian function LR is continuous, concave, and subdifferentiable over its finite domain [3]. These properties suggest a hill-climbing approach for finding the optimal Lagrangian multipliers $\boldsymbol{\pi}^*_{\boldsymbol{b}}$: start with some initial guess, and iteratively improve it by moving in the direction of the subgradient of the Lagrangian function. For a given $\boldsymbol{\pi_b} > \boldsymbol{0}$, an optimal solution $\boldsymbol{x}^*$ to $LR(\boldsymbol{\pi_b})$ provides a subgradient $\boldsymbol{g} \coloneqq (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^*)$, representing the violation of the complicating constraints for the Lagrangian function at $\boldsymbol{\pi_b}$ [3]. We then update our current guess of the optimal Lagrangian multipliers $\boldsymbol{\pi}^*_{\boldsymbol{b}}$ by moving in the direction of the subgradient $\boldsymbol{g}$, i.e., $\boldsymbol{\pi}^*_{\boldsymbol{b}} \leftarrow \boldsymbol{\pi}^*_{\boldsymbol{b}} + \alpha \boldsymbol{y}$, where α is a step size. This process is repeated until convergence.

Since a primal optimal solution of IP also yields a dual optimal solution, any IP solver can be viewed as a dual solver for LDP. In the context of a Dantzig-Wolfe reformulation solved by column generation, this becomes particularly interesting, as the pricing problem is equivalent to the Lagrangian subproblem. Column generation can be viewed as a more elaborate update scheme for the Lagrangian multipliers, using multiple solutions to the subproblem to update our guess of the optimal dual values by solving MP [3]. Hence, we could also use the subgradient method to solve

a Dantzig-Wolfe reformulation, solving the *MP* only to ensure we find a solution satisfying the complicating constraints.

Both the subgradient method and column generation face a common issue: the updates of the dual values can overshoot the optimal dual values, leading to oscillation and slow convergence of the dual values [3, 13, 14]. This is undesirable, as it takes longer to find a good lower bound on the *IP*, and similarly as it takes longer to find important columns for the *MP*. To mitigate this issue, sophisticated update schemes have been developed to provide explicit control over updating the dual values in a column generation setting. This **dual value stabilization** can be achieved by smoothing the dual values over the iterations. At iteration t, we determine the smoothed dual values $\widetilde{\boldsymbol{\pi}}^t$ by interpolating the current dual values $\boldsymbol{\pi}^t$ and the previous smoothed dual values $\widetilde{\boldsymbol{\pi}}^{t-1}$:

$$\widetilde{\boldsymbol{\pi}}^t \coloneqq \alpha\boldsymbol{\pi}^t + (1-\alpha)\widetilde{\boldsymbol{\pi}}^{t-1}$$

We can improve upon this by moving from a fixed α to an auto-adaptive α-schedule: decrease α if $\boldsymbol{\pi}^t$ is a good estimate of the optimal dual values, and increase α if it is not [14, 15]. The quality of our guess $\boldsymbol{\pi}^t$ can be assessed using the subgradients $(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^t)$ available from the pricing problem. The angle the subgradient forms with $\boldsymbol{\pi}^t - \widetilde{\boldsymbol{\pi}}^{t-1}$ inversely determines the smoothing coefficient α [14, 15]. This approach requires no parameterization.

Since we are already using the subgradients from the pricing problem, we can also correct the direction of our update. This hybrid approach combines the auto-adaptive α-schedule with the subgradient ascent method. It requires no parameter tuning and only adds a minimal computational effort. This hybrid method can improve the convergence of the dual values for some instances but does not necessarily outperform the auto-adaptive α-schedule for reformulations with identical subproblems. Further details are available in [14, 15].

Chapter 4

SCIP Optimization Suite

The SCIP Optimization Suite is a comprehensive collection of software tools designed to address a wide range of mathematical optimization problems. Central to this suite is the SCIP (Solving Constraint Integer Programs) framework [16, 17], which serves both as a branch-price-and-cut solver and a development platform manly for mixed-integer programming (*MIP*) and constraint integer programming (CIP).

In addition to SCIP itself, the SCIP Optimization Suite includes several other tools that complement its functionality, such as a *LP* solver, a modeling language, and a parallelization layer for exploiting multi-core and distributed computing resources. For further information, we refer the reader to the official SCIP website[1] as well as [18].

4.1 GCG

The Generic Column Generation (GCG) solver developed by Gamrath et al. [11] is a solver implemented using the SCIP framework, specifically designed to implement the Dantzig-Wolfe reformulation and solve optimization problems using column generation and branch-price-and-cut techniques. It works by detecting a suitable decomposition of the problem, reformulating it as a master problem with a set of subproblems using a Dantzig-Wolfe reformulation (see Section 3.2), and then solving the master problem using column generation (see Section 3.1). To solve *MIP*s, GCG utilizes the branch-price-and-cut algorithm (Section 3.5), providing multiple branching strategies, including Vanderbeck's generic branching scheme (see Section 3.5.2.1). Additionally, GCG supports the stabilization of dual values to improve the convergence of the column generation process (see Section 6.3).

[1] https://www.scipopt.org/

Chapter 5

Component Bound Branching

In this chapter, we introduce the **component bound branching rule** (`COMPBND`) for branching on the master variables of the discretized reformulation of any type of bounded *IP*. This branching rule, initially formalized by Desrosiers et al. in [3], builds upon the same fundamental ideas as Vanderbeck's generic branching scheme (Section 3.5.2.1), offering a simpler alternative to branching on component bounds. We will first demonstrate how to enforce component bounds to create a binary search tree. Then, we will explore the algorithm responsible for determining suitable branching decisions. Finally, we will compare and contrast Vanderbeck's generic branching scheme with this new approach, highlighting their similarities and differences.

5.1 Overview of the branching scheme

As discussed in Section 3.5.2, given a fractional master solution $\boldsymbol{\lambda}^*_{RMP}$, we can always find a subset $\emptyset \subset Q' \subset Q \coloneqq \ddot{P}$ such that:

$$\sum_{q \in Q'} \lambda^*_q =: K \notin \mathbb{Z} \tag{5.1}$$

This allows us to eventually enforce the integrality of $\boldsymbol{\lambda}_{MP}$, for example, by adding one of the following branching constraints to each child node:

$$\begin{aligned} \sum_{q \in Q'} \lambda_q &\leq \lfloor K \rfloor \quad [\gamma] \\ \sum_{q \in Q'} \lambda_q &\geq \lceil K \rceil \quad [\gamma] \end{aligned} \tag{5.2}$$

Adding such constraints to the master problem requires us to modify the pricing problem in the following way:

$$
\begin{aligned}
z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \gamma y - \pi_0 \\
\text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\
& y = 1 \Leftrightarrow \boldsymbol{x} \in Q' \\
& \boldsymbol{x} \in \mathbb{Z}^n_+ \\
& y \in \{0, 1\}
\end{aligned}
\tag{5.3}
$$

where y becomes the column entry for the row added to the master, and $y = 1 \Leftrightarrow \boldsymbol{x} \in Q'$ is expressible using a finite set of linear constraints.

To find such a Q' that is expressible in SP, Vanderbeck proposes to use bounds on the components of the columns. Similarly, in our branching scheme, we also find such component bounds. Let us reiterate the notation introduced for Vanderbeck's branching scheme in Section 3.5.2.1:

$$
B \coloneqq (x_i, \eta, v) \in \{x_i \mid 1 \leq i \leq n\} \times \{\leq, \geq\} \times \mathbb{Z} \tag{5.4}
$$

$$
\bar{B} \coloneqq (x_i, \bar{\eta}, v)\,, \bar{\eta} \coloneqq \begin{cases} \leq & \text{if } \eta = \geq \\ \geq & \text{if } \eta = \leq \end{cases} \tag{5.5}
$$

We define a component bound sequence as follows:

$$
S \coloneqq \{(x_{i,1}, \eta_1, v_1)\,, \ldots, (x_{j,k}, \eta_k, v_k)\} \in 2^{\{x_i \mid 1 \leq i \leq n\} \times \{\leq, \geq\} \times \mathbb{Z}} \tag{5.6}
$$

and restrictions of S to only upper bounds $\bar{S}$ and lower bounds $\underline{S}$ respectively:

$$
\begin{aligned}
\bar{S} &\coloneqq \{(x_i, \leq, v) \mid (x_i, \leq, v) \in S\} \\
\underline{S} &\coloneqq \{(x_i, \geq, v) \mid (x_i, \geq, v) \in S\}
\end{aligned}
\tag{5.7}
$$

We continue using the following shorthand notation:

$$
\eta(a, v) \Leftrightarrow \begin{cases} a \leq v & \text{if } \eta = \leq \\ a \geq v & \text{if } \eta = \geq \end{cases} \tag{5.8}
$$

Similar to Vanderbeck's branching, we can find such a subset Q' by finding a component bound sequence S such that:

$$
\sum_{q \in Q(S)} \lambda^*_q =: K \notin \mathbb{Z} \tag{5.9}
$$

where $Q(S) \coloneqq \{q \in Q \mid \forall (x_i, \eta, v) \in S.\eta(x_{qi}, v)\}$.

Proof 3.5.2.1 shows that such an S always exists if the master solution is not integral. After obtaining such an S, we create two child nodes, the down- and up-branches, by first adding the branching decision to the master problem:

$$\sum_{q \in Q(S)} \lambda_q \leq \lfloor K \rfloor \quad [\gamma_\downarrow \leq 0] \qquad \Bigg| \qquad \sum_{q \in Q(S)} \lambda_q \geq \lceil K \rceil \quad [\gamma_\uparrow \geq 0] \tag{5.10}$$

We now must ensure that newly priced columns $x_{q'}$ are assigned a coefficient of $y = 1$ for the branching decision if $q' \in Q(S)$, i.e., if $\forall (x_i, \eta, v) \in S.\eta(x_{q'i}, v)$ and otherwise $y = 0$. We achieve this by introducing additional binary variables $\bar{y}_s, \underline{y}_{s'}$ for each $B_s \in \bar{S}$ and for each $B_{s'} \in \underline{S}$ respectively, along with the following constraints, in the *SP* [3]:

$$\begin{aligned}
y = 1 \Leftrightarrow \sum_{B_s \in \bar{S}} \bar{y}_s + \sum_{B_s \in \underline{S}} \underline{y}_s = |S| & \\
\bar{y}_s = 1 \Leftrightarrow x_s \leq v_s & \quad \forall B_s \in \bar{S} \\
\underline{y}_s = 1 \Leftrightarrow x_s \geq v_s & \quad \forall B_s \in \underline{S} \\
y \in \{0, 1\} & \\
\bar{y}_s \in \{0, 1\} & \quad \forall B_s \in \bar{S} \\
\underline{y}_s \in \{0, 1\} & \quad \forall B_s \in \underline{S}
\end{aligned} \tag{5.11}$$

What remains is to express all logical equivalences using a finite set of linear constraints. For this, the following observations are crucial [3]:

- In the down branch, since $-\gamma_\downarrow \geq 0$, y naturally takes the value 0 and so do all $\bar{y}_s$ and $\underline{y}_{s'}$. Thus, in the down branch, we need to force all $\bar{y}_s$ and $\underline{y}_{s'}$ to 1 if the corresponding component bounds are satisfied, and force y to 1 if all $\bar{y}_s$ and $\underline{y}_{s'}$ equal 1.
- In the up branch, the opposite is the case: since $-\gamma_\uparrow \leq 0$, y and all $\bar{y}_s, \underline{y}_{s'}$ naturally take the value 1, requiring us to force all $\bar{y}_s$ and $\underline{y}_{s'}$ to 0 if their corresponding component bounds are not satisfied, and force y to 0 if any of the $\bar{y}_s, \underline{y}_{s'}$ equals 0.

Given that we require a bounded *IP* to begin with, let us denote the lower and upper bounds of a variable x_i as lb_i and ub_i respectively. Using the above observations, we can now express the logical equivalences mandated by the branching decision as follows [3]:

$$\begin{aligned}
y \geq 1 + \sum_{B_s \in \bar{S}} \bar{y}_s + \sum_{B_s \in \underline{S}} \underline{y}_s - |S| & \qquad \Bigg| \qquad y \leq \bar{y}_s & \quad \forall B_s \in \bar{S} \\
\bar{y}_s \geq \frac{(v_s + 1) - x_i}{(v_s + 1) - \mathrm{lb}_i} & \quad \forall B_s \in \bar{S} & y \leq \underline{y}_s & \quad \forall B_s \in \underline{S} \\
\underline{y}_s \geq \frac{x_i - (v_s - 1)}{\mathrm{ub}_i - (v_s - 1)} & \quad \forall B_s \in \underline{S} & \bar{y}_s \leq \frac{\mathrm{ub}_i - x_i}{\mathrm{ub}_i - v_s} & \quad \forall B_s \in \bar{S} \\
& & \underline{y}_s \leq \frac{x_i - \mathrm{lb}_i}{v_s - \mathrm{lb}_i} & \quad \forall B_s \in \underline{S}
\end{aligned} \tag{5.12}$$

We have now successfully defined the branching decision in the master problem and the corresponding constraints in the pricing problem. Until we find an optimal integral solution of master variables, we will continue to branch using a suitable component bound sequence S, creating a binary search tree. In the next section, we present an algorithm responsible for finding such an S given a fractional master solution $\boldsymbol{\lambda}^*_{RMP}$.

5.2 Separation Procedure

Definition 5.1. *The **fractionality of** $\boldsymbol{\lambda}^*_{RMP}$ **with respect to** S is given by:*

$$F_S = \sum_{q \in \mathcal{Q}(S)} \left(\lambda^*_q - \lfloor \lambda^*_q \rfloor\right) \geq 0 \tag{5.13}$$

When $S = \emptyset$, we have $Q(S) = Q$, and thus $F_S > 0$ since at least one λ^*_q is fractional. In this case, $F_S \in \mathbb{Z}_+ \backslash \{0\}$ due to the convexity constraint $\sum_{q \in Q} \lambda_q = 1$ in the *MP* (analogous in aggregated subproblems, see Section 3.4).

In general, for any S one of three cases can occur:

- $F_S = 0$: $Q(S)$ contains no column with fractional λ^*_q. Thus, branching on S would not cut off the current fractional solution $\boldsymbol{\lambda}^*_{RMP}$. Adding further component bounds to S would not change this.

- $a < F_S < a + 1, a \in \mathbb{Z}_+$. Using Equation (5.13), we can rewrite this as:

$$\sum_{q \in \mathcal{Q}(S)} \lfloor \lambda^*_q \rfloor < \sum_{q \in \mathcal{Q}(S)} \lambda^*_q < \sum_{q \in \mathcal{Q}(S)} \lfloor \lambda^*_q \rfloor + 1 \tag{5.14}$$

  The sum $\sum_{q \in \mathcal{Q}(S)} \lambda^*_q =: K$ is fractional, enabling us to branch on S (see Equation (5.1)).

- $F_S \in \mathbb{Z}_+ \backslash \{0\}$. In this case, $\sum_{q \in \mathcal{Q}(S)} \lambda^*_q \in \mathbb{Z}_+$, and therefore branching on S would not cut off the current fractional solution. However, using 3.1, we can find two distinct columns $q_1, q_2 \in Q(S)$, i.e., where $x_{i,q_1} < x_{i,q_2}$ for some $i \in \{1, \ldots, n\}$, such that $\lambda^*_{q_1}$ and $\lambda^*_{q_2}$ are fractional. Denote the rounded median of these two column entries as $v \coloneqq \lfloor \frac{x_{i,q_1} + x_{i,q_2}}{2} \rfloor$. Since $x_{i,q_1} \leq v < v + 1 \leq x_{i,q_2}$, we can separate q_1 from q_2 by imposing a bound on the component x_i, i.e., expand S to either S_1 or S_2, where:

$$\begin{aligned} S_1 &\coloneqq S \cup \{(x_i, \leq, v)\} \\ S_2 &\coloneqq S \cup \{(x_i, \geq, v + 1)\} \end{aligned} \tag{5.15}$$

  Note that $F_S = F_{S_1} + F_{S_2}$, thus we can always at least halve the fractionality of the current solution. Furthermore, both $Q(S_1)$ and $Q(S_2)$ are guaranteed to contain at least one fractional column, ensuring $F_{S_1}, F_{S_2} > 0$.

These observations suggest the following separation procedure: initialize $S^0 = \emptyset$, i.e., $F_{S^0} > 0$. While $F_{S^k} \in \mathbb{Z}_+ \backslash \{0\}$, find a component bound x_i to branch on, yielding S_1 and S_2. Proceed with either as S^{k+1}. Finally, F_{S^k} will be fractional, and we can branch on S^k [3].

Proposition 5.1. *At no iteration $k \geq 0$ will the separation procedure produce a component bound sequence S^k with $F_{S^k} = 0$.*

Proof. As previously discussed, $F_\emptyset > 0$, i.e., S^0 satisfies the proposition.

Assume S^k satisfies the proposition, i.e., $F_{S^k} > 0$. If $F_{S^k} \notin \mathbb{Z}_+$, the procedure terminates, and the proposition holds. Else $F_{S^k} \notin \mathbb{Z}_+ \backslash \{0\}$. In this case, let us assume $F_{S^{k+1}} = 0$. Then $Q(S^{k+1})$ contains no fractional columns, which contradicts the design of S^{k+1}. By contradiction, $F_{S^{k+1}} > 0$ must hold, and by induction, the proposition holds. □

Proposition 5.2. *Given that $\boldsymbol{\lambda}^*_{RMP}$ contains finitely many non-zero values, the separation procedure will terminate after a finite number of iterations.*

Proof. Let us denote the restriction of $Q(S)$ to the columns q with fractional λ^*_q as $Q_f(S)$. By our assumption $|Q_f(S)| < \infty$. At each iteration k, we only remove columns from $Q_f(S^k)$, i.e., $\left|Q_f(S^{k+1})\right| < \left|Q_f(S^k)\right|$. Since $|Q_f(S^0)| < \infty$, the separation procedure must terminate after a finite number of iterations. □

5.2.1 Choice of Component Bounds

The separation procedure described above is not complete, as we have not yet defined which bounds we impose on which components. This choice can significantly impact the performance of the subsequent solving of the child nodes. In the worst case, the separation procedure will yield a component bound sequence S for which $Q(S)$ only contains one column, i.e., dichotomous branching. Maintaining balance within the tree is generally beneficial, but the time required to find an optimal S can grow arbitrarily large and must be traded off against improved performance that comes with a balanced tree. We propose the following two-staged approach:

In the first stage, using one or multiple heuristics, we recursively determine a set of valid component bound sequences $S_1, \ldots, S_m$ for the current fractional master solution $\boldsymbol{\lambda}^*_{RMP}$. For this, we adapt the previously described separation procedure to explore both options S_1 and S_2 whenever F_S is integral. A first-stage heuristic is now only responsible for finding a separating component x_i and bound value $v \in \mathbb{Z}$. Both the lower bound $x_i \leq v$ and the upper bound $x_i \geq v + 1$ will be explored further; we do not have to choose between them at this stage. In particular, we propose two heuristics for this first stage:

- `MaxRangeMidrange` Heuristic: At each iteration k, we choose the component x_i for which the components $x_{i,q}$ of the columns $q \in Q_f(S^k)$ are most spread out. We then bound x_i by the midrange of these components. Formally, we define:

$$\begin{aligned} max_j &\coloneqq \arg\max_{q \in Q_f(S^k)} x_{j,q} && \forall j \in \{1, \dots, n\} \\ min_j &\coloneqq \arg\min_{q \in Q_f(S^k)} x_{j,q} && \forall j \in \{1, \dots, n\} \\ x_i &= \arg\max_{j \in \{1,\dots,n\}} max_j - min_j \\ v &\coloneqq \frac{max_i - min_i}{2} \end{aligned}$$

- `MostDistinctMedian` Heuristic: At each iteration k, we choose the component x_i for which the components $x_{i,q}$ of the columns $q \in Q_f(S^k)$ have the most distinct values. We then choose v to be the median of these components.

In the second stage, another heuristic now scores every component bound sequence, and we continue branching using the highest scoring S_j. Specifically, we propose to choose the smallest component bound sequence, i.e., $S_j = \arg\min_{S_1,\dots,S_m} |S_j|$. This minimizes the modifications we make to the pricing problem. In case there are multiple such minimal component bound sequences, we propose to use one of two further heuristics to select one component bound sequence out of all those with minimal cardinality:

- `ClosedToZHalf` Heuristic: For each S_j, we calculate $K_j \coloneqq \sum_{q \in Q(S_j)} \lambda_q^*$. We then select the one where K_j is closest to $\frac{Z}{2}$. Here, Z denotes the number of aggregated subproblems in the current block (see Section 3.4 and Section 5.2.3). The intuition behind this is once again to maintain balance within the tree: if K_j was far off from $\frac{Z}{2}$, i.e., either close to 0 or close to Z, branching on S_j would be similar to dichotomous branching, since it would either forbid almost all or almost no solutions (see Section 3.5.2).

- `MostFractional` Heuristic: Here, we also calculate K_j for each S_j. We then select the one where K_j is most fractional, i.e., where $K_j - \lfloor K_j \rfloor$ is closest to 0.5. This heuristic is motivated by the idea that such a most fractional selection of master variables can be interpreted as the *RMP* being most uncertain about.

5.2.2 Post-processing of Component Bound Sequences

Depending on the heuristics chosen, there is no guarantee that the separation procedure will find a component bound sequence S in which each component

x_i has at most one upper bound (lower bound analogous). While this is not a problem from a mathematical standpoint, only the least upper bound (greatest lower bound, respectively) is relevant, and so adding variables and constraints for the other upper bounds (lower bounds) is unnecessary and could potentially slow down the solving process of *SP*. Therefore, post-processing of the component bound sequences, i.e., removing redundant bounds, is advisable. For example, if we had $S = \{(x_1, \leq, 3), (x_1, \leq, 4)\}$, the first bound already implies the second, and we can remove the second bound from S, yielding $\{(x_1, \leq, 3)\}$.

5.2.3 Branching with Multiple Subproblems

The component bound branching rule described above can be applied to instances with a single subproblem, as well as instances with multiple identical subproblems aggregated into a single subproblem (see Section 3.4). However, there are instances consisting of at least two distinct subproblems, also known as blocks, where the master problem yields a solution $\boldsymbol{\lambda}^{k*}_{RMP}$ for each block k. Since each component x_i belongs to a specific block, not all columns q_1, q_2 in *RMP* will have an entry for x_i, thus the separation scheme is not directly applicable across multiple blocks.

Given that more than one block has fractional master solutions, we propose to pick one of those blocks to branch on and then apply the separation procedure as described above within the selected block.

5.3 Comparison to Vanderbeck's Generic Branching

Both the generic branching scheme by Vanderbeck (Section 3.5.2.1) and the proposed component bound scheme involve imposing bounds on the components of the columns within the *SP* to branch in the master problem. However, the methods for enforcing these component bounds differ significantly.

Vanderbeck's `GENERIC` branching scheme treats these bounds as hard constraints, effectively subdividing the solution space in the subproblem. As a result, when the optimal solution $\boldsymbol{x}^*$ to the *IP* is found in a node of the *RMP*, it adheres to all component bounds imposed by the branching decisions from the root to that node. In contrast, our component bound branching rule introduces these bounds as soft constraints, allowing the generation of columns that both satisfy and violate the component bounds.

While the component bound branching rule might be simpler to implement, Vanderbeck's `GENERIC` branching offers a notable advantage: it only requires tightening the bounds of the components in the *SP*, without introducing new

variables and constraints. This simplicity maintains the structure of the pricing problem, enabling many dynamic programming solvers for specific *IP*s to efficiently generate columns despite changing variable bounds. Conversely, our approach alters the pricing problem with each branching decision, potentially necessitating the use of a generic *MIP* solver. Moreover, the `GENERIC` scheme incrementally tightens the bounds as the search tree deepens, making the pricing problem progressively easier to solve. In contrast, `COMPBND` branching adds more variables and constraints, complicating the *SP* as the branching process continues.

Chapter 6

Master Constraints without corresponding Original Problem Constraints

6.1 Conceptual Framework and Definition

Beyond implementing the component bound branching rule (Chapter 5), key objective of this thesis is to enable future GCG developers to easily create new branching rules and separators operating within the reformulation inside the framework (Section 4.1). Currently, GCG faces limitations in this regard: branching rules must either produce decisions formulated in the original problem, which can then be Dantzig-Wolfe reformulated and added to the master and pricing problems (as seen when branching on original variables in Section 3.5.1), or they must produce constraints for the master problem without requiring modifications to the pricing problem, as seen with Ryan-Foster branching. Other branching rules, such as Vanderbeck's generic branching scheme (Section 3.5.2.1), cannot be implemented without significant changes to the GCG framework. These changes would involve, for example, applying and removing component bounds in the pricing problem when a node in the search tree is entered or left. Our proposed component bound branching rule (Chapter 5) and any separators using the master problem (Section 3.6.2) would also require such changes. The reason is that GCG does not currently support imposing constraints in the master problem that necessitate modifications to at least one SP, where the master constraints and induced pricing problem modifications cannot necessarily be described as a product of a Dantzig-Wolfe reformulation, i.e., do not necessarily have a counterpart in the original formulation.

In this chapter, we will specify the notation of such constraints, referred to as **generic mastercuts**. We will present our integration of these constraints into the

GCG framework as part of a new interface and demonstrate how to apply dual value stabilization to these constraints.

First, let us define the concept of a generic mastercut, which unites Vanderbeck's generic branching scheme, our component bound branching rule, and any master separators.

Definition 6.1. *A **generic mastercut** is a constraint in the master problem that does not have a counterpart in the original problem, and therefore requires modification to one or multiple SP to ensure its validity in the master. Specifically, it takes the following form, where the implicit function f maps columns p and r to their respective coefficients in the master constraint:*

$$\sum_{p \in P} f(p)\lambda_p + \sum_{r \in R} f(r)\lambda_r \leq h \quad [\gamma]$$

The subproblems are now responsible for correctly determining the coefficients $f(p)$ and $f(r)$ of all newly generated columns p and r. Therefore, one generic mastercut is associated with a set of pricing modifications, one for each subproblem that the constraint in the master affects.

Definition 6.2. *A **pricing modification** to the subproblem SP^k in block k, associated with a generic mastercut with dual value γ, is a set of constraints and variables added to the subproblem to ensure the validity of the generic mastercut in the master problem with respect to new variables.*

*Every pricing modification includes at least one mandatory variable $y \in Y$ of some domain Y (e.g. $Y = \mathbb{Z}_+$) with an objective coefficient of $-\gamma$ in the SP^k. The solution value of y is used as the column entry for the master constraint of the generic mastercut, i.e., $f(p)$ or $f(r)$. For this reason, this variable is known as the **coefficient variable** of the pricing modification, and we modify the pricing problem as follows:*

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi_b}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \gamma y - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & y = f(\boldsymbol{x}) \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \\ & y \in Y \end{aligned}$$

Expressing $y = f(\boldsymbol{x})$ may require auxiliary variables and constraints. Due to their auxiliary role, these variables have an objective coefficient of zero and do not correspond to a row in the master problem.

This generic mastercut construct can be used by both the branching rules `GENERIC` (Section 3.5.2.1) and the `COMPBND` (Chapter 5): in both, we choose f to

be an indicator function that equals one if and only if the column in question fully satisfies a given component bound sequence S. Thus, y would be a binary decision variable ($Y = \{0, 1\}$).

In Vanderbeck's generic branching scheme, the pricing problems are only permitted to generate columns in the region defined by S. We enforce this, by creating auxiliary constraints $x_i \eta_i n_i$ for each $(x_{*,i}, \eta_i, n_i) \in S$, and forcing $y = 1 (= f(\boldsymbol{x}))$ (or $Y = \{1\}$), though y could be omitted altogether through presolving. In contrast, `COMPBND` additionally allows columns violating S to be generated, thus we create auxiliary variables and constraints to determine whether all bounds in S were satisfied, as discussed in Section 5.1.

The benefit of this construct is its generality, and thus its versatility. It does not presume anything about the origin of a generic mastercut. For this reason, it is not only applicable to branching rules, but, for example, also to separators. In Section 3.6.2, we have briefly discussed the possibility of separating a fractional master solution by finding cutting planes solely based on the Dantzig-Wolfe reformulation. Cuts found by such a master separator would exactly fit the definition of a generic mastercut.

In the following, we will see that a technical issue affecting the validity of the constraints arises when generic mastercuts are created locally in non-root nodes. We will briefly present how the current implementation of Vanderbeck's generic branching scheme in `GCG` deals with this issue, and then introduce our solution that is correct, more efficient, and more broadly applicable. Finally, we will discuss how we can continue using dual value stabilization in `GCG` with generic mastercuts.

6.2 Mastervariable Synchronization across the entire Search Tree

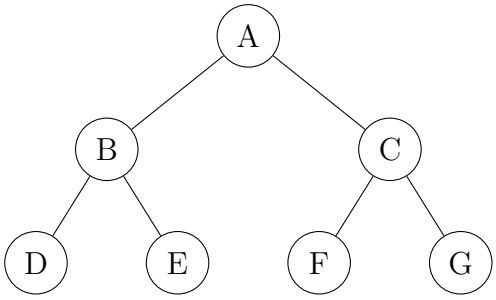


Figure 6.1: An exemplary search tree created by the component bound branching rule, where the lexicographic order of the nodes resembles the order in which they were created.

As we have mentioned multiple times, all columns in the *RMP* must have the correct coefficient set in the master constraint. Just one column with an incorrect coefficient can lead to invalid mastercuts. For example, if a column q satisfies a component bound sequence S, it should have a coefficient of 1 in the mastercut. Any other coefficient would lead to an incomplete branching scheme. This very same reason is why the pricing modifications of a generic mastercut are essential in the first place.

So, we must ensure that all columns in the *RMP* have the correct coefficients set. This can be easily achieved when creating the generic mastercut, as well as when a new column is generated in the subtree of the node where the generic mastercut was created. In the former case, we may simply compute the coefficients of all variables in the *RMP* upfront. And in the later case, the solution value of the coefficient variable in the *SP* already determines the correct coefficient in the master. However, consider the following scenario in a search tree, for example a tree generated with the component bound branching rule using generic mastercuts, as depicted in Figure 6.1: we are currently processing node F in the search tree and generate a new column q'. After deactivating node F and activating node D, it is possible that $x_{q'}$ satisfies the component bounds imposed in D. Thus, q' should have a coefficient of 1 in the mastercut of D. However, since the column was generated in F, the information that q' was created was not communicated to D. And therefore, the coefficient of q' in the mastercut of D is not set correctly. This problem also occurs for cuts produced by master separators.

To prevent this, all generic mastercuts must be made aware of these columns to update their coefficients accordingly. Specifically, we want to synchronize newly generated master variables across the entire search tree lazily, i.e., only when a node is activated and thus the update is required. Moreover, since `GCG` can remove columns it deems unnecessary from the *MP*, the synchronization must consider the case where a newly generated column is deleted before it is fully synchronized across the search tree.

In this section, we will first analyze the current approach taken by the implementation of Vanderbeck's generic branching in `GCG`. Then, we will present a more efficient approach, which we refer to as **history tracking**, and further improve it.

6.2.1 Current Approach used by the Implementation of Vanderbeck's Generic Branching

In the current implementation of Vanderbeck's generic branching in `GCG`, each node created by this branching rule stores the number of master variables it is aware of. This number is updated whenever a node is deactivated to reflect any newly generated columns. Upon node activation, the current number of variables in the

master is compared against how many variables were present the last time the node was active. If new columns have been added to the *RMP* in the meantime, this counter might increase. If so, the coefficient for the new columns will be determined and set in the *RMP*. More specifically, since the number of variables in the master can grow quite large, it avoids updating the coefficients of all columns in the *RMP*. Instead, it assumes the master variables indexed from the last known number of variables to the current number of variables are new and sets their coefficients accordingly.

Unfortunately, for any generic mastercut in general, this approach is not sufficient, as not all new columns are necessarily detected. For instance, if one column was generated and another column was deleted in the meantime, the counter would not increase, as the number of columns in the master would remain the same. Consequently, the coefficient for the new column would not be set in the mastercut, potentially leading to invalid mastercuts.

Additionally, this approach is not developer-friendly. Developers of branching rules and master separators should not have to manage when and where columns are generated and deleted. This is a task that should be handled by `GCG`, abstracting away the origins of the columns.

6.2.2 History Tracking Approach

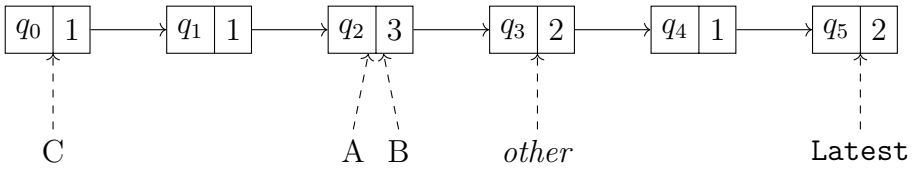


Figure 6.2: Reference-counted linked list of the history of columns added to the *RMP*, with external references drawn dashed from below, e.g., those from the search tree nodes A, B, and C. Each element holds a reference to the master variable belonging to column q_i, as well as the number of references to itself.

We propose an efficient approach to lazily notify all nodes in the search tree upon node activation of new columns generated while also considering deleted columns. We introduce a reference-counted linked list of variables added to the *RMP*, where the order of the variables in the list is determined by their generation order. Each node in the search tree holds its own external reference to this construct. The specific element in the list that a search tree node points to indicates the last column in the *RMP* when the node was last active. All subsequent variables, i.e., the elements next in the list, are new columns generated elsewhere in the tree.

Additionally, we hold one external reference to the tail of the list, representing the last generated column. This construct is illustrated in Figure 6.2. Since the linked list tracks which variables were created when we will refer to this list as the `varhistory`.

Let us consider a search tree with root node A and child nodes B and C. Currently, we are solving node B, and therefore nodes A and B are active. While solving B, we have already generated columns q_3, q_4, and q_5. Assume we have solved the relaxation of B to optimality, finding a fractional solution, and have created two child nodes D and E. Both nodes will be created using all columns currently in the *RMP*, i.e., $q_i, i \in \{0, 5\}$. For this reason, we use the `Latest` pointer to initialize the reference to the `varhistory` of D and E (Figure 6.3).

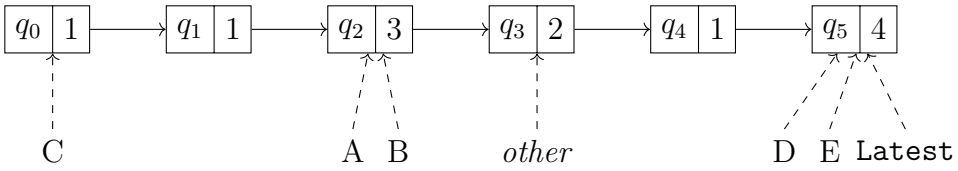


Figure 6.3: `varhistory` after creating child nodes D and E of node B.

Continuing this scenario, let `GCG` deem the column q_2 unnecessary and remove it from the *RMP*. Since there may be external references to this variable, which in this case there are, we do not remove the element in the list holding q_2. Instead, we mark it as deleted. Next, we would like to solve node C. For this, we must deactivate node B, and activate node C. Whenever we deactivate a node, we know that it and all its ancestors are already aware of all columns in the *RMP*. Therefore, we may jump all the active node's pointers to the `Latest` pointer. This is illustrated in Figure 6.4.

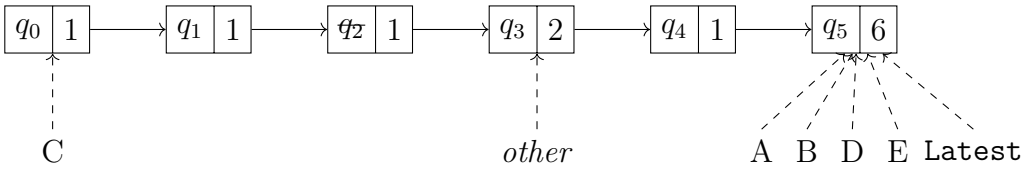


Figure 6.4: `varhistory` after deletion of column q_2 and deactivation of node B.

Finally, we can activate node C. Upon node activation, we realize that the element that C points to in the `varhistory` has a next element. This means that there are new columns that have been generated since the last time C was active. We forward the pointer of C one by one until we reach the `Latest` pointer. Each time we forward the pointer, if the variable q_i has not been marked as deleted, we calculate the coefficient of q_i in the generic mastercut of C.

Whenever we forward a pointer, either step-by-step or by jumping to the `Latest` pointer, the internal reference count of the elements in the list is updated. As soon as this reference count reaches zero, the element will be safely freed. This ensures that only necessary variables, i.e., those that still need to be synchronized across the entire search tree, are kept in memory. This is illustrated in Figure 6.5.

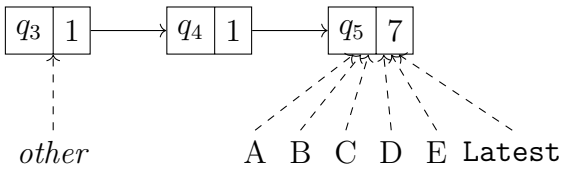


Figure 6.5: `varhistory` after activation of node C.

Assume node C generates a new column q_6. We add this column to the `varhistory` by allocating a new list element, setting its reference count to 1, setting the next pointer of the current `Latest` element to the new element, and finally forwarding the `Latest` pointer to the new element. This is illustrated in Figure 6.6.

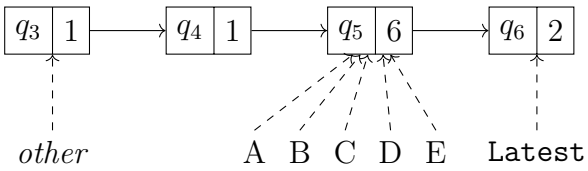


Figure 6.6: `varhistory` after generating column q_6 in node C.

This approach is correct in the sense that all active generic mastercuts are guaranteed to be aware of all columns in the *RMP*. This correctness is given, since the deletion of variables can not cause us to miss any new variables. Since close to no management is required, its performance impact is negligible: we must only update reference counts, forward pointers, and append and free list elements. Furthermore, we hold the memory footprint to a minimum, as the `varhistory` construct will only hold variables that still need to be synchronized. But once all external references have seen some variable q_i, i.e., its reference count reaches zero, we can automatically free the memory of the element in the list holding q_i.

And as a final note, this approach is not limited for synchronization of master variables for generic mastercuts used as branching decisions, but can also be used for other purposes, which have symbolized by the *other* reference in the above figures. Such other purposes, for example would be keeping cutting planes generated by master separators up-to-date (Section 3.6.2).

6.2.3 History Tracking using Unrolled Linked Lists Approach

While already efficient, the previous approach has an opportunity for improvement: the elements of the `varhistory` might be allocated in completely different memory locations, leading to poor cache utilization during traversal. Storing the entire list in a contiguous memory block could improve cache locality but could be costly if reallocation and copying are needed when space runs out.

We can improve cache utilization by unrolling the linked list into blocks storing a fixed number of columns, with each block having its own reference count. These blocks are linked together, forming a list of blocks. The references to the `varhistory` point to the blocks and hold an offset within the block. This way, each reference still refers to some unique column q_i, retaining the ability to forward a pointer one column at a time. A new variable is added to the tail block if its capacity isn't maxed out; otherwise, a new block is allocated. This concept is illustrated in Figure 6.7.

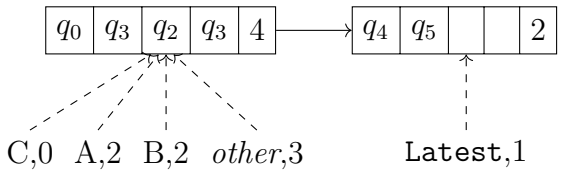


Figure 6.7: `varhistory` of Figure 6.2 unrolled into blocks of size 4.

This approach improves cache locality and reduces the total number of memory allocation and deallocation operations. From an outside perspective, the fundamental operations of the `varhistory` construct remain the same.

6.3 Dual Value Stabilization for Generic Mastercuts

As discussed in Section 3.7, the hybrid ascent dual value stabilization method implemented in `GCG` requires subgradients of the Lagrangian function to update the dual values. These subgradients are precisely the violations of the complicating constraints in the pricing problem, represented by $(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^*)$ for a given pricing solution $\boldsymbol{x}^*$. Here, $\boldsymbol{A}\boldsymbol{x}^*$ represents the column coefficients for the constraints in the master problem.

To stabilize the dual value γ of a generic mastercut (Definition 6.1), we use the same principle. The violation of a generic mastercut is given by $(f - y^*)$. Utilizing

this violation as a subgradient aligns with the idea of refining our estimate of the optimal dual values by moving in the direction indicated by this violation. Thus, by determining the violation $(f - y^*)$, we can apply the existing hybrid ascent dual value stabilization method to generic mastercuts effectively.

Chapter 7

Implementation

7.1 Generic Mastercuts

In this thesis, we have extended the architecture of GCG to manage generic mastercuts (Chapter 6) that developers of branching rules or master separators might want to add to the solver. We have created an interface for interacting with such mastercuts. Specifically, we define a generic mastercut as a wrapper around a constraint in the master, as well as variables and constraints to be added to a specific pricing problem:

Listing 7.1: Generic Mastercut Data Structure

```
1  struct GCG_PRICINGMODIFICATION {
2     int                   blocknr;
3     SCIP_VAR*             coefvar;
4     SCIP_VAR**            additionalvars;
5     int                   nadditionalvars;
6     SCIP_CONS**           additionalconss;
7     int                   nadditionalconss;
8  };
9
10 enum GCG_MASTERCUTTYPE {
11    GCG_MASTERCUTTYPE_CONS,
12    GCG_MASTERCUTTYPE_ROW
13 };
14
15 union GCG_MASTERCUTCUT {
16    SCIP_CONS*            cons;
17    SCIP_ROW*             row;
18 };
```

```
19
20 #define GCG_DECL_MASTERCUTGETCOEFF(x) SCIP_RETCODE x
      (SCIP* scip, GCG_MASTERCUTDATA* mastercutdata,
      SCIP_VAR** solvars, SCIP_Real* solvals, int
      nsolvars, int probnr, SCIP_Real* coef)
21
22 struct GCG_MASTERCUTDATA {
23    GCG_MASTERCUTTYPE     type;
24    GCG_MASTERCUTCUT      cut;
25    GCG_PRICINGMODIFICATION* pricingmodifications;
26    int                   npricingmodifications;
27    void*                 data;
28    GCG_DECL_MASTERCUTGETCOEFF ((*mastercutGetCoeff));
29 };
```

This setup closely aligns with our definition of a generic mastercut, in that one master constraint is associated with at least one pricing modification. Each pricing modification adds variables and constraints to the subproblem SP^k associated with block k. Two more considerations were made here:

First, we allow the mastercut to be either a constraint or a row in the master. In SCIP, separators create cutting planes as rows that can be dynamically added and removed from the problem. Constraints, on the other hand, are typically considered part of the formulation at a specific node. Since our goal is for master separators to use this interface, we permit both types of cuts in the master.

Second, deep within GCG's pricing loop, the row coefficients for the master variables are often recalculated. In such a situation, we could, of course, calculate the mastercut coefficient of a master variable by solving the pricing problem while fixing all original pricing variables $\boldsymbol{x}$ to find the solution value of the coefficient variable y. However, this would impose a significant computational overhead. Instead, for each generic mastercut, we pass along a callback function that calculates the coefficient of a master variable in the mastercut given the pricing solution $\boldsymbol{x}^*$. As this function might require external data, we allow the user to store a pointer to such data in the `data` field. For example, in the `COMPBND` branching rule (Chapter 5), the function stored in `mastercutGetCoeff` would return 1.0 if the column satisfies the component bound sequence S pointed to by `data`, and 0.0 otherwise.

To the outside, we expose methods for creating and freeing a generic mastercut. Within GCG we have adapted the stabilization to handle generic mastercuts according to Section 6.3. Notably, we have modified the pricing loop to apply and remove the pricing modifications of each generic mastercut before and after the pricing problem is solved.

The only remaining part is to make GCG aware that such generic mastercuts have been added to the problem. It would be incorrect to apply the modifications of all generated mastercuts to the pricing problem. For instance, a generic mastercut used as a branching constraint in the left subtree is not necessarily valid in the right subtree. Even if a generic mastercut is valid, it might not be active in the model if SCIP believes the cutting plane to be unnecessary. Thus, we require a method to determine the set of currently active generic mastercuts. To facilitate such mastercuts used for branching, we have extended the existing branching interface of GCG by a callback that simply returns the generic mastercut of the current node, if any. By traversing the tree from the root to the current node, we can collect all active generic mastercuts.

7.2 Mastervariable Synchronization

As discussed in Section 6.2, there is a need to synchronize the information of newly generated columns across the entire search tree. This requirement arose in the context of branching using generic mastercuts, but it is generally necessary for any branching rule that does not formulate its decisions in the original problem. Therefore, we have decoupled the synchronization of master variables from the generic mastercut interface and implemented it as a separate internal module within GCG.

Listing 7.2: Variable History Construct

```
1  struct GCG_VARHISTORYBUFFER {
2     SCIP_VAR*              vars[50];
3     int                    nvars;
4     GCG_VARHISTORYBUFFER*  next;
5     int                    nuses;
6  };
7
8  struct GCG_VARHISTORY {
9     GCG_VARHISTORYBUFFER*  buffer;
10    int                    pos;
11 };
```

To enable such mastervariable synchronization, we have implemented the history tracking approach using unrolled linked lists as described in Section 6.2.3. Each element, or buffer, in the unrolled linked list has a default capacity of 50 variables. The variable nuses, which acts as a reference count, keeps track of how many strong pointers of type GCG_VARHISTORY are currently pointing to the buffer. The GCG_VARHISTORY structure is a simple wrapper around the buffer, keeping track of

the current position in the buffer. Consequently, each strong pointer still points to a specific variable. The GCG pricer is responsible for managing the global variable history list and appending new variables whenever a new column is generated. In Section 6.2, we have denoted this central reference as the Latest pointer.

We can now synchronize master variables by attaching a strong reference to each node in the search tree. Specifically, upon node creation, we create a reference identical to the Latest pointer stored in the GCG pricer. Then, following the procedure described in Section 6.2.2, we forward these strong pointers upon node (de-)activation. To inform a branching rule in a specific node about the creation of a new variable, we have extended the GCG branching rule interface with the following callback function. The branching rule can then, for example, determine the constraint coefficient for this new master variable.

Listing 7.3: Branching Rule Interface Extension

```
1 #define GCG_DECL_BRANCHNEWCOL(x) SCIP_RETCODE x (SCIP*
     scip, GCG_BRANCHDATA* branchdata, SCIP_VAR*
     mastervar)
```

Finally, we note that each SCIP_VAR already contains a flag indicating whether it has been deleted from the problem. We use this flag instead of marking such variables as deleted in the history buffer ourselves.

7.3 Component Bound Branching

We have implemented the component bound branching rule as detailed in Chapter 5 within GCG. This rule necessitates the addition of constraints and variables to the subproblem to enforce its branching decisions, which we have facilitated using the new generic mastercut interface.

The integration of the component bound branching rule into the GCG framework was straightforward due to the generic mastercut interface. This interface allows us to efficiently manage the necessary modifications to the subproblem, ensuring that the branching decisions are correctly enforced.

The component bound branching rule is now fully integrated into the GCG framework, and can be used seamlessly alongside other branching rules.

Chapter 8

Evaluation

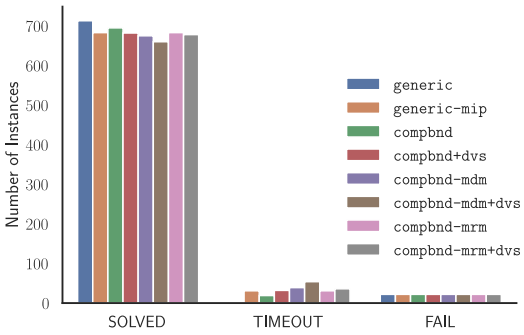
8.1 Test Set of Instances

To evaluate the component bound branching rule and compare it to Vanderbeck's generic branching scheme, we assembled a test set of mixed integer instances. We focused on instances where `GCG` itself chooses the generic branching scheme or the component bound branching rule, ensuring our evaluation is not skewed by forcing such branching on instances.

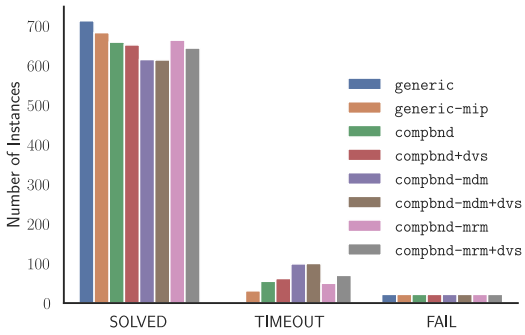
To create this test set, we filtered the `strIPlib` collection [19] for instances previously solved using the generic branching scheme, resulting in an initial set of 1053 mixed integer instances. However, the metadata for these instances was generated with an older version of `GCG`, which may not reflect the current solver's behavior. Consequently, we post-processed these instances, removing those no longer solved using the generic branching scheme. This refinement yielded a final test set of 736 mixed integer instances, primarily comprising cutting stock and scheduling problems in various sizes and formulations.

8.2 Comparison of the different Separation Heuristics

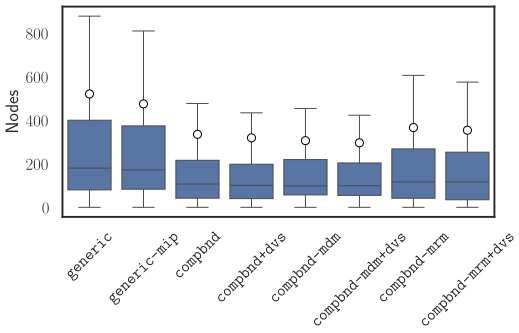
We compared different separation heuristics with and without full-tree dual value stabilization by running the test set on 12 configurations of the component bound branching rule. These configurations varied the first- and second-stage separation heuristics (Section 5.2) and the stabilization method. The naming conventions for these runs are as follows: runs using both the `MaxRangeMidrange` and `MostDistinctMedian` first-stage heuristics are named `compbnd`. Runs using only `MaxRangeMidrange` are named `compbnd-mrm`, and those using only the



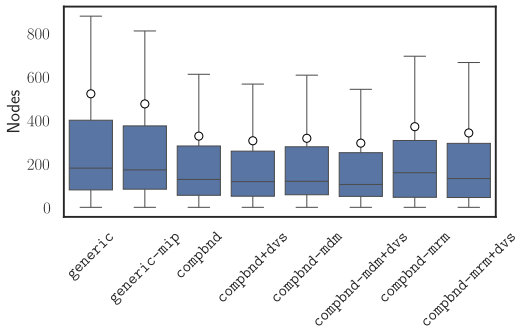
(a) `MostFractional`: solve status



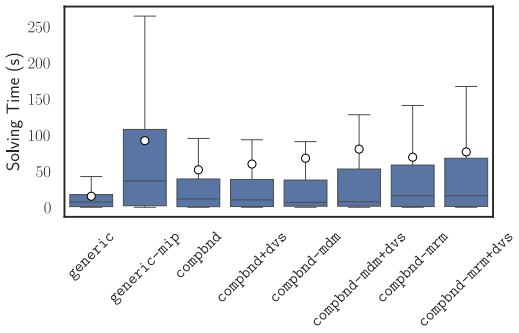
(b) `ClosestToZHalf`: solve status



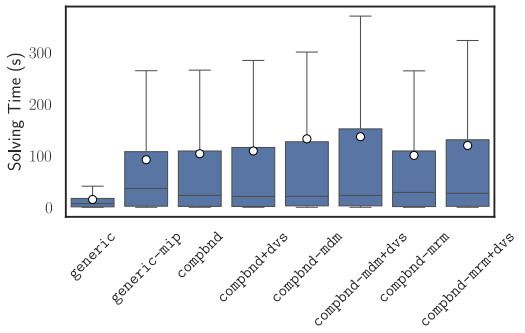
(c) `MostFractional`: nodes



(d) `ClosestToZHalf`: nodes

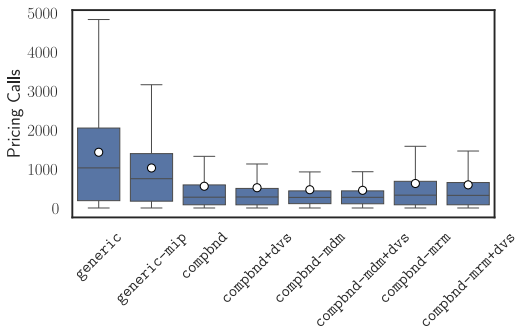


(e) `MostFractional`: times

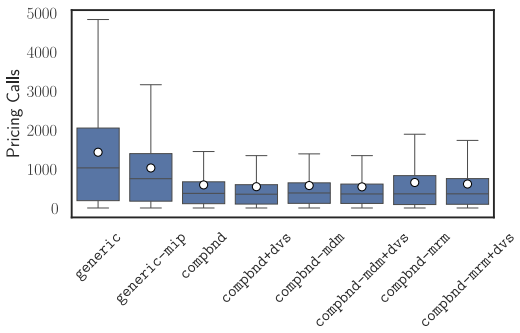


(f) `ClosestToZHalf`: times

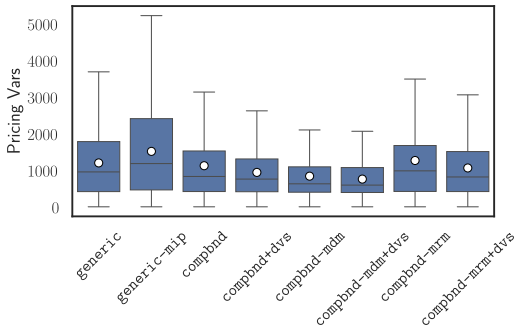
Figure 8.1: Comparison of all run configurations. In the boxplots the dots represent the arithmetic mean of the data. Outliers are not visualized. Note the differing scales in Figures 8.1e and 8.1f.



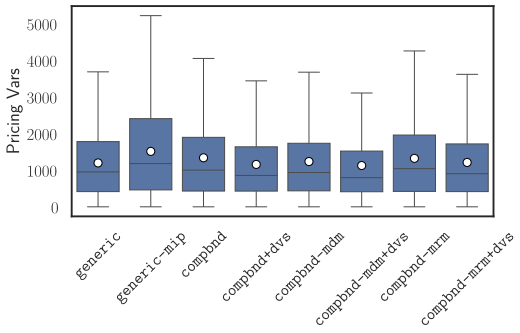
(a) `MostFractional`: pricing calls



(b) `ClosestToZHalf`: pricing calls



(c) `MostFractional`: priced variables



(d) `ClosestToZHalf`: priced variables

Figure 8.2: Comparison of all run configurations (extended). In the boxplots the dots represent the arithmetic mean of the data. Outliers are not visualized.

`MostDistinctMedian` heuristic are named `compbnd-mdm`. If full-tree dual value stabilization was applied, `+dvs` is appended to the name. For example, `compbnd-mdm+dvs` uses the `MostDistinctMedian` heuristic with full-tree dual value stabilization.

We also proposed two options for the second-stage heuristic: `ClosestToZHalf` and `MostFractional`. For readability, we grouped all runs by their second-stage heuristic and presented their statistics in separate figures.

For reference, we included Vanderbeck's generic branching scheme results, denoted as `generic`. This allows us to compare the component bound branching rule's performance against generic branching scheme. As discussed in Section 5.3, we expect the generic branching scheme to outperform the component bound branching rule, partially due to the latter having to fall back to a general *MIP* solver, while the former may retain the ability to use special-case solving algorithms after branching. To measure the impact of having to resort to a *MIP* solver, we

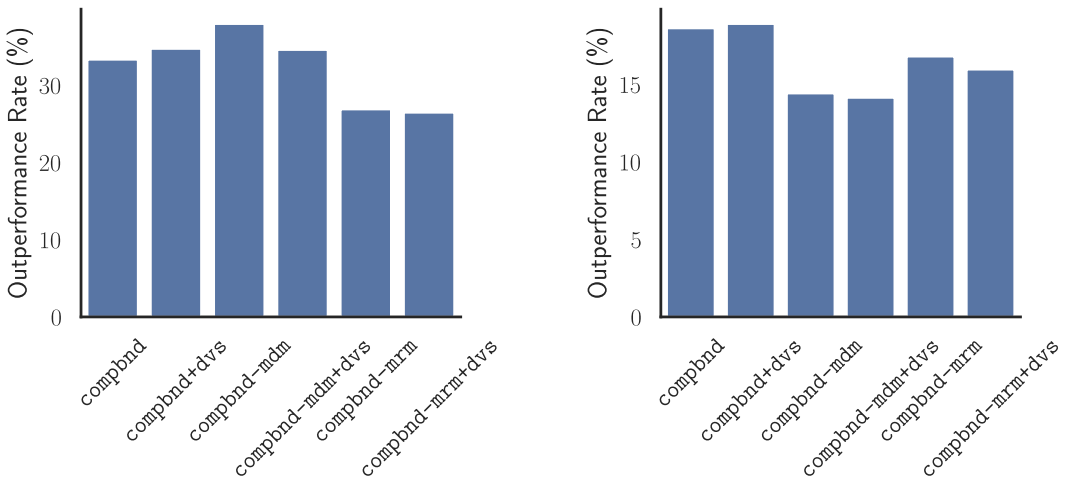


Figure 8.3: Percentage of instances solved faster by the configurations of the component bound branching rule compared to `generic`.

also included a configuration of Vanderbeck's scheme in which we force the use of a *MIP* solver at all non-root nodes, denoted as `generic-mip`.

Analyzing the results in Figure 8.1, several key observations emerge. First, full-tree dual value stabilization generally degrades the performance of the component bound branching rule. Second, runs using the `MostFractional` heuristic significantly outperform those using the `ClosestToZHalf` heuristic.

That the `MostFractional` heuristic outperforms the `ClosestToZHalf` heuristic may be explained by the former producing a smaller search tree, which in turn leads to fewer pricing calls and priced variables, as shown in Figure 8.2. As for the impact of dual value stabilization, we observe the opposite effect: although the search tree is smaller, pricing is called less often, and fewer variables are priced, the overall performance with respect to solving status and time is worse. This suggests that the management cost of dual value stabilization in non-root nodes outweighs the potential performance gains.

Among the best-performing configurations, i.e., those using the `MostFractional` second-stage heuristic, the number of instances solved and the solving times significantly improve when using both first-stage heuristics instead of just one. This pattern suggests that neither first-stage heuristic universally finds the optimal component bound sequences, highlighting the importance of the second-stage heuristic in selecting the best branching decisions.

8.3 Comparison to Vanderbeck's Generic Branching

As discussed in Section 5.3, the main difference between Vanderbeck's generic scheme and the component bound branching rule is in their modifications to the pricing problem. The `GENERIC` rule retains the pricing structure, allowing the use of specialized algorithms (e.g., knapsack solvers) at all nodes. In contrast, the `COMPBND` rule adds variables and constraints to the pricing problem, often necessitating a fallback to a general *MIP* solver, which may degrade performance. Additionally, as the search tree deepens, the `COMPBND` rule further complicates the pricing problem by adding more variables and constraints, whereas the `GENERIC` rule's pricing problems become easier to solve due to tighter bounds. Thus, we expected the `GENERIC` rule to outperform the `COMPBND` rule, particularly for larger instances.

This expectation is confirmed by our results (Figure 8.1). Vanderbeck's generic branching scheme solves more instances and does so in significantly less time compared to any configuration of the component bound branching rule.

Figure 8.3 shows how often the component bound branching rule outperforms Vanderbeck's generic branching scheme. The `MostFractional` second-stage heuristic consistently outperforms the `ClosestToZHalf` heuristic. Notably, the highest outperformance rate is achieved when only using the `MostDistinctMedian` first-stage heuristic. Combining it with the `MaxRangeMidrange` heuristic actually decreases the outperformance rate, suggesting that `MostDistinctMedian` is the most effective first-stage heuristic, while `MaxRangeMidrange` may not be as beneficial.

The surprisingly large outperformance rates we see for the `MostFractional` second-stage heuristic should be taken with a grain of salt. When we take a closer look at those instances, we observe that most of them are solved within 40 seconds by either branching rule, and often only a few seconds were saved. Therefore, the impact of this outperformance is limited. Especially given that the generic scheme is generally faster and solves more instances, it remains the preferred choice for most instances.

Things change quite a bit for Vanderbeck's generic branching when we enforce the use of a *MIP* solver for the pricing problem in non-root nodes. Although this `generic-mip` configuration produces fewer nodes, possibly due to more suitable columns being generated, having to rely on a *MIP* solver imposes a great performance penalty, see Figure 8.1. It seems as if creating a smaller search tree is far more beneficial than having to solve more complex pricing problems. Even the configurations of the weaker `ClosestToZHalf` second-stage heuristic seemingly perform on par with the generic branching scheme when it is forced to use a *MIP* solver for pricing. The `MostFractional` second-stage heuristic configurations

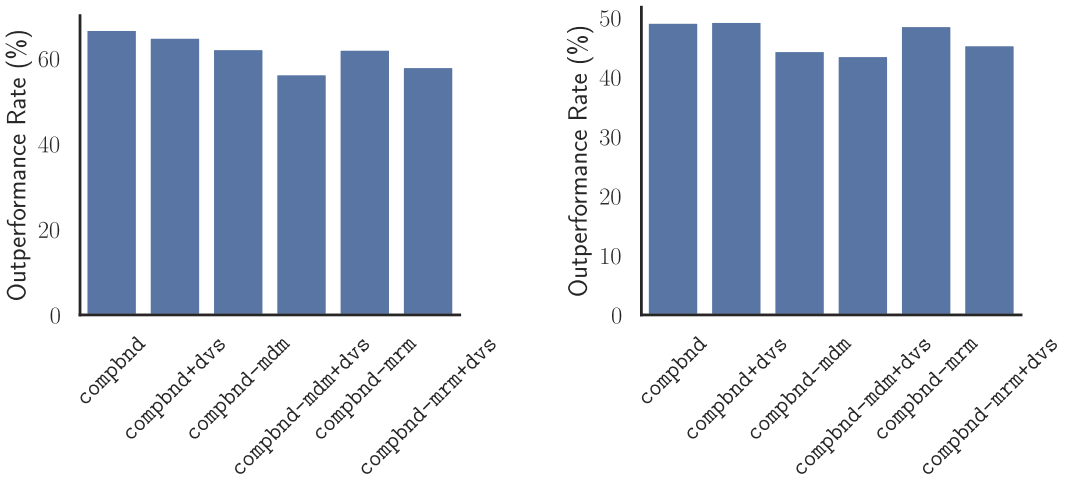


Figure 8.4: Percentage of instances solved faster by the configurations of the component bound branching rule compared to `generic-mip`.

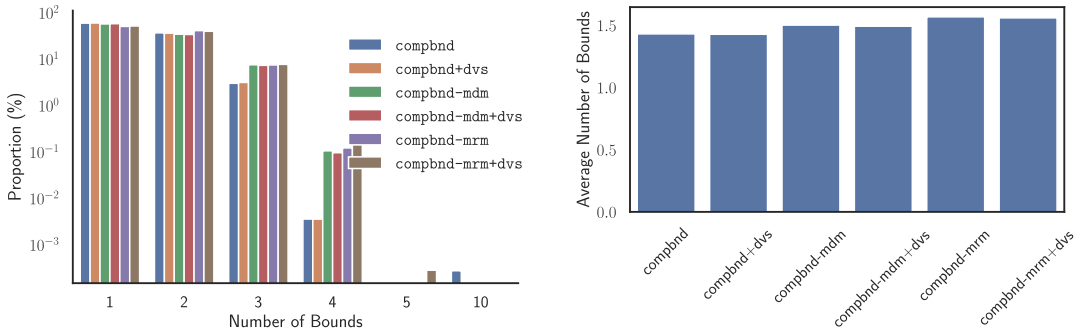
clearly outperform the generic branching scheme in this scenario, as shown in Figure 8.4.

These findings suggest that the component bound branching rule can be a viable alternative to Vanderbeck's branching scheme when no specialized algorithms are available for the pricing problem. However, the generic branching scheme remains the preferred choice when such algorithms are available.

8.4 In-Depth Analysis of the First-Stage Separation Heuristics and the Effect of Dual Value stabilization

We now examine the first-stage separation heuristics in more detail, focusing on the selected component bound sequences for branching. Since the `MostFractional` second-stage heuristic significantly outperforms the `ClosestToZHalf` heuristic, we limit our analysis to configurations using the former. For each configuration and all instances, we logged the size of the component bound sequences at each node.

Although the `compbnd` configuration always selects the minimal component bound sequence from the two first-stage heuristics, this does not mean it branches with fewer component bounds on average compared to the `compbnd-mrm` or `compbnd-mdm` configurations. Current branching decisions influence future opportunities, and mixing both first-stage heuristics can lead to more component bounds per



(a) Distribution of the number of bounds created while branching

(b) Average number of bounds created while branching

Figure 8.5: Left: distribution of the number of bounds created at each node for the different configurations. Note the logarithmic scale. Right: average number of bounds created overall for the different configuration.

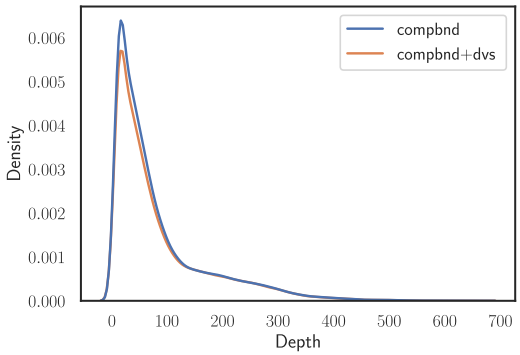
branching decision.

Figure 8.5a shows the distribution of the number of component bounds created for each branching decision. The data indicates that it is rare to branch with a component bound sequence larger than 4. Most cases involve sequences of size 1 or 2, as seen in Figure 8.5b. Additionally, both first-stage heuristics individually create more bounds on average than their combination, likely explaining why the `compbnd` configuration outperforms the others (Section 8.2).

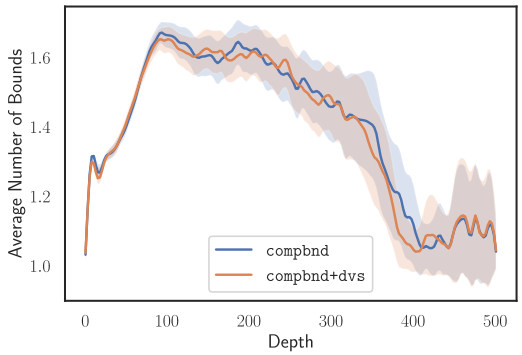
Figure 8.6 illustrates the number of branching decisions per depth and the average number of component bounds per branching decision across different depths. Given that the `COMPBND` rule creates a binary search tree and each instance is solved with only a few hundred nodes (Figure 8.1c), most branching decisions occur at depths in the low hundreds. Consequently, we plotted the average number of component bounds per branching decision up to depth 500.

Our first observation is that full-tree dual value stabilization has little effect on the average number of component bounds per branching decision. Since each configuration produces a similar number of nodes with and without stabilization (Figure 8.1c), we further estimate that both search trees are similar. Therefore, as roughly the same amount of nodes are solved with similar complexities of the pricing problems, the performance degradation discussed in Section 8.2 suggests that the management cost of dual value stabilization in non-root nodes outweighs the potential performance gains. This is in line with our initial thoughts on the impact of dual value stabilization.

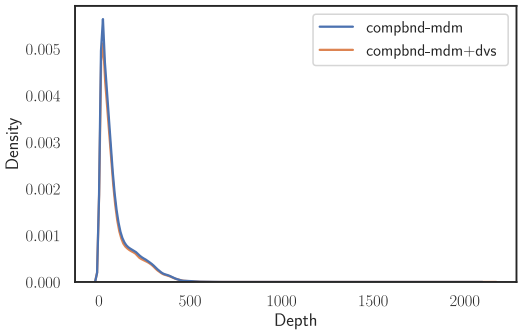
For all configurations, the average number of component bounds per branching decision rises steeply until around depth 100, then stabilizes and gradually falls,



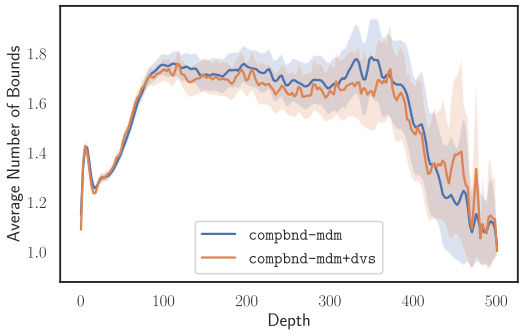
(a) `compbnd(+dvs)`: Number of branching decisions per depth



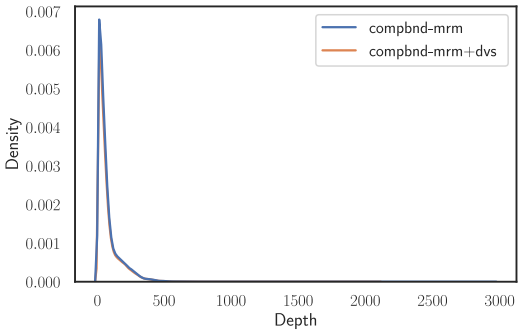
(b) `compbnd(+dvs)`: Average number of bounds per depth (95% CI)



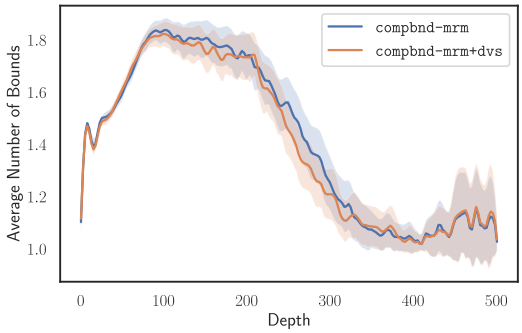
(c) `compbnd-mdm(+dvs)`: Number of branching decisions per depth



(d) `compbnd-mdm(+dvs)`: Average number of bounds per depth (95% CI)



(e) `compbnd-mrm(+dvs)`: Number of branching decisions per depth



(f) `compbnd-mrm(+dvs)`: Average number of bounds per depth (95% CI)

Figure 8.6: Left: number of branching decisions per depth. Right: average number of component bounds per depth, with 95% confidence interval (smoothed).

though variation increases. This behavior can be explained as follows: in the initial levels, there are many fractional columns, but few branching decisions have been imposed, making it easy to split the columns into two groups. As branching continues, the number of fractional columns remains high, but many constraints have already been imposed, making it harder to find separating component bound sequences, requiring more component bounds. Eventually, the number of fractional columns decreases, making it easier to find separating component bounds again. The point at which this tipping occurs likely depends on the instance.

We also observe differences between using only the `MostDistinctMedian`, the `MaxRangeMidrange`, or both first-stage separation heuristics. The combination of the two peaks at the least value out of the three configurations, while the `MaxRangeMidrange` configuration peaks the highest. However, it is also the fastest to drop back to an average of just over one, while the `MostDistinctMedian` configuration hovers at its peak for over 200 depths before decreasing.

The narrow confidence interval until depth 100, despite significant changes in the mean, is surprising. The sudden drop in the average number of component bounds at very shallow depths, followed by a steep rise, remains unexplained.

Chapter 9

Conclusion

In this thesis, we have formalized and explored constraints that exist solely within the reformulated problem in column generation, where their validity in the master problem is ensured only by specific modifications to the pricing problem. To address these needs, we have extended the `GCG` solver by introducing an interface that facilitates the creation and management of these specialized constraints, termed `generic mastercuts`. This new interface is designed to streamline the implementation of advanced branching rules and separators that rely on conditions specific to the reformulated problem.

Leveraging this interface, we have implemented the component bound branching rule, as presented by Desrosiers et al. [3]. This rule offers a simpler alternative to Vanderbeck's generic branching scheme [4] for branching on component bound sequences. Our evaluation reveals that, while Vanderbeck's scheme significantly outperforms the new component bound branching rule, this advantage is largely due to the preservation of the pricing problem's structure. This preservation allows the continued use of specialized optimization algorithms throughout the entire search tree. When Vanderbeck's scheme is forced to rely on a generic *MIP* solver for pricing, as is required by the component bound branching rule, the latter actually performs better on average. This finding underscores the critical importance of maintaining the structure of the pricing problem to enhance the performance of the column generation algorithm, thereby emphasizing the need for careful selection of the decomposition strategy.

Although the component bound branching rule does not substantially improve the overall performance of `GCG`, its implementation highlights the versatility and potential of the new generic mastercut interface. This interface makes it feasible to implement complex branching rules or separators that operate exclusively within the reformulated problem. As a further refinement, expanding the interface to accommodate constraints that do not require additional variables in the pricing problem, such as those in Vanderbeck's generic branching, would broaden its

applicability and utility.

Looking ahead, future research could focus on developing new branching rules and master separators that can fully exploit the capabilities of this interface. This work will likely open up new avenues for improving the efficiency and effectiveness of column generation-based algorithms in a wider range of applications. Moreover, given the importance of choosing the right component bound sequence for branching, as evidenced by our evaluation, further work could explore more effective heuristics for identifying these sequences. Investigating whether a theoretically optimal component bound sequence exists or developing more sophisticated heuristics could significantly enhance the performance of the component bound branching rule and, by extension, the efficiency of column generation algorithms in solving large-scale integer programs.

Bibliography

[1] François Vanderbeck and Laurence A Wolsey. *Reformulation and decomposition of integer programs*. Springer, 2010.

[2] François Vanderbeck and Laurence A Wolsey. "An exact algorithm for IP column generation". In: *Operations research letters* 19.4 (1996), pp. 151–159.

[3] Jacques Desrosiers et al. *Branch-and-Price*. Les Cahiers du GERAD G-2024-36. GERAD, Montréal QC H3T 2A7, Canada: Groupe détudes et de recherche en analyse des décisions, June 2024, pp. 1–689. eprint: `https://www.gerad.ca/papers/G-2024-36.pdf?locale=fr`. URL: `https://www.gerad.ca/fr/papers/G-2024-36`. published.

[4] François Vanderbeck. "Branching in branch-and-price: a generic scheme". In: *Mathematical Programming* 130 (2011), pp. 249–294.

[5] Nathan Chappell. "Minkowski-Weyl Theorem". In: (2019).

[6] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.

[7] George B Dantzig and Mukund N Thapa. *The simplex method*. Springer, 1997.

[8] Nikolaos Ploskas and Nikolaos Samaras. "Pivoting rules for the revised simplex algorithm". In: *Yugoslav Journal of Operations Research* 24.3 (2014), pp. 321–332.

[9] Jií Matouek and Bernd Gärtner. *Understanding and using linear programming*. Vol. 1. Springer, 2007.

[10] Marcel Schmickerath. "Experiments on Vanderbeck's generic Branch-and-Price scheme". In: (2012).

[11] Gerald Gamrath. "Generic branch-cut-and-price". MA thesis. 2010.

[12] Jonas T Witt. "Separation of Generic Cutting Planes in Branch-and-Price". PhD thesis. Master's thesis. RWTH Aachen University, 2013.

[13] Michael Bastubbe, Marco E Lübbecke, and Jonas T Witt. "A Computational Investigation on the Strength of Dantzig-Wolfe Reformulations". In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[14] Artur Pessoa et al. "In-out separation and column generation stabilization by dual price smoothing". In: *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*. Springer. 2013, pp. 354–365.

[15] Artur Pessoa et al. "Automation and combination of linear-programming based stabilization techniques in column generation". In: *INFORMS Journal on Computing* 30.2 (2018), pp. 339–360.

[16] Tobias Achterberg. "Constraint integer programming". In: (2007).

[17] Tobias Achterberg. "SCIP: solving constraint integer programs". In: *Mathematical Programming Computation* 1 (2009), pp. 1–41.

[18] Suresh Bolusani et al. "The SCIP Optimization Suite 9.0". In: *arXiv preprint arXiv:2402.17702* (2024).

[19] Michael Bastubbe et al. "strIPlib: Structured Integer Programming Library". 2025. URL: `https://striplib.or.rwth-aachen.de`.