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

June 16, 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

Abstract

This master thesis integrates the component bound branching rule, proposed by Vanderbeck et al. [1, 2], into the branch-price-and-cut solver GCG. This rule, similarly to Vanderbeck's generic branching scheme [3], 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 improvements in the branch-price-and-cut framework, including dual value stabilization, without necessitating alterations to 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

Chapter 2

Preliminaries

In this preliminary chapter we will provide a brief rundown of theorems and algorithms on which the techniques described in later chapters, such as Column Generation in Section 3.1, are building upon. Understanding these concepts is essential to understanding the theory later presented. If, however, one is familiar with these, we invite the reader to skip ahead to Chapter 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:

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

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.*

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

Definition 2.4. *Let $\mathcal{P}$ be a convex set. A vector $\boldsymbol{r} \in \mathbb{R}_0^n \backslash \{0\}$ 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 $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k \in \mathbb{R}_+^n$ we denote as:

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

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.*

Proof. Let $\mathcal{H} = \{\boldsymbol{x} \mid \boldsymbol{f}^\intercal \boldsymbol{x} = f\}$ be a hyperplane. Let $k \in \mathbb{N}$, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \in \mathcal{H}$. For any $\alpha_1, \ldots, \alpha_k \in \mathbb{R}_+$ with $\sum_{i=1}^k$:

$$\begin{aligned} \boldsymbol{f}^\intercal \left(\sum_{i=1}^k \alpha_i \boldsymbol{x}_i \right) &= \sum_{i=1}^k \alpha_i \boldsymbol{f}^\intercal \boldsymbol{x}_i \\ &= \sum_{i=1}^k \alpha_i \cdot f \\ &= f \cdot \sum_{i=1}^k \alpha_i \\ &= f \end{aligned}$$

Therefore, the convex combination $\sum_{i=1}^k \alpha_i \boldsymbol{x}_i$ is in the hyperplane $\mathcal{H}$. □

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

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

Proof. Let $\mathcal{H}^+ = \{\boldsymbol{x} \mid \boldsymbol{f}^\intercal \boldsymbol{x} > f\}$ be an open halfspace (analogous for $\mathcal{H}^- = \{\boldsymbol{x} \mid \boldsymbol{f}^\intercal \boldsymbol{x} < f\}$, and for the closed halfspaces). Let $k \in \mathbb{N}$, $\boldsymbol{x_1}, \ldots, \boldsymbol{x_k} \in \mathcal{H}$. For any $\alpha_1, \ldots, \alpha_k \in \mathbb{R}_+$ with $\sum_{i=1}^k$:

$$\begin{aligned} \boldsymbol{f}^\intercal \left(\sum_{i=1}^k \alpha_i \boldsymbol{x}_i \right) &= \sum_{i=1}^k \alpha_i \boldsymbol{f}^\intercal \boldsymbol{x}_i \\ &> \sum_{i=1}^k \alpha_i \cdot f \\ &= f \cdot \sum_{i=1}^k \alpha_i \\ &= f \end{aligned}$$

Therefore, the convex combination $\sum_{i=1}^k \alpha_i \boldsymbol{x}_i$ is in the halfspace $\mathcal{H}$. □

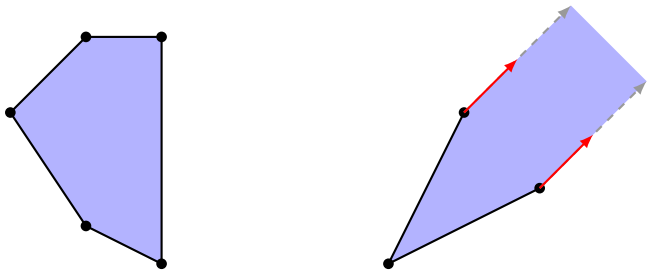

Figure 2.1: Illustration of the Minkowski-Weyl theorem. The left figure shows a fully encapsulated polyhedron which can be represented only by its extreme points. Unbounded polyhedra, such as the one on the right, require extreme rays, drawn in red, to be described completely.

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 $\boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m$.*

By Corollaries 2.1 and 2.3, a polyhedron is also a convex set of points.

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)**.** *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. Most polyhedra can be represented in this way using only their 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.

Theorem 2.2 (Nemhauser-Wolsey)**.** *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. $\text{rank}(\boldsymbol{Q}) = m \leq n \wedge \mathcal{P} \neq \emptyset$.*

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:

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

In the Nemhauser-Wolsey theorem, the conditions of the Minkowski-Weyl theorem are clearly encoded: the second and third lines ensure that the convex set of the extreme points are considered in the first line (Definition 2.2), the last playing a part in the cone of extreme rays (Definition 2.4), and the first line being the Minkowski sum of the convex hull of extreme rays and the cone of extreme rays.

The Nemhauser-Wolsey theorem has also been adapted to integral polyhedra:

Theorem 2.3 (Nemhauser-Wolsey)**.** *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 $\mathcal{Q} \coloneqq \mathcal{P} \cap \mathbb{Z}^n \neq \emptyset$ by the integer hull of $\mathcal{P}$.*

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

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

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

Have 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 finds an optimal solution by moving from one extreme point of the polyhedron to the next, therefore always remaining feasible. A central part 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 gives rise to the **pricing problem**, which either verifies the optimality of the current basic solution, and otherwise determines the non-basic variable x_l, $l \in \mathcal{N}$ with the least reduced cost ($\bar{c}_l < 0$) to be swapped into the basis next, according to Dantzig's rule (TODO cite). Formally, this can be written as:

$$l \in \arg\min_{j \in \mathcal{N}} c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j \tag{2.4}$$

or as the linear program:

$$\bar{c}(\boldsymbol{\pi}) = \min_{j \in \mathcal{N}} c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j \tag{2.5}$$

Solving the pricing problem thus plays an integral role in 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
  $\boldsymbol{\pi}^\intercal \leftarrow \boldsymbol{c}_\mathcal{B}^\intercal \boldsymbol{A}_\mathcal{B}^{-1}$; $\bar{\boldsymbol{b}} \leftarrow \boldsymbol{A}_\mathcal{B}^{-1}\boldsymbol{b}$;
  $\bar{c}_j \leftarrow c_j - \boldsymbol{\pi}^\intercal \boldsymbol{a}_j; \qquad \forall j \in \mathcal{N}$
  $l \leftarrow \arg\min_{j \in \mathcal{N}} \bar{c}_j$; $\bar{c}(\boldsymbol{\pi}) \leftarrow \bar{c}_l$;
  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
  $s \leftarrow \arg\min_{i \in \{1,\dots,m\}} \frac{\bar{b}_i}{\bar{a}_{il}}$; $x_l \leftarrow \frac{\bar{b}_s}{\bar{a}_{sl}}$; $\mathcal{B} \leftarrow \mathcal{B} \cup \{l\} \subseteq \{s\}$; $\mathcal{N} \leftarrow \mathcal{N} \cup \{s\} \subseteq \{l\}$;

Chapter 3

Column Generation and Branch-and-Price

3.1 Column Generation

Let us consider the following linear program, which we will henceforth call 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 huge, i.e. a lot larger than the number of constraints ($m \ll |\mathcal{X}| < \infty$). Because of this, solving MP in a reasonable amount of time, sometimes at all, is infeasible.

We can, however, make use of 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 also function using a manageable subset of variables $\mathcal{X}' \subseteq \mathcal{X}$, finding a possibly non-optimal, yet still feasible solution for the entire optimization problem MP. We denote this master problem restricted to a subset of variables as 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 aspects of finding an optimal solution to *MP* are still missing: first, how do we find a subset $\mathcal{X}'$ of the variables, such that *RMP* stays feasible? Without this property of the set of variables, no solution of *RMP* can be found, and therefore none can be found for *MP*, which would contradict the feasibility of *MP*. Secondly, assuming a solution of *RMP* was found, possibly even optimal for the *RMP*, how could we build upon this solution to eventually find an optimal solution for *MP*?

In the following we will dive into these two questions in detail (Sections 3.1.1 and 3.1.2), making way for the final column generation algorithm (Section 3.1.3).

3.1.1 Farkas Pricing

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

Theorem 3.1 (Farkas' lemma)**.** *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. $\exists \boldsymbol{\pi} \in \mathbb{R}^n_+ . \boldsymbol{\pi}^\intercal \boldsymbol{A} \leq \boldsymbol{0} \wedge \boldsymbol{\pi}^\intercal \boldsymbol{b} > 0$

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

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

Furthermore, from the infeasibility of *RMP* we can also derive the following statement:

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

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

This process of finding corresponding columns $\boldsymbol{a}_{\boldsymbol{x}}$ to add to the *RMP* can be formalized as a pricing problem with cost coefficients $c_{\boldsymbol{x}} = 0$ (see Equation (2.5)). 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 solution 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.

3.1.2 Reduced Cost Pricing

Assume the *RMP* is feasible. Using a solver of our choice, we can now construct a solution that is optimal within the *RMP*, providing us with the dual values $\boldsymbol{\pi}$. One now must verify whether this solution is also optimal for the *MP*. For this purpose, we can utilize the pricing problem we are already familiar with from the primal simplex algorithm (see Equation (2.5)). Let us denote this subproblem as the *RCP-SP*:

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

Note, that due to the optimality of *RMP* the reduced costs of all variables $\boldsymbol{x} \in \mathcal{X}'$ are already non-negative. If now $\bar{c}(\boldsymbol{\pi}) \geq 0$, we have also proven optimality of the current solution for the *MP*. Otherwise, if $\bar{c}(\boldsymbol{\pi}) < 0$, then there is some $\boldsymbol{x} \in \mathcal{X} \backslash \mathcal{X}'$ with $\bar{c}(\boldsymbol{\pi}) = c_{\boldsymbol{x}} - \boldsymbol{\pi}^\intercal \boldsymbol{a_x} < 0$. Similarly to how the primal simplex algorithm would then 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 the *RMP*. An important property of this process is that the *RMP* remains feasible.

3.1.3 Column Generation Algorithm

The column generation algorithm can now be viewed as a variation of the primal simplex algorithm: We start to solve our problem, the *MP*, with a subset of the original variables, initialized as the empty set or by 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 is feasible, or until there are no new variables to add, proving infeasibility of *MP*. As soon as *RMP* is feasible, we solve it to optimality, using reduced cost pricing to verify whether the solution is also optimal for the *MP*. If it is, we have found the optimal solution to the *MP*. Otherwise, we add the corresponding column to the *RMP* and repeat the process.

Note 3.1. All columns in the *RMP* are distinct by design. If a subproblem produced a column that was already present in the *RMP*, its reduced cost would be non-negative, and it would not be added to the *RMP* in the first place. Therefore, we can assume that all columns in *RMP*, and thus in $Q(S)$, are distinct.

Algorithm 3.1: Column generation algorithm

Input: *RMP* with subset $\mathcal{X}' \backslash \mathcal{X}$, *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 especially practical when we can directly formulate our optimization problem using a master and a pricing problem. Oftentimes, however, we do not have these constructions readily available. Instead, many problems are given in the more general form of a LP. Using the Dantzig-Wolfe reformulation, we can automatically transform such a LP into a master and pricing problem, allowing us to apply column generation. In this section, we will introduce this technique and show how it can be used to solve a LP.

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

Take the above LP as an example. The solution space if this LP, defined by its constraints, can also be viewed as the intersection of the following two polyhedra:

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

After 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 $\{\boldsymbol{x}_r\}_{r \in R}$. For this, we substitute the original variables $\boldsymbol{x}$ with these extreme points and extreme rays using:

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

Let us also use the following shorthand notations:

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

As a result, we have obtained 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 afterwards to reconstruct a solution using the original $\boldsymbol{x}$ variables.

As the number of extreme points and extreme rays of $\mathcal{D}$ might be huge, it is most often than not practically infeasible to solve the *MP* directly. Instead, using this setup, we can easily generate these columns on the fly 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 easily 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 of by solving the *RMP* using a subset of the extreme points $P' \subset P$ and extreme rays $R' \subset R$, giving us the dual values $\boldsymbol{\pi_b}$ and π_0 for the *SP*. Solving this *SP* to optimality then leads a solution $\boldsymbol{x}^*$ with objective value z^*_{SP}. The value of z^*_{SP} is now the deciding factor whether we add a column to *RMP*, and if so, which column we 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 the *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 the *RMP*.

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

While in theory it does not matter how we group the constraints of our original formulation LP for the Dantzig-Wolfe reformulation, since all groupings result in equivalent optimal solutions, in practice the choice of grouping can have a significant impact on the performance of the column generation algorithm. Since most of the time many iterations of the column generation algorithm are required to find an optimal solution, ideally one wants the SP to be efficiently solvable. Many highly efficient algorithms for specific optimization problems exist, and by grouping constraints in a way that the SP corresponds to such structures, one can leverage these algorithms to solve the SP efficiently. Thankfully, there are ways of finding such groupings automatically, although this goes 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 dive into how we then solve such an integer program using column generation.

Take the following integer program as an example:

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

Once 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:

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

From here, there are two ways to proceed. The straightforward approach, called **Convexification**, follows the approach seen in the Dantzig-Wolfe reformulation of linear programs, in addition to keeping the integrality constraints on $\boldsymbol{x}$ in both the master and pricing problem. On the other hand, during **Discretization**, we modify our approach slightly, adding integrality constraints to the master variables to ensure integrality of the original variables.

3.3.1 Convexification

As we have 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 gives us 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, plays a crucial role during the solving process to ensure the integrality of the original variables, and therefore cannot simply be computed after a solution has been found. The master problem has the following pricing subproblem:

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

This is it. These two changes marked in blue are the only 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 integer hull of $\{\boldsymbol{x} \geq \boldsymbol{0} \mid \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d}\}$, regardless

of how well the constraints $\boldsymbol{Dx} \geq \boldsymbol{d}$ actually approach this integer hull. Therefore, we implicitly make use of the integer hull of $\mathcal{D}$, without having to explicitly define it.

λ solutions to the *MP* might lead to fractional $\boldsymbol{x}$ solutions. In this case, we have no choice but to branch on those fractional original variables. We will discuss this in more detail in Section 3.5.1.

3.3.2 Discretization

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

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

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. We will discuss this in more detail in Section 3.5.2.

Keeping in mind that $\ddot{P}$ is a subset of integer points of $\mathcal{D}$, i.e. 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 very trivial, since in mathematical optimization one only tries to find the most optimal solutions, i.e. the extreme points. We can, however, postpone this concern for now, and use the same pricing problem as in the convexification approach:

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

As we will find out later in Section 3.5.2, this concern of generating interior points is addressed during the branching process, which allows 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 several (different) families of variables and constraints, which can be decomposed into several (different) subproblems. Column generation can also be adapted to this scenario, where we have a set K of subproblems SP^k generating variables $\boldsymbol{x}^k \in \mathcal{X}^k$: 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 now proceeds as before with the adaption that it now 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 LPs and IPs. 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, i.e.:

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

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

LPs is (analogous for convexification and discretization of IPs):

$$\begin{aligned}
z^*_{MP} = \min \quad & \sum_{k\in K}\sum_{p\in P^k} c^k_p\lambda^k_p + \sum_{k\in K}\sum_{r\in R^k} c^k_r\lambda^k_r & & \\
\text{s. t.} \quad & \sum_{k\in K}\sum_{p\in P^k} \boldsymbol{a}^k_p\lambda^k_p + \sum_{k\in K}\sum_{r\in R^k} \boldsymbol{a}^k_r\lambda^k_r \geq \boldsymbol{b} & & [\boldsymbol{\pi_b}] \\
& \sum_{p\in P^k} \lambda^k_p = 1 & & \left[\pi^k_0\right] \forall k \in K \\
& \lambda^k_p \geq 0 & & \forall k \in K, \forall p \in P^k \\
& \lambda^k_r \geq 0 & & \forall k \in K, \forall r \in R^k \\
& \sum_{p\in P^k} \boldsymbol{x}^k_p\lambda^k_p + \sum_{r\in R^k} \boldsymbol{x}^k_r\lambda^k_r = \boldsymbol{x}^k \geq \boldsymbol{0} & & \forall k \in K
\end{aligned} \tag{3.24}$$

And each subproblem SP^k is given by:

$$\begin{aligned}
z^*_{SP^k} = \min \quad & \left(\boldsymbol{c}^{k\intercal} - \boldsymbol{\pi}^\intercal_{\boldsymbol{b}}\boldsymbol{A}^k\right)\boldsymbol{x}^k - \pi^k_0 & & \\
\text{s. t.} \quad & \boldsymbol{D}^k\boldsymbol{x}^k \geq \boldsymbol{d}^k & & \left[\boldsymbol{\pi}^k_{\boldsymbol{d}}\right] \\
& \boldsymbol{x}^k \geq \boldsymbol{0} & &
\end{aligned} \tag{3.25}$$

Let us 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 the MP different λ^k_p (λ^k_r) variables for different k correspond to the same extreme point $\boldsymbol{x}_p$ ($\boldsymbol{x}_r$), which is redundant, and could therefore slow down the solving process. In a process called **aggregation** we can improve upon this by aggregating these variables:

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

Substituting these aggregated variables in the *MP* yields:

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

For which columns are generated by the following subproblem:

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

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 the *MP* are the integrality conditions on $\boldsymbol{x}^k$ in constraint (3.27k). In convexification, however, we can only ensure 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 the *RMP* an original solution.

Disaggregation, however, offers a powerful alternative. Its *MP* for identical

subproblem looks as follows:

$$
\begin{aligned}
z^*_{MP} = \min & \sum_{p\in P} c_p\lambda_p + \sum_{r\in R^k} c_r\lambda_r && && && (3.29a)\\
\text{s. t.} & \sum_{p\in P} \boldsymbol{a}_p\lambda_p + \sum_{r\in R^k} \boldsymbol{a}_r\lambda_r \geq \boldsymbol{b} && && [\boldsymbol{\pi_b}] && (3.29b)\\
& \sum_{p\in P} \lambda_p = |K| && && [\pi_{agg}] && (3.29c)\\
& \lambda_p \in \mathbb{Z}_+ && && \forall p \in P && (3.29d)\\
& \lambda_r \in \mathbb{Z}_+ && && \forall r \in R && (3.29e)\\
& \sum_{k\in K} \lambda^k_p = \lambda_p && && \forall p \in P && (3.29f)\\
& \sum_{k\in K} \lambda^k_r = \lambda_r && && \forall r \in R && (3.29g)\\
& \sum_{p\in P} \lambda^k_p = 1 && && \forall k \in K && (3.29h)\\
& \lambda^k_p \in \mathbb{Z}_+ && && \forall k \in K, \forall p \in P && (3.29i)\\
& \lambda^k_r \in \mathbb{Z}_+ && && \forall k \in K, \forall r \in R && (3.29j)\\
& \sum_{p\in P} \boldsymbol{x}_p\lambda^k_p + \sum_{r\in R} \boldsymbol{x}_r\lambda^k_r = \boldsymbol{x}^k \in \mathbb{Z}^n_+ && && \forall k \in K && (3.29k)
\end{aligned}
$$

In Section 3.3.2 we have already observed that the integrality constraints on the original variables $\boldsymbol{x}^k$ are already enforced by ensuring 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 remove the integrality constraints on the disaggregated master variables, as those are implied by the integrality of the aggregated variables λ_p and λ_r. Therefore, during the entire solving process, we can omit the constraints (3.29f) to (3.29k) entirely.

On a final note, it is of course 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 have seen how to reformulate an integer program into a master and pricing problem, specifically focusing on the integrality conditions. In this section, we will dive into how we then solve such an integer master program using column generation. First, let us remember what branching is for. Recall, that often we cannot solve an integer problem directly. Instead, we rely on the LP

relaxations of the problem which in turn can be solved by algorithms such as the simplex method. An optimal solution of the *LP* relaxation might have some fractional values for the integer variables, i.e. produce infeasible solutions for the *IP*. To overcome this, we branch on these fractional variables, creating subproblems, which explicitly cut off these fractional solutions. By recursively solving these subproblems, we eventually find an optimal integer solution. This process is widely known as **branch-and-bound**.

In the context of column generation for integer master programs, we proceed similarly: first, we relax the integrality constraints of the master problem, which allows 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. Combining branching with column generation, we obtain the term **branch-and-price**.

We have gotten to know two distinct approaches of reformulating an *IP* into a (integer) master and pricing problem: convexification (Section 3.3.1) and discretization (Section 3.3.2). Since we require integrality of the original variables in both approaches, it is always possible to branch on fractional solutions of the original variables. We have seen, however, that discretization additionally introduces integrality constraints on the master variables which in turn imply integrality of the original variables. Therefore, in discretization, we can branch on the master variables as well. 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 is some $x^*_j \notin \mathbb{Z}$ for some integer variable x_j. Then we can cut off this fractional solution by creating two subbranches (**dichotomous branching**), one where $x_j \leq \lfloor x^*_j \rfloor$ and one where $x_j \geq \lceil x^*_j \rceil$. In each of these subtrees, a solution to the *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 as well.

Note 3.2. Branching on the original variables obviously 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 actually two ways to enforce this branching decision:

3.5.1.1 Branching in the Master Problem

Recall that the *MP* includes the following constraint:

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

Obviously, 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 the *MP* (analogous for the up-branch):

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

In order to keep generating only improving columns after branching, we must consider the dual variable α_j in the pricing problem:

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

3.5.1.2 Branching in the Pricing Problem

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

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

Unfortunately, the *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 could be achieved by removing such columns altogether, or by adding the following constraint to the *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 $Q \coloneqq \ddot{P} \cup R$. Assume our master solution $\boldsymbol{\lambda}_{RMP}^*$ is fractional, i.e. $\lambda_q^* \notin \mathbb{Z}$ for at least one $q \in Q$. Unfortunately, in this context, dichotomous branching on such a singular variable λ_q is very weak:

Assume $q \in \ddot{P}$. 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 in fact only little better than enumerating all possible solutions. Cutting off multiple fractional solutions in each child node would be more desirable.

Alternatively, given a fractional master solution $\boldsymbol{\lambda}^*_{RMP}$, we can find a subset $\emptyset \subset Q' \subset Q$ of variables, for which the following holds:

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

It is obvious that such a subset Q' always exists, e.g. for dichotomous branching choose $Q' = \{\lambda_q\}$. 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 now be adapted to ensure the validity of the branching decision in the master:

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

The challenge which remains is determining a routine to find such a subset Q' in the master solution, for which the logical equivalence to be added to the *SP*, i.e. the set inclusion rule, is also *expressible using a finite set of linear constraints*, that can be added to the *SP* instead.

Note 3.3. Adding the variable y to the subproblem is the deciding property which 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.

Note 3.4. Aggregation of subproblems (Section 3.4) is not of an issue during branching. In such cases, we would simply branch on the aggregated variables within each block (group of identical subproblems). Yet, for the purpose of more readable notation, we will not consider formulations with multiple subproblems from here on out, focusing only on cases with a single non-aggregated block.

3.5.2.1 Vanderbeck's Generic Branching Scheme

Vanderbeck has proposed an elaborate scheme (`GENERIC`) to find such a subset Q' in the master solution, which can be used to branch on the master variables for any type of bounded *IP*, i.e. which has no extreme rays ($Q = \ddot{P}$). This branching rule is based on component bounds on original variables:

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

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

where η is the type of the 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:

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

Let us further introduce the following shorthand notation:

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

For a given component bound sequence S, a set of columns Q, we can define the restriction of Q to S, i.e. all columns that satisfy 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 chooses to strictly divide up the solution space, i.e. the polyhedron, along the component bounds into multiple sub-polyhedra. In this way, each child branch will be able to only generate points within its own sub-polyhedron, and the master solution will be forced to be integrally within one of these sub-polyhedra. In fact, this scheme closely resembles 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 possible sub-polyhedra. As an exponential increase in nodes is undesirable, we can instead 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 in the following way: 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 the *RMP* of node j add the following constraint:

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

Finally, modify the pricing problem as follows:

$$\begin{aligned} z_{SP}^* = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\, \boldsymbol{x} - \gamma_j - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & x_i \leq \lfloor v \rfloor \qquad \forall\, (x_i, \leq, v) \in S_j \\ & x_i \geq \lfloor v \rfloor + 1 \quad \forall\, (x_i, >, v) \in S_j \\ & \boldsymbol{x} \in \mathbb{Z}_+^n \end{aligned} \tag{3.47}$$

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

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

The procedure of finding a component bound sequence S as described in Proof 3.5.2.1 leads to dichotomous branching. As discussed before, branching on a single master variable leads to an extremely unbalanced tree. To overcome this, Vanderbeck proposes a routine that more evenly divides the solution space into multiple branches. To begin with, we initialize the component bound sequence with as $S = \emptyset$ in the root node, and otherwise set it equal to the component bound sequence of the parent node. Then, we iteratively add component bounds to S as follows:

Algorithm 3.2: Vanderbeck's Generic Branching Separation Routine

Input: parentnode
Output: S
if parentnode = null **then**
  $S \leftarrow \emptyset$;
else
  $S \leftarrow$ parentnode.S;
end
loop
  $\boldsymbol{\alpha} \leftarrow \sum_{q \in Q(S)} \boldsymbol{x}_q \lambda_q^*$;
  if $\forall 1 \leq i \leq n . \alpha_i \in \mathbb{Z}$ **then**
    return S;
  end
  $i \leftarrow \text{any}(i \mid \alpha_i \notin \mathbb{Z})$;
  if Choose(i) **then**
    $S \leftarrow S \cup \{(x_i, \leq, \lfloor x_i^* \rfloor)\}$;
  else
    $S \leftarrow S \cup \{(x_i, >, \lfloor x_i^* \rfloor)\}$;
  end

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

3.6 Branch-Price-and-Cut

From solving *IP*s we know that adding cutting planes, i.e. valid inequalities, can significantly improve the performance of the branch-and-bound algorithm. Such cutting planes can be generated and added to the *LP* relaxation in any stage of the solving process. This is now known as the branch-and-cut algorithm.

We can apply the same idea 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*, hopefully strengthening the relaxation. Therefore, we can extend the branch-and-price algorithm to a branch-price-and-cut algorithm. In general, separators generating cuts for the *MP* are either operating on the original formulation or within the master problem of the Dantzig-Wolfe reformulation. In the following, we will superficially cover these two types of separators. For more detail about cutting planes for column generation and their effectiveness we refer to [5, 6].

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 now be projected back into a solution $\boldsymbol{x}^*$ of the original formulation. We can now 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 now 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:

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

In this way, existing separators originally intended for use in a branch-and-cut scenario can be reused to generate cutting planes for the Dantzig-Wolfe reformulation. Some caveats apply, however, for example do some separators rely on a basis solution. Since the Dantzig-Wolfe reformulation might be stronger than the original formulation (see Section [**TODO**] TODO), an interior point of the polyhedron could be the optimal solution for the relaxed master problem. 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

Recall that through discretization we obtain a *MP* with integral master variables. To strengthen the *LP* relaxation of the *MP*, we would also like 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 have to find cuts over all variables in *MP*, i.e. cuts that can also be imposed in the subproblem to limit which columns can be generated. More formally, we would like to find a function $\mathrm{g} : \mathbb{R}^n \to \mathbb{R}$ such that the cut is expressible as:

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

which requires the following modifications to the pricing problem:

$$\begin{aligned} z^*_{SP} = \min \quad & (\boldsymbol{c}^\intercal - \boldsymbol{\pi}_{\boldsymbol{b}}^\intercal \boldsymbol{A})\,\boldsymbol{x} - \gamma^\intercal g_{\boldsymbol{x}} - \pi_0 \\ \text{s. t.} \quad & \boldsymbol{D}\boldsymbol{x} \geq \boldsymbol{d} \\ & g_{\boldsymbol{x}} = \mathrm{g}(\boldsymbol{x}) \\ & \boldsymbol{x} \in \mathbb{Z}^n_+ \\ & g_{\boldsymbol{x}} \geq 0 \end{aligned} \tag{3.53}$$

Similar to what we have observed for branching on master variables (Section 3.5.2), the non-trivial task is now to express $g_{\boldsymbol{x}} = \mathrm{g}(\boldsymbol{x})$ using a finite set of linear constraints.

// TODO: Reference to Chantal's thesis

3.7 Dual Value Stabilization

During column generation, it has been observed that the dual values oscillate erratically, which means it takes more column generation iterations to generate columns that are considered profitable. One can, however, stabilize the dual values, decreasing such oscillations, resulting in a significant performance improvement of the column generation algorithm. This section covers the fundamentals of the hybridization of the dynamic alpha-schedule stabilization with an ascent method, as proposed by Possea et al., for which we will introduce the building blocks step by step. For more detail, we refer to [7, 8].

// TODO

Chapter 4

SCIP Optimization Suite

4.1 SCIP

4.2 GCG

Chapter 5

Component Bound Branching

In this chapter we will present the **component bound branching rule** (`COMPBND`) for branching on the master variables of the discretized reformulation of any type of bounded *IP*. Building upon the same fundamental ideas of Vanderbeck's generic branching scheme (Section 3.5.2.1), our goal with this new branching rule is to provide a simpler alternative to branching on component bounds. In the following, we will first present how we can enforce component bounds in such a way that leads to a binary branch-and-bound search tree. Afterwards, we will dive into the algorithm responsible for finding suitable branching decisions, and finally, we will highlight notable similarities and differences between Vanderbeck's generic branching scheme and our new approach.

5.1 Overview of the branching scheme

Recall from Section 3.5.2, given a fractional master solution $\boldsymbol{\lambda}^*_{RMP}$, we can always find a subset $\emptyset \subset Q' \subset Q \coloneqq \ddot{P}$ for which the following holds:

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

and we therefore can eventually enforce 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.

For finding such a Q', which is expressible in the SP, Vanderbeck uses bounds on the original variables (Section 3.5.2.1). Similar, we introduce a variation of component bounds on the original variables. Notably, instead of allowing fractional bounds, we now require integral bound values, and therefore enabling us to use $\geq$ instead of $>$.

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

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

Again, we define a component bound sequence:

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

as well as 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 now redefine the following shorthand notation:

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

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

Using an analogous proof as 3.5.2.1, it can be shown that such a S always exists if the master solution is not integral. After we have obtained such a S, we can now create two child nodes, the down- and up-branches respectively, 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*:

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

What remains is expressing all logical equivalences using a finite set of linear constraints. For this, we can use the following observation:

- Since in the down branch $-\gamma_\downarrow \geq 0$, y naturally takes on value 0 and therefore also 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 on 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.

Recall, that we require a bounded *IP* to begin with. Let us denote the lower and upper bound 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:

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

We have now successfully defined the branching decision in the master problem, and the corresponding constraints in the pricing problem. Until we have found an optimal integral solution of master variables, we will continue to branch using a suitable component bound sequence S, creating a binary branch-and-bound search tree. In the next section, we will present an algorithm responsible for finding such a 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 particular, $F_S \in \mathbb{Z}_+ \backslash \{0\}$ in this case, due to the convexity constraint $\sum_{q \in Q} \lambda_q = 1$ in the MP (analogous in aggregated subproblems, see Section 3.4).

Now take $S \neq \emptyset$, 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 now rewrite this as:

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

  We observe the sum $\sum_{q \in \mathcal{Q}(S)} \lambda^*_q =: K$ to be fractional, which enables 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. If we denote the rounded median of these two column entries as $v \coloneqq \lfloor \frac{x_{i,q_1} + x_{i,q_2}}{2} \rfloor$, observe that $x_{i,q_1} \leq v < v + 1 \leq x_{i,q_2}$, and therefore we can now 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 $F_S = F_{S_1} + F_{S_2}$, and therefore we can always halve the fractionality of the current solution with respect to the current component bound sequence.

Furthermore, both $Q(S_1)$ and $Q(S_2)$ are guaranteed to contain at least one fractional column, thus $F_{S_1}, F_{S_2} > 0$.

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.

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 as described above is not complete yet, as we have not yet defined on which of oftentimes multiple components x_i we impose which bounds. This choice can have a significant impact on 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. Maintaining balance within the branch-and-bound tree is generally thought to be beneficial, however the time required to find a suitable S can be significant, and must be traded off against the time saved by having a balanced tree.

There are numerous heuristics for selecting a component to separate on, for example choosing the component that has the most distinct values within all fractional columns of $Q(S^k)$, and there are countless possibilities for deciding which bound to impose on the selected component x_i, such as imposing lower bounds only if $v - x_{i,q_1} < x_{i,q_2} - v$, or choosing the component bound sequence for which $|Q(S)|$ is larger.

We propose choosing the component x_i for which:

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

i.e. x_i is the component with the largest range dispersion within the fractional columns of $Q(S^k)$. With this heuristic we hope to subdivide the domain of the original variables as evenly as possible, and therefore maintain a balanced branch-and-bound tree.

After we have selected a component x_i to branch on, we propose to proceed with the component bound on x_i, i.e. either $(x_i, \leq, v)$ or $(x_i, \geq, v+1)$, for which the resulting component bound sequence S has the least fractionality. This choice is motivated by the fact that we then always at least halve the fractionality of the current selection, eventually falling between 0 and 1. Making use of this property, we expect to reduce the number of iterations required to find a suitable S, thus reducing the processing time, and also possibly minimizing the size of S, which decreases the amount of variables and constraints to be added to the SP.

Since there are endless combinations of heuristics for selecting a component and a bound, one could alternatively rewrite the separation procedure as a recursive algorithm which returns (a subset of) all possible component bound sequences. From this selection, one could then choose some S to branch on, though even here the choice of S is not unique. One could also prepare a set of promising component bound sequences and then apply the strong branching heuristic to continue with the candidate that gives the best progress [9].

5.2.2 Post-processing of Component Bound Sequences

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 is actually relevant, and so adding variables and constraints for the other upper bounds is unnecessary, and could potentially slow down the solving process of the SP. Therefore, post-processing of the component bound sequences, i.e. removing redundant bounds, is advisable.

5.2.3 Branching with Multiple Subproblems

The component bound branching rule as 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). There are, however, instances consists of at least two distinct (aggregated) 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 blocks have 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 branch in the master by imposing bounds on the original variables within the SP. The main difference is, that Vanderbeck enforces these bounds as hard constraints, while we only add soft constraints. One could say, that by applying Vanderbeck branching, one literally partitions the solution space of the original variables. As a result, when we find the optimal solution $\boldsymbol{x}^*$ to the IP in some node in the branch-and-bound tree of the RMP, we know that $\boldsymbol{x}^*$ satisfies all the component bounds imposed by the branching decisions along the path from the root to this node. In contrast, in our approach both columns satisfying and violating the component bounds can be generated.

While the component bound branching rule might be easier to implement, branching using Vanderbeck's generic scheme has a significant advantage: It only requires tightening the bounds of the original variables in the SP, whereas our approach introduces new variables and constraints. This is a significant advantage, since the structure of the pricing problem does not change. Many dynamic programming solvers for specific IPs can handle changing variable bounds, and can therefore continue to generate columns efficiently, even after branching. This is unfortunately not the case for our approach, where the pricing problem changes with each branching decision, forcing us to fall back to a generic IP solver.

Chapter 6

Master Constraints without corresponding Original Problem Constraints

Besides implementing the component bound branching rule (Chapter 5), a major goal of this thesis is to allow future GCG-developers to easily create new branching rules and separators within the framework (Section 4.2). In the current state of GCG this is very limited: branching rules, for example, must either produce branching decisions formulated in the original problem, which can then be Dantzig-Wolfe reformulated and added to the master and pricing problem, as seen when branching on original variables (Section 3.5.1), or they must only produce constraints for the master problem without requiring modification 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. Such significant changes, in the case of Vanderbeck's branching, would for example be applying and removing the components bounds in the pricing problem when a B&B-node is entered or left. Also, our proposed component bound branching rule (Chapter 5) as well as separators using the master problem (Section 3.6.2) would require such changes. The reason for this is that GCG itself does not provide a way to impose constraints in the master problem that require modification to at least one *SP*, where the master constraints and induced pricing problem modifications are not product of a Dantzig-Wolfe reformulation of the original problem, i.e. do not necessarily have a counterpart in the original problem. In this chapter we will specify the notation of such constraints, which we will henceforth refer to as **generic mastercuts**, present our integration of these constraints into the GCG framework as part of a new interface, and finally show how we can apply dual value stabilization to these constraints.

Let us first define the concept of a generic mastercut, the concept that 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.*

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

*Every pricing modification contains at least one mandatory variable y that has 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. For this reason, y is known as the **coefficient variable** of the pricing modification. All additional variables will take on an auxiliary role in the pricing problem and have an objective coefficient of zero.*

// TODO: which places do we add them? // TODO: how can we use them in branching rules?

6.1 Mastervariable Synchronization across the entire B&B-Tree

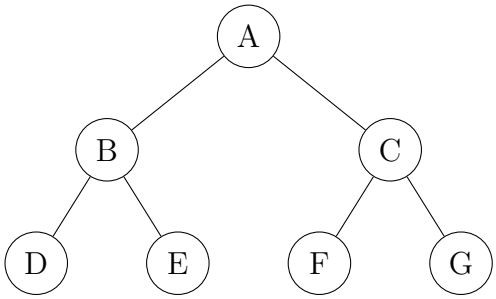

Figure 6.1: An exemplary B&B-tree, created by the component bound branching rule, where the lexicographic order of the nodes resembles the order in which they were created.

Generic mastercuts that are used as branching decisions need to be aware of all columns that currently exist in the *RMP*, e.g. in the case of component bound branching (Chapter 5), all columns that satisfy the branching decision must have

a coefficient of 1 for the mastercut. This can be easily achieved when creating the branching decision, i.e. when creating the mastercut, and could be done automatically when a new column is generated in the subtree of the node where the branching decision was made. However, consider the following case in a branch-and-bound tree generated with the component bound branching rule (Figure 6.1): we currently process node F in the B&B-tree and within that node generate a new column q'. Afterwards, we deactivate node F and activate node D. Now it could be that $x_{q'}$ satisfies the component bounds imposed in D, i.e. q' should have a coefficient of 1 in the mastercut of D. However, since the column was generated in F, it is not known to D, which might invalidate the mastercut of D.

To prevent this, we must make the branching decisions aware of these columns such that they can update their coefficients accordingly. In particular, we would like to synchronize newly generated master variables across the entire B&B-tree lazily, i.e. only when a node is being activated. Furthermore, in `GCG` master variables that are deemed unnecessary can be removed from the master problem. Deleted variables should not have to be synchronized.

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

6.1.1 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 being 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 in the meantime new columns have been added to the *RMP*, this counter *might* increase. If so, the coefficient for the new columns will be determined and set in the *RMP*.

Unfortunately, this approach is not complete, as not all new columns are necessarily detected. For instance, consider the case where 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. And for this reason, the coefficient for the new column would not be set in the mastercut. This could lead to invalid mastercuts, as described in the beginning of this section.

Furthermore, this approach is not very developer friendly. In our opinion, the developer of a branching rule should not have to deal with knowing when and where

which columns are being generated and deleted. This should be handled by the framework, and the developer should only have to provide an implementation for an interface method responsible for setting the coefficient of any column, regardless of whether it was generated in the current node or not.

6.1.2 History Tracking Approach

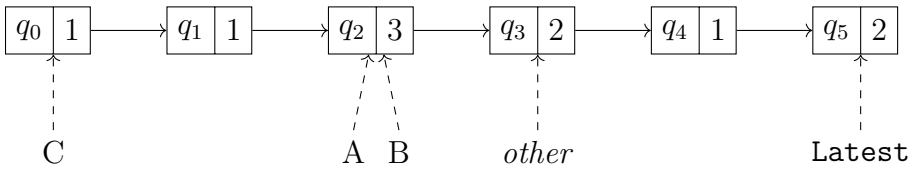

Figure 6.2: Reference-Counted Linked List of the History of Columns added to the *RMP*, with external references drawn dashed from below, e.g. those from the B&B-tree nodes A, B, and C. Each element holds a reference to the master variable belonging to column q_i, as well as the number of references to itself.

We propose a highly efficient approach to lazily notify all nodes in the B&B-tree of new columns that have been 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 the order in which they were generated. Each node in the B&B-tree holds its own external reference to this construct. In particular, the specific element in the list that such a B&B-tree points to signifies the last column that was in the *RMP* when the node was last active. All further variables, i.e. the elements next in the list, are new columns that have been generated elsewhere in the list. Additionally, we hold one external reference to the tail of the list, which is the last column that was generated. This construct is illustrated in Figure 6.2. Since the linked list tracks which variables where created when, we will henceforth refer to this list as the `varhistory`. We will now describe how this construct is used to synchronize the master variables across the entire B&B-tree.

Let us consider a B&B-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 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 therefore created two child nodes D and E. Both nodes will be created using all columns that are currently in the *RMP*, i.e. $q_i, i \in \{0, 5\}$. For this reason, we use the `Latest` pointer initialize the reference to the `varhistory` of D and E (Figure 6.3).

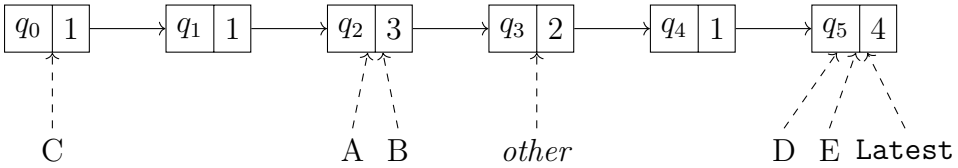

Figure 6.3: `varhistory` after creating child nodes D and E of node B.

Continuing this scenario, let `GCG` now deem the column q_2 to be 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 simply mark it as deleted. Next, we would like to solve node C. For this, we must deactivate node B, and activate node A. 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.

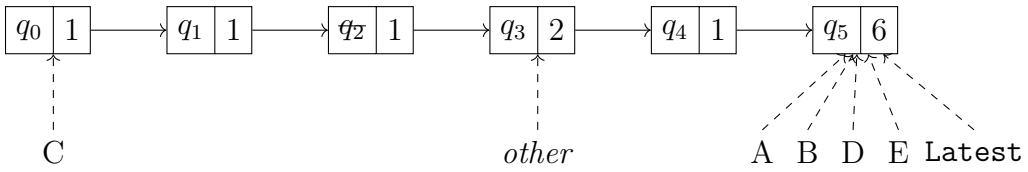

Figure 6.4: `varhistory` after deletion of column q_2 and deactivation of node B.

Finally, we 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 now forward the pointer of C one by one until there is no next element, i.e. until we reach the `Latest` pointer. Each time we forward the pointer, if the variable q_i is not marked as deleted, we pass it to the branching rule, which can then set the coefficient of q_i in the 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 are updated. As soon as this reference count reaches zero, the element can be safely freed. This way, we ensure that only variables that are actually needed, i.e. those that still need to be synchronized across the entire B&B-tree are kept in memory. This is illustrated in Figure 6.5.

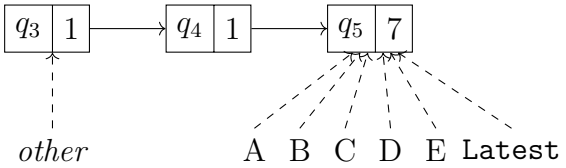

Figure 6.5: `varhistory` after activation of node C.

Assume in node C we now generate a new column q_6. We add this column to the `varhistory` by allocating a new list element, setting its reference count to 1, and 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.

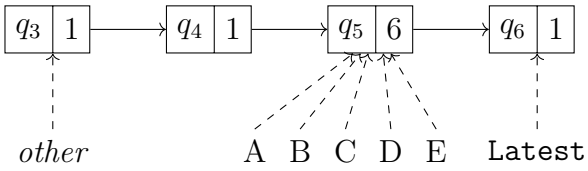

Figure 6.6: `varhistory` after generating column q_6 in node C.

This approach is highly efficient, as we only need to update the reference count of the elements in the list, and we only need to pass the variables to the branching rule that are actually new. The deletion of variables is also not of an issue here, as we simply do not pass them to the branching rules. As soon as all external references have seen some variable q_i, i.e. its reference count reaches zero, we can then automatically free the memory of the element in the list holding q_i. In this way, we can also minimize the memory footprint of the `varhistory` construct.

And as a final note, this approach is not limited for synchronization of master variables across the B&B-tree, 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 master separators up-to-date (Section 3.6.2) . For more details on this, we refer to the thesis of Chantal Reinartz Groba [**TODO**].

6.1.3 History Tracking using Unrolled Linked Lists Approach

The approach described above has one optimization opportunity: as the elements of the `varhistory` might be allocated in completely different memory locations, the traversal of the list, i.e. during forwarding, would lead to poor cache utilization, resulting in many memory accesses. However, storing the entire list in a contiguous

memory block, growing to the right, could lead to expensive reallocation and copying of the entire list, if no more space to the right is available.

We can combine these two ideas by unrolling our linked list into blocks storing a fixed amount of columns, where now each block has its own reference count. Once again, these blocks are linked together, forming a list of blocks. The references to the `varhistory` do not point to individual elements, but rather to the blocks, in addition to holding an offset within the block. This way, each reference still refers to some q_i, thus enabling us to forward a pointer one column at a time. Whenever a new variable is generated, we only allocate a new block if the current tail block is full. Otherwise, we simply add the new variable to the tail block. This concept is illustrated in Figure 6.7.

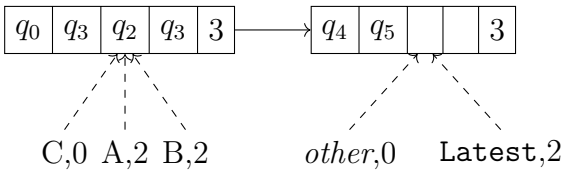

Figure 6.7: `varhistory` of Figure 6.2 unrolled into blocks of size 4.

This way, we can improve cache locality, and as a bonus also reduce the total number of memory allocation and deallocation operations, though from an outside perspective the fundamental operations of the `varhistory` construct remain the same.

6.2 Dual Value Stabilization for Generic Mastercuts

Chapter 7

Implementation

7.1 Generic Mastercuts

7.2 Mastervariable Synchronization

7.3 Component Bound Branching

Chapter 8

Evaluation

8.1 Testset of Instances

8.2 Validation of Correct Implementation

8.3 Practical Comparison to Vanderbeck's Generic Branching

Chapter 9

Conclusion

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

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

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

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

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

[8] 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.

[9] Tobias Achterberg, Thorsten Koch, and Alexander Martin. "Branching rules revisited". In: Operations Research Letters 33.1 (2005), pp. 42–54.