

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 22, 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

1	Introduction	5
2	Preliminaries	7
2.1	Polyhedron Representation	7
2.2	Primal Simplex Algorithm	11
3	Column Generation and Branch-and-Price	13
3.1	Column Generation	13
3.1.1	Farkas Pricing	14
3.1.2	Reduced Cost Pricing	15
3.1.3	Column Generation Algorithm	16
3.2	Dantzig-Wolfe Reformulation	16
3.3	Dantzig-Wolfe Reformulation for Integer Programs	18
3.3.1	Convexification	19
3.3.2	Discretization	20
3.4	Several and Identical Subproblems	21
3.5	Branch-and-Price	24
3.5.1	Branching on the Original Variables	25
3.5.2	Branching on the Master Variables	26
3.6	Branch-Price-and-Cut	30
3.6.1	Separators using the Original Formulation	31
3.6.2	Separators using the Master Problem	31
3.7	Dual Value Stabilization	32
4	SCIP Optimization Suite	33
4.1	SCIP	33
4.2	GCG	33
5	Component Bound Branching	35
5.1	Overview of the branching scheme	35
5.2	Separation Procedure	38

5.2.1	Choice of Component Bounds	39
5.2.2	Post-processing of Component Bound Sequences	40
5.2.3	Branching with Multiple Subproblems	41
5.3	Comparison to Vanderbeck's Generic Branching	41
6	Master Constraints without corresponding Original Problem Constraints	43
6.1	Mastervariable Synchronization across the entire B&B-Tree	44
6.1.1	Current Approach used by the Implementation of Vanderbeck's Generic Branching	45
6.1.2	History Tracking Approach	46
6.1.3	History Tracking using Unrolled Linked Lists Approach	48
6.2	Dual Value Stabilization for Generic Mastercuts	49
7	Implementation	51
7.1	Generic Mastercuts	51
7.2	Mastervariable Synchronization	51
7.3	Component Bound Branching	51
8	Evaluation	53
8.1	Testset of Instances	53
8.2	Validation of Correct Implementation	53
8.3	Practical Comparison to Vanderbeck's Generic Branching	53
9	Conclusion	55

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

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

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

$$\text{conv}(\mathbf{x}_1, \dots, \mathbf{x}_k) := \left\{ \sum_{i=1}^k \alpha_i \mathbf{x}_i \mid \sum_{i=1}^k \alpha_i = 1 \wedge \forall i \in \{1, \dots, k\}. \alpha_i \geq 0 \right\}$$

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

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

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

Definition 2.4. Let \mathcal{P} be a convex set. A vector $\mathbf{r} \in \mathbb{R}_0^n \setminus \{0\}$ is a **ray** of \mathcal{P} iff $\forall \mathbf{x} \in \mathcal{P}. \forall \beta \in \mathbb{R}_+. \mathbf{x} + \beta \mathbf{r} \in \mathcal{P}$.

The cone of rays $\mathbf{r}_1, \dots, \mathbf{r}_k \in \mathbb{R}_+^n$ is denoted as:

$$\text{cone}(\mathbf{r}_1, \dots, \mathbf{r}_k) := \left\{ \sum_{i=1}^k \alpha_i \mathbf{r}_i \mid \forall i \in \{1, \dots, k\}. \alpha_i \geq 0 \right\}$$

Definition 2.5. A ray \mathbf{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 \mathbf{r} , i.e.

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

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 $\mathbf{f} \in \mathbb{R}^n$ and a scalar $f \in \mathbb{R}$ as $\mathcal{H} = \{\mathbf{x} \mid \mathbf{f}^\top \mathbf{x} = f\}$.

Corollary 2.2. Any hyperplane is a convex set.

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

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

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

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

Corollary 2.3. Any halfspace is a convex set.

Proof. Let $\mathcal{H}^+ = \{\mathbf{x} \mid \mathbf{f}^\top \mathbf{x} > f\}$ be an open halfspace (analogous for $\mathcal{H}^- = \{\mathbf{x} \mid \mathbf{f}^\top \mathbf{x} < f\}$, and for the closed halfspaces). Let $k \in \mathbb{N}$, $\mathbf{x}_1, \dots, \mathbf{x}_k \in \mathcal{H}^+$. For any $\alpha_1, \dots, \alpha_k \in \mathbb{R}_+$ with $\sum_{i=1}^k \alpha_i = 1$:

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

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

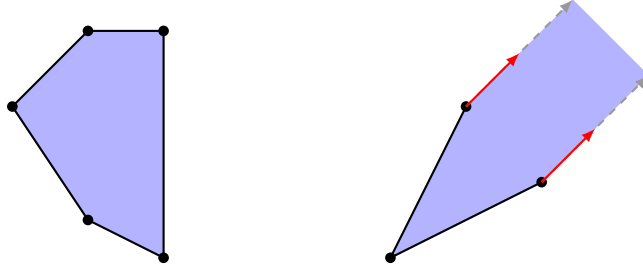


Figure 2.1: Illustration of the Minkowski-Weyl theorem. The left figure shows a fully bounded polyhedron represented by its extreme points. Unbounded polyhedra, such as the one on the right, require extreme rays, shown in red, for a complete description.

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

Corollary 2.4. A polyhedron is a convex set.

Proof. Follows from Corollaries 2.1 and 2.3. □

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

$$P \oplus Q := \{\mathbf{p} + \mathbf{q} \mid \mathbf{p} \in P \wedge \mathbf{q} \in Q\}$$

Theorem 2.1 (Minkowski-Weyl [4]). 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 $\mathbf{A} \in \mathbb{R}^{m \times n}$ and some vector $\mathbf{b} \in \mathbb{R}^m$ such that $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$
2. There exist finitely many vectors $\mathbf{v}_1, \dots, \mathbf{v}_s \in \mathbb{R}^n$ and finitely many vectors $\mathbf{r}_1, \dots, \mathbf{r}_t \in \mathbb{R}_+^n$, such that $P = \text{conv}(\mathbf{v}_1, \dots, \mathbf{v}_s) \oplus \text{cone}(\mathbf{r}_1, \dots, \mathbf{r}_t)$

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 $\{\mathbf{x}_p\}_{p \in P}$ and extreme rays $\{\mathbf{x}_r\}_{r \in R}$, using hyperplanes. Here, the sets P, R are used to index the extreme points and extreme rays, respectively.

Theorem 2.2 (Nemhauser-Wolsey [5]). *Consider the polyhedron $\mathcal{P} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Q}\mathbf{x} \geq \mathbf{b}\}$ with full row rank matrix $\mathbf{Q} \in \mathbb{R}^{m \times n}$, i.e., $\text{rank}(\mathbf{Q}) = m \leq n \wedge \mathcal{P} \neq \emptyset$. An equivalent description of \mathcal{P} using its extreme points $\{\mathbf{x}_p\}_{p \in P}$ and extreme rays $\{\mathbf{x}_r\}_{r \in R}$ is:*

$$\mathcal{P} = \left\{ \mathbf{x} \in \mathbb{R}^n \left| \begin{array}{ll} \sum_{p \in P} \mathbf{x}_p \lambda_p + \sum_{r \in R} \mathbf{x}_r \lambda_r = \mathbf{x} \\ \sum_{p \in P} \lambda_p = 1 \\ \lambda_p \geq 0 \quad \forall p \in P \\ \lambda_r \geq 0 \quad \forall r \in R \end{array} \right. \right\} \quad (2.1)$$

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

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

Theorem 2.3 (Nemhauser-Wolsey [5]). *Consider the polyhedron $\mathcal{P} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Q}\mathbf{x} \geq \mathbf{b}\}$ with full row rank matrix $\mathbf{Q} \in \mathbb{R}^{m \times n}$, i.e., $\text{rank}(\mathbf{Q}) = m \leq n \wedge \mathcal{P} \neq \emptyset$. Have $\mathcal{Q} := \mathcal{P} \cap \mathbb{Z}^n \neq \emptyset$ be the integer hull of \mathcal{P} .*

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

$$\mathcal{Q} = \left\{ \mathbf{x} \in \mathbb{Z}^n \left| \begin{array}{ll} \sum_{p \in \tilde{P}} \mathbf{x}_p \lambda_p + \sum_{r \in R} \mathbf{x}_r \lambda_r = \mathbf{x} \\ \sum_{p \in \tilde{P}} \lambda_p = 1 \\ \lambda_p \in \{0, 1\} \quad \forall p \in \tilde{P} \\ \lambda_r \in \mathbb{Z}_+ \quad \forall r \in R \end{array} \right. \right\} \quad (2.2)$$

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

2.2 Primal Simplex Algorithm

Consider the following linear program in standard form:

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s. t.} \quad & \mathbf{Ax} = \mathbf{b} \quad [\boldsymbol{\pi}] \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2.3}$$

The primal simplex algorithm [6] finds an optimal solution by moving from one extreme point of the polyhedron to the next, therefore always remaining feasible. A central aspect of this algorithm is the sufficient optimality condition. For a basic solution $\mathbf{X} = [\mathbf{x}_\mathcal{B}, \mathbf{x}_\mathcal{N}]$ at a given extreme point to be optimal, the reduced costs $\bar{c}_j := c_j - \boldsymbol{\pi}^\top \mathbf{a}_j$ for $j \in \mathcal{N}$ must be non-negative.

This sufficient optimality condition leads to the **pricing problem**, which either verifies the optimality of the current basic solution or identifies the non-basic variable x_l , $l \in \mathcal{N}$, with the least reduced cost ($\bar{c}_l < 0$) to be introduced into the basis next, following Dantzig's rule [6, 7]. Formally, this is expressed:

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

or as the linear program:

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

Solving the pricing problem is integral 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 (\mathbf{x}, z)

```
1 loop
2    $\boldsymbol{\pi}^\top \leftarrow \mathbf{c}_\mathcal{B}^\top \mathbf{A}_\mathcal{B}^{-1}; \bar{\mathbf{b}} \leftarrow \mathbf{A}_\mathcal{B}^{-1} \mathbf{b};$ 
3    $\bar{c}_j \leftarrow c_j - \boldsymbol{\pi}^\top \mathbf{a}_j; \quad \forall j \in \mathcal{N}$ 
4    $l \leftarrow \arg \min_{j \in \mathcal{N}} \bar{c}_j; \bar{c}(\boldsymbol{\pi}) \leftarrow \bar{c}_l;$ 
5   if  $\bar{c}(\boldsymbol{\pi}) \geq 0$  then
6     | return  $([\bar{\mathbf{b}}, \mathbf{0}], \mathbf{c}_\mathcal{B}^\top \mathbf{x}_\mathcal{B})$  by optimality
7   end
8    $\bar{\mathbf{a}}_l \leftarrow \mathbf{A}_\mathcal{B}^{-1} \mathbf{a}_l;$ 
9   if  $\bar{\mathbf{a}}_l \leq \mathbf{0}$  then
10    | return None by unboundedness
11  end
12   $s \leftarrow \arg \min_{i \in \{1, \dots, m\}} \frac{\bar{b}_i}{\bar{a}_{il}}; x_l \leftarrow \frac{\bar{b}_s}{\bar{a}_{sl}}; \mathcal{B} \leftarrow \mathcal{B} \cup \{l\} \subseteq \{s\}; \mathcal{N} \leftarrow \mathcal{N} \cup \{s\} \subseteq \{l\};$ 
```

Chapter 3

Column Generation and Branch-and-Price

3.1 Column Generation

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

$$\begin{aligned} z_{MP}^* = \min \quad & \sum_{x \in \mathcal{X}} c_x \lambda_x \\ \text{s. t.} \quad & \sum_{x \in \mathcal{X}} \mathbf{a}_x \lambda_x \geq \mathbf{b} \quad [\boldsymbol{\pi}] \\ & \lambda_x \geq 0 \quad \forall \mathbf{x} \in \mathcal{X} \end{aligned} \tag{3.1}$$

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

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

$$\begin{aligned} z_{RMP}^* = \min \quad & \sum_{x \in \mathcal{X}'} c_x \lambda_x \\ \text{s. t.} \quad & \sum_{x \in \mathcal{X}'} \mathbf{a}_x \lambda_x \geq \mathbf{b} \quad [\boldsymbol{\pi}] \\ & \lambda_x \geq 0 \quad \forall \mathbf{x} \in \mathcal{X}' \end{aligned} \tag{3.2}$$

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

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

3.1.1 Farkas Pricing

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

Theorem 3.1 (Farkas' lemma [8]). *Given $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$, then exactly one of the following statements holds:*

1. $\exists \mathbf{x} \in \mathbb{R}_+^n. \mathbf{Ax} \geq \mathbf{b}$
2. $\exists \boldsymbol{\pi} \in \mathbb{R}^m. \boldsymbol{\pi}^\top \mathbf{A} \leq \mathbf{0} \wedge \boldsymbol{\pi}^\top \mathbf{b} > 0$

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

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

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

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

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

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

$$F(\boldsymbol{\pi}) = \min_{\mathbf{x} \in \mathcal{X}} -\boldsymbol{\pi}^\top \mathbf{a}_x \quad (3.5)$$

We can add all solutions \mathbf{x} with a value of $F(\boldsymbol{\pi}) < 0$ to $\mathcal{X}'' := \mathcal{X}' \cup \{\mathbf{x}_i\}$, adding the corresponding column $\begin{bmatrix} 0 \\ \mathbf{a}_x \end{bmatrix}$ to the problem, thus turning any infeasible RMP feasible.

3.1.2 Reduced Cost Pricing

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

$$\bar{c}(\boldsymbol{\pi}) = \min_{\mathbf{x} \in \mathcal{X}} c_{\mathbf{x}} - \boldsymbol{\pi}^\top \mathbf{a}_{\mathbf{x}} \quad (3.6)$$

Note that due to the optimality of RMP , the reduced costs of all variables $\mathbf{x} \in \mathcal{X}'$ are already non-negative. If $\bar{c}(\boldsymbol{\pi}) \geq 0$, we have also proven the optimality of the current solution for MP . Otherwise, if $\bar{c}(\boldsymbol{\pi}) < 0$, there exists some $\mathbf{x} \in \mathcal{X} \setminus \mathcal{X}'$ with $\bar{c}(\boldsymbol{\pi}) = c_{\mathbf{x}} - \boldsymbol{\pi}^\top \mathbf{a}_{\mathbf{x}} < 0$. Similar to how the primal simplex algorithm would swap this variable into the basis, during column generation we add the corresponding column $\begin{bmatrix} c_{\mathbf{x}} \\ \mathbf{a}_{\mathbf{x}} \end{bmatrix}$ to RMP . This process ensures that RMP remains feasible.

Algorithm 3.1: Column Generation Algorithm

Input: RMP with subset $\mathcal{X}' \subseteq \mathcal{X}$, $RCP\text{-}SP$, $FP\text{-}SP$

Output: Optimal Solution $(\boldsymbol{\lambda}, z)$ for the MP

```

1 while IsInfeasible( $RMP$ ) do
2    $(\text{None}, \boldsymbol{\pi}) \leftarrow \text{Solve}(RMP)$ ;
3    $(\mathbf{x}, F(\boldsymbol{\pi})) \leftarrow \text{Solve}(FP\text{-}SP, \boldsymbol{\pi})$ ;
4   if  $F(\boldsymbol{\pi}) \geq 0$  then
5     return None by MP infeasibility
6   end
7    $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\mathbf{x}\}$ ;
8    $\mathbf{A} \leftarrow [\mathbf{A} \quad \mathbf{a}_{\mathbf{x}}]$ ;
9 end
10 loop
11    $(\boldsymbol{\lambda}_{RMP}, \boldsymbol{\pi}) \leftarrow \text{Solve}(RMP)$ ;
12    $(\mathbf{x}, \bar{c}(\boldsymbol{\pi})) \leftarrow \text{Solve}(RCP\text{-}SP, \boldsymbol{\pi})$ ;
13   if  $\bar{c}(\boldsymbol{\pi}) \geq 0$  then
14     return  $(\boldsymbol{\lambda}_{RMP}, \mathbf{c}_{\mathcal{B}}^\top \mathbf{x}_{\mathcal{B}})$  by optimality
15   end
16    $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\mathbf{x}\}$ ;
17    $\mathbf{A} \leftarrow [\mathbf{A} \quad \mathbf{a}_{\mathbf{x}}]$ ;

```

3.1.3 Column Generation Algorithm

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

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

3.2 Dantzig-Wolfe Reformulation

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

$$\begin{aligned} z_{LP}^* = \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s. t.} \quad & \mathbf{Ax} \geq \mathbf{b} \quad [\boldsymbol{\sigma}_b] \\ & \mathbf{Dx} \geq \mathbf{d} \quad [\boldsymbol{\sigma}_d] \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.7}$$

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

$$\begin{aligned} \mathcal{A} &:= \{\mathbf{x} \geq \mathbf{0} \mid \mathbf{Ax} \geq \mathbf{b}\} \neq \emptyset \\ \mathcal{D} &:= \{\mathbf{x} \geq \mathbf{0} \mid \mathbf{Dx} \geq \mathbf{d}\} \neq \emptyset \end{aligned} \tag{3.8}$$

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

and extreme rays using:

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

Using the following shorthand notations:

$$\begin{aligned}
c_p &:= \mathbf{c}^\top \mathbf{x}_p & c_r &:= \mathbf{c}^\top \mathbf{x}_r \\
\mathbf{a}_p &:= \mathbf{A} \mathbf{x}_p & \mathbf{a}_r &:= \mathbf{A} \mathbf{x}_r
\end{aligned} \tag{3.10}$$

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

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

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

Since the number of extreme points and extreme rays of \mathcal{D} might be enormous, solving the *MP* directly is typically infeasible. Instead, we can 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 formulate this pricing problem as follows:

$$\begin{aligned}
z_{SP}^* = \min \quad & (\mathbf{c}^\top - \boldsymbol{\pi}_b^\top \mathbf{A}) \mathbf{x} - \pi_0 \\
\text{s. t.} \quad & \mathbf{D} \mathbf{x} \geq \mathbf{d} \quad [\boldsymbol{\pi}_d] \\
& \mathbf{x} \geq \mathbf{0}
\end{aligned} \tag{3.12}$$

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

- If $-\infty < z_{SP}^* < 0$, \mathbf{x}^* is an extreme point $\mathbf{x}_p, p \in P \setminus P'$, and we add column $\begin{bmatrix} \mathbf{c}^\top \mathbf{x}^* \\ \mathbf{A} \mathbf{x}^* \\ 1 \end{bmatrix}$ to *RMP*.
- If $z_{SP}^* = -\infty$, \mathbf{x}^* is an extreme ray $\mathbf{x}_r, r \in R \setminus R'$, and we add column $\begin{bmatrix} \mathbf{c}^\top \mathbf{x}^* \\ \mathbf{A} \mathbf{x}^* \\ 0 \end{bmatrix}$ to *RMP*.
- If $z_{SP}^* \geq 0$, there exists no improving column for *RMP*, and the column generation algorithm terminates.

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

3.3 Dantzig-Wolfe Reformulation for Integer Programs

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

Consider the following integer program:

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

Again, we group the constraints into two sets:

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

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:

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

There are two ways to proceed from here. The straightforward approach, called **Convexification**, follows the method seen in the Dantzig-Wolfe reformulation of linear programs, retaining the integrality constraints on \mathbf{x} in both the master and pricing problem. Alternatively, in **Discretization**, we modify our approach slightly, adding integrality constraints to the master variables to ensure the integrality of the original variables.

3.3.1 Convexification

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

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

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

$$\begin{aligned}
z_{SP}^* = \min \quad & (\mathbf{c}^\top - \boldsymbol{\pi}_b^\top \mathbf{A}) \mathbf{x} - \pi_0 \\
\text{s. t.} \quad & \mathbf{D}\mathbf{x} \geq \mathbf{d} \quad [\boldsymbol{\pi}_d] \\
& \mathbf{x} \in \mathbb{Z}_+^n
\end{aligned} \tag{3.17}$$

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

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

λ solutions to the *MP* might lead to fractional \mathbf{x} solutions. In this case, we must 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 adaptation of the Nemhauser-Wolsey Theorem to integer polyhedra (Theorem 2.3) to reformulate the polyhedron \mathcal{D} , yielding the following master problem:

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

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., it might include interior points, we must find a pricing problem that can generate not only extreme

points (and rays) of \mathcal{D} but also interior points. This, however, is not trivial, since mathematical optimization typically focuses on finding the most optimal solutions, i.e., the extreme points. We can postpone this concern for now and use the same pricing problem as in the convexification approach:

$$\begin{aligned} z_{SP}^* = \min \quad & (\mathbf{c}^\top - \boldsymbol{\pi}_b^\top \mathbf{A}) \mathbf{x} - \pi_0 \\ \text{s. t.} \quad & \mathbf{D}\mathbf{x} \geq \mathbf{d} \quad [\boldsymbol{\pi}_d] \\ & \mathbf{x} \in \mathbb{Z}_+^n \end{aligned} \quad (3.19)$$

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

3.4 Several and Identical Subproblems

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

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

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

$$z_{SP^k}^* = \min_{\mathbf{x}^k \in \mathcal{X}^k} c_{\mathbf{x}^k} - \boldsymbol{\pi}^\top \mathbf{a}_{\mathbf{x}^k} \quad (3.21)$$

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

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

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

In many applications, the coefficient matrix \mathbf{D} has a block diagonal structure:

$$\mathbf{D} = \begin{bmatrix} \mathbf{D}^1 & & \\ & \ddots & \\ & & \mathbf{D}^{|K|} \end{bmatrix} \quad \text{and} \quad \mathbf{d} = \begin{bmatrix} \mathbf{d}^1 \\ \vdots \\ \mathbf{d}^{|K|} \end{bmatrix} \quad (3.23)$$

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

$$\begin{aligned} z_{MP}^* = \min \quad & \sum_{k \in K} \sum_{p \in P^k} c_p^k \lambda_p^k + \sum_{k \in K} \sum_{r \in R^k} c_r^k \lambda_r^k \\ \text{s. t.} \quad & \sum_{k \in K} \sum_{p \in P^k} \mathbf{a}_p^k \lambda_p^k + \sum_{k \in K} \sum_{r \in R^k} \mathbf{a}_r^k \lambda_r^k \geq \mathbf{b} \quad [\boldsymbol{\pi}_b] \\ & \sum_{p \in P^k} \lambda_p^k = 1 \quad [\pi_0^k] \forall k \in K \\ & \lambda_p^k \geq 0 \quad \forall k \in K, \forall p \in P^k \\ & \lambda_r^k \geq 0 \quad \forall k \in K, \forall r \in R^k \\ & \sum_{p \in P^k} \mathbf{x}_p^k \lambda_p^k + \sum_{r \in R^k} \mathbf{x}_r^k \lambda_r^k = \mathbf{x}^k \geq \mathbf{0} \quad \forall k \in K \end{aligned} \quad (3.24)$$

Each subproblem SP^k is given by:

$$\begin{aligned} z_{SP^k}^* = \min \quad & (\mathbf{c}^{k\top} - \boldsymbol{\pi}_b^\top \mathbf{A}^k) \mathbf{x}^k - \pi_0^k \\ \text{s. t.} \quad & \mathbf{D}^k \mathbf{x}^k \geq \mathbf{d}^k \quad [\boldsymbol{\pi}_d^k] \\ & \mathbf{x}^k \geq \mathbf{0} \end{aligned} \quad (3.25)$$

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

$$\lambda_p := \sum_{k \in K} \lambda_p^k, \quad \forall p \in P \quad \text{and} \quad \lambda_r := \sum_{k \in K} \lambda_r^k, \quad \forall r \in R \quad (3.26)$$

Substituting these aggregated variables in MP yields:

$$z_{MP}^* = \min \sum_{p \in P} c_p \lambda_p + \sum_{r \in R^k} c_r \lambda_r \quad (3.27a)$$

$$\text{s. t. } \sum_{p \in P} \mathbf{a}_p \lambda_p + \sum_{r \in R^k} \mathbf{a}_r \lambda_r \geq \mathbf{b} \quad [\boldsymbol{\pi}_b] \quad (3.27b)$$

$$\sum_{p \in P} \lambda_p = |K| \quad [\pi_{agg}] \quad (3.27c)$$

$$\lambda_p \geq 0 \quad \forall p \in P \quad (3.27d)$$

$$\lambda_r \geq 0 \quad \forall r \in R \quad (3.27e)$$

$$\sum_{k \in K} \lambda_p^k = \lambda_p \quad \forall p \in P \quad (3.27f)$$

$$\sum_{k \in K} \lambda_r^k = \lambda_r \quad \forall r \in R \quad (3.27g)$$

$$\sum_{p \in P} \lambda_p^k = 1 \quad \forall k \in K \quad (3.27h)$$

$$\lambda_p^k \geq 0 \quad \forall k \in K, \forall p \in P \quad (3.27i)$$

$$\lambda_r^k \geq 0 \quad \forall k \in K, \forall r \in R \quad (3.27j)$$

$$\sum_{p \in P} \mathbf{x}_p \lambda_p^k + \sum_{r \in R} \mathbf{x}_r \lambda_r^k = \mathbf{x}^k \geq \mathbf{0} \quad \forall k \in K \quad (3.27k)$$

Columns for this MP are generated by the following subproblem:

$$\begin{aligned} z_{SP^{agg}}^* = \min \quad & (\mathbf{c}^\top - \boldsymbol{\pi}_b^\top \mathbf{A}) \mathbf{x} - \pi_{agg} \\ \text{s. t.} \quad & \mathbf{D}\mathbf{x} \geq \mathbf{d} \quad [\boldsymbol{\pi}_d] \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \quad (3.28)$$

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

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

$$z_{MP}^* = \min \sum_{p \in P} c_p \lambda_p + \sum_{r \in R^k} c_r \lambda_r \quad (3.29a)$$

$$\text{s. t. } \sum_{p \in P} \mathbf{a}_p \lambda_p + \sum_{r \in R^k} \mathbf{a}_r \lambda_r \geq \mathbf{b} \quad [\boldsymbol{\pi}_b] \quad (3.29b)$$

$$\sum_{p \in P} \lambda_p = |K| \quad [\pi_{agg}] \quad (3.29c)$$

$$\lambda_p \in \mathbb{Z}_+ \quad \forall p \in P \quad (3.29d)$$

$$\lambda_r \in \mathbb{Z}_+ \quad \forall r \in R \quad (3.29e)$$

$$\sum_{k \in K} \lambda_p^k = \lambda_p \quad \forall p \in P \quad (3.29f)$$

$$\sum_{k \in K} \lambda_r^k = \lambda_r \quad \forall r \in R \quad (3.29g)$$

$$\sum_{p \in P} \lambda_p^k = 1 \quad \forall k \in K \quad (3.29h)$$

$$\lambda_p^k \in \mathbb{Z}_+ \quad \forall k \in K, \forall p \in P \quad (3.29i)$$

$$\lambda_r^k \in \mathbb{Z}_+ \quad \forall k \in K, \forall r \in R \quad (3.29j)$$

$$\sum_{p \in P} \mathbf{x}_p \lambda_p^k + \sum_{r \in R} \mathbf{x}_r \lambda_r^k = \mathbf{x}^k \in \mathbb{Z}_+^n \quad \forall k \in K \quad (3.29k)$$

In Section 3.3.2, we have observed that the integrality constraints on the original variables \mathbf{x}^k are already enforced by ensuring the integrality of the disaggregated master variables λ_p^k and λ_r^k . 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 possible to have both identical and differing subproblems in the same MP . In this case, we introduce classes C of identical subproblems, use one column generator per class, and aggregate the variables within each class.

3.5 Branch-and-Price

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

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

In the context of column generation for integer master programs, we follow a similar approach: first, we relax the integrality constraints of the master problem, allowing us to solve the relaxation using column generation to optimality. Then, we check if the integrality conditions are satisfied. If not, we must cut off the fractional solution by branching. Combining branching with column generation results in **branch-and-price**.

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

3.5.1 Branching on the Original Variables

Assume we have a fractional solution \mathbf{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 . We can cut off this fractional solution for example 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 *RMP* should be guaranteed to only use columns that satisfy the branching decision, and during the solving process, the pricing problems should only be able to generate such columns.

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

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

3.5.1.1 Branching in the Master Problem

Recall that the *MP* includes the following constraint:

$$\sum_{p \in P} \mathbf{x}_p \lambda_p + \sum_{r \in R} \mathbf{x}_r \lambda_r = \mathbf{x} \in \mathbb{Z}_+^n \quad (3.30)$$

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

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

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

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

3.5.1.2 Branching in the Pricing Problem

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

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

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

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

3.5.2 Branching on the Master Variables

Let $Q := \ddot{P} \cup R$. Assume our master solution $\boldsymbol{\lambda}_{RMP}^*$ is fractional, i.e., $\lambda_q^* \notin \mathbb{Z}$ for at least one $q \in Q$. Unfortunately, dichotomous branching on a such a single master variable λ_q is very weak: Assume $q \in \ddot{P}$. 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 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 in RMP , for which the following holds:

$$\sum_{q \in Q'} \lambda_q^* =: K \notin \mathbb{Z} \quad (3.35)$$

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

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

The corresponding subproblem must be adapted to ensure the validity of the branching decision in the master:

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

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

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

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

3.5.2.1 Vanderbeck's Generic Branching Scheme

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

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

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

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

$$S := \{(x_{i,1}, \eta_1, v_1), \dots, (x_{j,k}, \eta_k, v_k)\} \in 2^{\{x_i | 1 \leq i \leq n\} \times \{\leq, \geq\} \times \mathbb{Z}} \quad (3.40)$$

Let us further introduce the following shorthand notation:

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

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

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

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:

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

Proposition 3.1. *If λ_{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 $Q_{frac} := \{q \in Q \mid \lambda_q^* \notin \mathbb{Z}\} \neq \emptyset$ be the set of columns with currently fractional master variables. Then take $q^* := \arg \min_{q \in Q_{frac}} \mathbf{x}_q$ as any minimal undominated column in Q_{frac} . From q^* , we can now construct a component bound sequence S , which is only satisfied by q^* out of all $q \in Q_{frac}$, as follows:

$$S := \{(x_i, \leq, \lfloor x_{q^*} \rfloor) \mid x_i \in \{x_j \mid q_j \in Q_{frac}\}\} \quad (3.43)$$

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

Vanderbeck's scheme divides the solution space along the component bounds into multiple sub-polyhedra. In this way, each child branch can only generate points within its own sub-polyhedron, and the master solution will be integral within one of these sub-polyhedra. In fact, this scheme closely resembles the main idea of dichotomous branching in branch-and-bound, where the solution space is divided into two halves. For a given component bound sequence $S = \{B_1, \dots, B_m\}$, where each variable x_i has at least one upper and one lower component bound, there are up to $2^n - 1$ possible sub-polyhedra. To avoid an exponential increases in nodes,

we group some sub-polyhedra together, creating a total of $n + 1$ nodes. Each of the $1 \leq j \leq m + 1$ nodes is now modified as follows: first define the component bound sequence S_j for the j -th node as:

$$S_j := \begin{cases} \{B_1, \dots, B_{j-1}, \bar{B}_j\} & \text{if } j \leq m \\ \{B_1, \dots, B_m\} & \text{if } j = m + 1 \end{cases} \quad (3.44)$$

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

$$K_j := \sum_{q \in Q(S_j)} \lambda_q^* \quad (3.45)$$

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

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

Finally, modify the pricing problem as follows:

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

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

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

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

evenly. We initialize the component bound sequence 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

```

1 if parentnode = null then
2   |  $S \leftarrow \emptyset$ ;
3 else
4   |  $S \leftarrow \text{parentnode.S}$ ;
5 end
6 loop
7   |  $\alpha \leftarrow \sum_{q \in Q(S)} x_q \lambda_q^*$ ;
8   | if  $\forall 1 \leq i \leq n. \alpha_i \in \mathbb{Z}$  then
9     | return S;
10  | end
11  |  $i \leftarrow \text{any}(i \mid \alpha_i \notin \mathbb{Z})$ ;
12  | if Choose( $i$ ) then
13    |  $S \leftarrow S \cup \{(x_i, \leq, \lfloor x_i^* \rfloor)\}$ ;
14  | else
15    |  $S \leftarrow S \cup \{(x_i, \geq, \lfloor x_i^* \rfloor + 1)\}$ ;
16  | end

```

This presentation of Vanderbeck’s generic branching scheme covers the main ideas and concepts. For a more in-depth derivation of this rule, detailed descriptions of the routines, and further optimizations such as node pruning, refer to [1–3, 9].

3.6 Branch-Price-and-Cut

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

We can extend this concept to branch-and-price. Whenever we have a solution for the LP relaxation of the MP , we can add additional valid inequalities to the RMP , aiming to strengthen the relaxation. This extension transforms the branch-and-price algorithm into a branch-price-and-cut algorithm. In general, separators generating cuts for the MP operate either on the original formulation or within the

master problem of the Dantzig-Wolfe reformulation. We will briefly discuss these two types of separators. For more detailed information about cutting planes for column generation and their effectiveness, refer to [10, 11].

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 λ^* , which can be projected back into a solution x^* of the original formulation. We can then call any separation algorithms that operate on the original formulation to generate cuts of the general form:

$$F^\top x \geq f \quad (3.49)$$

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

$$\sum_{p \in P} F x_p \lambda_p + \sum_{r \in R} F x_r \lambda_r \geq f \quad [\alpha] \quad (3.50)$$

and imposing the constraints in the pricing problem:

$$\begin{aligned} z_{SP}^* = \min \quad & (c^\top - \pi_b^\top A - \alpha^\top F) x - \pi_0 \\ \text{s. t.} \quad & D x \geq d \\ & x \in \mathbb{Z}_+^n \end{aligned} \quad (3.51)$$

This approach allows existing separators originally intended for use in a branch-and-cut scenario to be reused to generate cutting planes for the Dantzig-Wolfe reformulation. However, some caveats apply. For example, some separators rely on a basis solution. Since the Dantzig-Wolfe reformulation might be stronger than the original formulation [12], an interior point of the polyhedron could be the optimal solution for the relaxed *RMP*. In this case, the basis solution is not available, and such a separator cannot be applied directly.

3.6.2 Separators using the Master Problem

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

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

requiring the following modifications to the pricing problem:

$$\begin{aligned} z_{SP}^* = \min \quad & (\mathbf{c}^\top - \boldsymbol{\pi}_b^\top \mathbf{A}) \mathbf{x} - \gamma g_{\mathbf{x}} - \pi_0 \\ \text{s. t.} \quad & \mathbf{D}\mathbf{x} \geq \mathbf{d} \\ & g_{\mathbf{x}} = g(\mathbf{x}) \\ & \mathbf{x} \in \mathbb{Z}_+^n \\ & g_{\mathbf{x}} \in \mathbb{R} \end{aligned} \quad (3.53)$$

As with branching on master variables (Section 3.5.2), the challenge is expressing $g_{\mathbf{x}} = g(\mathbf{x})$ using a finite set of linear constraints. In her master thesis, Chantal Reinartz Groba implements such a master separator within the **GCG** framework [13].

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 [14, 15].

// TODO

Chapter 4

SCIP Optimization Suite

4.1 SCIP

4.2 GCG

Chapter 5

Component Bound Branching

In this chapter, we introduce the **component bound branching rule** (COMPBND) for branching on the master variables of the discretized reformulation of any type of bounded *IP*. This new branching rule builds upon the fundamental ideas of Vanderbeck's generic branching scheme (Section 3.5.2.1), aiming to provide a simpler alternative to branching on component bounds. We will begin by demonstrating how to enforce component bounds to create a binary branch-and-bound search tree. Next, we will delve into the algorithm responsible for determining suitable branching decisions. Finally, we will compare and contrast Vanderbeck's generic branching scheme with our new approach, highlighting their similarities and differences.

5.1 Overview of the branching scheme

As discussed in Section 3.5.2, given a fractional master solution λ_{RMP}^* , we can always find a subset $\emptyset \subset Q' \subset Q := \tilde{P}$ such that:

$$\sum_{q \in Q'} \lambda_q^* =: K \notin \mathbb{Z} \quad (5.1)$$

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

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

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

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

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

To find such a Q' that is expressible in SP , Vanderbeck uses bounds on the original variables. Similarly, in our branching scheme, we also find such component bounds. Let us reiterate the notation introduced for Vanderbeck's branching scheme in Section 3.5.2.1:

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

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

We define a component bound sequence as follows:

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

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

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

We continue using the following shorthand notation:

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

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

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

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

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

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

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 $\forall (x_i, \eta, v) \in S. \eta(x_{q'}, v)$ 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 :

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

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

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

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

$$\begin{aligned} y \geq 1 + \sum_{B_s \in \bar{S}} \bar{y}_s + \sum_{B_{s'} \in \underline{S}} \underline{y}_{s'} - |S| & \quad \Bigg| \quad \begin{aligned} y &\leq \bar{y}_s && \forall B_s \in \bar{S} \\ y &\leq \underline{y}_{s'} && \forall B_{s'} \in \underline{S} \end{aligned} \\ \bar{y}_s \geq \frac{v_s + 1 - x_{i,s}}{v_s + 1 - \text{lb}_i} & \quad \forall B_s \in \bar{S} & \quad \Bigg| \quad \bar{y}_s \leq \frac{\text{ub}_i - x_{i,s}}{\text{ub}_i - v_s} & \quad \forall B_s \in \bar{S} \\ \underline{y}_{s'} \geq \frac{x_{i,s} - v_s}{\text{ub}_i - v_s} & \quad \forall B_{s'} \in \underline{S} & \quad \Bigg| \quad \underline{y}_{s'} \leq \frac{x_{i,s} - \text{lb}_i}{v_s - \text{lb}_i} & \quad \forall B_{s'} \in \underline{S} \end{aligned} \quad (5.12)$$

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

5.2 Separation Procedure

Definition 5.1. *The **fractionality of λ_{RMP}^* with respect to S** is given by:*

$$F_S = \sum_{q \in Q(S)} (\lambda_q^* - \lfloor \lambda_q^* \rfloor) \geq 0 \quad (5.13)$$

When $S = \emptyset$, we have $Q(S) = Q$, and thus $F_S > 0$, since at least one λ_q^* is fractional. In particular, $F_S \in \mathbb{Z}_+ \setminus \{0\}$ 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).

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

- $F_S = 0$: $Q(S)$ contains no column with fractional λ_q^* . Thus, branching on S would not cut off the current fractional solution λ_{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:

$$\sum_{q \in Q(S)} \lfloor \lambda_q^* \rfloor < \sum_{q \in Q(S)} \lambda_q^* < \sum_{q \in Q(S)} \lfloor \lambda_q^* \rfloor + 1 \quad (5.14)$$

We observe the sum $\sum_{q \in Q(S)} \lambda_q^* =: K$ to be fractional, which enables us to branch on S (see Equation (5.1)).

- $F_S \in \mathbb{Z}_+ \setminus \{0\}$. In this case, $\sum_{q \in Q(S)} \lambda_q^* \in \mathbb{Z}_+$, 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 $i \in \{1, \dots, n\}$, such that $\lambda_{q_1}^*$ and $\lambda_{q_2}^*$ are fractional. If we denote the rounded median of these two column entries as $v := \lfloor \frac{x_{i,q_1} + x_{i,q_2}}{2} \rfloor$, 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:

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

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 $F_{S^k} \in \mathbb{Z}_+ \setminus \{0\}$, 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 $F_{S^k} \in \mathbb{Z}_+ \setminus \{0\}$. 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. \square

Proposition 5.2. *Given that λ_{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., $|Q_f(S^{k+1})| < |Q_f(S^k)|$. Since $|Q_f(S^0)| < \infty$, the separation procedure must terminate after a finite number of iterations. \square

5.2.1 Choice of Component Bounds

The separation procedure described above is not complete yet, as we have not defined which component x_i we impose which bounds on. This choice can significantly impact the performance of the subsequent solving of the child nodes. In the worst case, the separation procedure will yield a component bound sequence S for which $Q(S)$ only contains one column. Maintaining balance within the branch-and-bound tree is generally beneficial, but 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 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:

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

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, thereby maintaining 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 number 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 [16].

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

5.2.3 Branching with Multiple Subproblems

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

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

5.3 Comparison to Vanderbeck’s Generic Branching

Both the generic branching scheme by Vanderbeck (Section 3.5.2.1) and the proposed component bound scheme involve branching in the master problem by imposing bounds on the original variables within the SP . The primary difference lies in how these component bounds are imposed. Vanderbeck enforces these bounds as hard constraints, effectively subdividing the solution space of the original variables. Consequently, when the optimal solution \mathbf{x}^* to the IP is found in a branch-and-bound node of the RMP , it satisfies all the component bounds imposed by the branching decisions from the root to that node. In contrast, our approach adds these bounds as soft constraints, allowing both columns that satisfy and those that violate the component bounds to 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 IP s 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

In addition to implementing the component bound branching rule (Chapter 5), a significant goal of this thesis is to enable future **GCG** developers to easily create new branching rules and separators within the framework (Section 4.2). Currently, **GCG** has limitations: branching rules must either produce decisions formulated in the original problem, which can then be Dantzig-Wolfe reformulated and added to the master and pricing problems (as seen when branching on original variables in Section 3.5.1), or they must produce constraints for the master problem without requiring modifications to the pricing problem, as seen with Ryan-Foster branching. Other branching rules, such as Vanderbeck’s generic branching scheme (Section 3.5.2.1), cannot be implemented without significant changes to the **GCG** framework. These changes would involve applying and removing component bounds in the pricing problem when a B&B-node is entered or left. Our proposed component bound branching rule (Chapter 5) and any separators using the master problem (Section 3.6.2) would also require such changes. The reason is that **GCG** does not currently support imposing constraints in the master problem that necessitate modifications to at least one SP , where the master constraints and induced pricing problem modifications are not a product of a Dantzig-Wolfe reformulation of the original problem and do not necessarily have a counterpart in the original problem.

In this chapter, we will specify the notation of such constraints, referred to as **generic mastercuts**. We will present our integration of these constraints into the **GCG** framework as part of a new interface and demonstrate how to apply dual value stabilization to these constraints.

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

separators.

Definition 6.1. A *generic mastercut* is a constraint in the master problem that does not have a counterpart in the original problem, and therefore requires modification to one or multiple SP to ensure its validity in the master. Specifically, it takes the following form:

$$\sum_{p \in P} f_p \lambda_p + \sum_{r \in R} f_r \lambda_r \leq f' \quad [\gamma]$$

The subproblems are now responsible for correctly determining the coefficients f_p and f_r of all newly generated columns p and r .

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

Every pricing modification includes at least one mandatory variable y with an objective coefficient of $-\gamma$ in the SP^k . The solution value of y is used as the column entry for the master constraint of the generic mastercut, i.e., f_p or f_r . This variable is known as the **coefficient variable** of the pricing modification. All additional variables will take on auxiliary roles 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

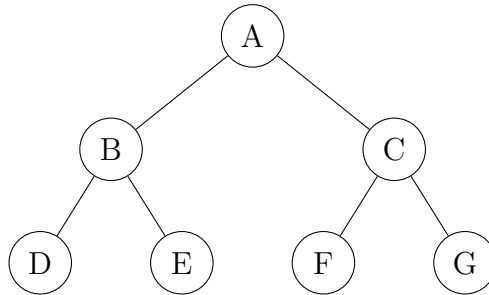


Figure 6.1: An exemplary B&B-tree created by the component bound branching rule, where the lexicographic order of the nodes resembles the order in which they were created.

Generic mastercuts used as branching decisions need to be aware of all columns currently existing in the *RMP*. For instance, in component bound branching (Chapter 5), all columns satisfying the branching decision must have a coefficient of 1 for the mastercut. This can be easily managed when creating the branching decision and when a new column is generated in the subtree of the node where the decision was made. However, consider the following scenario in a branch-and-bound tree generated with the component bound branching rule, as depicted in Figure 6.1: we are currently processing node F in the B&B-tree and generate a new column q' . After deactivating node F and activating node D, it is possible that $x_{q'}$ satisfies the component bounds imposed in D. Thus, q' should have a coefficient of 1 in the mastercut of D. However, since the column was generated in F, it is not known to D, potentially invalidating the mastercut of D.

To prevent this, branching decisions must be aware of these columns to update their coefficients accordingly. Specifically, we want to synchronize newly generated master variables across the entire B&B-tree lazily, i.e., only when a node is activated. Moreover, in *GCG*, master variables deemed unnecessary can be removed from the master problem, and deleted variables should not require synchronization.

In this section, we will first analyze the current approach taken by the implementation of Vanderbeck's generic branching in *GCG*. Then, we will present a more efficient approach, referred 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 deactivated to reflect any newly generated columns. Upon node activation, the current number of variables in the master is compared against how many variables were present the last time the node was active. If new columns have been added to the *RMP* in the meantime, this counter might increase. If so, the coefficient for the new columns will be determined and set in the *RMP*.

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

Additionally, this approach is not developer-friendly. Developers of branching rules should not have to manage when and where columns are generated and

deleted. This should be handled by the framework, requiring developers to only implement an interface method responsible for setting the coefficient of any column, regardless of where it was generated.

6.1.2 History Tracking Approach

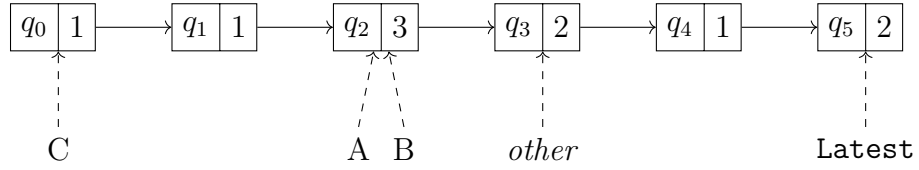


Figure 6.2: Reference-counted linked list of the history of columns added to the *RMP*, with external references drawn dashed from below, e.g., those from the B&B-tree nodes A, B, and C. Each element holds a reference to the master variable belonging to column q_i , as well as the number of references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

We propose a highly efficient approach to lazily notify all nodes in the B&B-tree of new columns generated while also considering deleted columns. We introduce a reference-counted linked list of variables added to the *RMP*, where the order of the variables in the list is determined by their generation order. Each node in the B&B-tree holds its own external reference to this construct. The specific element in the list that a B&B-tree node points to signifies the last column in the *RMP* when the node was last active. All subsequent variables, i.e., the elements next in the list, are new columns generated elsewhere in the list. Additionally, we hold one external reference to the tail of the list, representing the last generated column. This construct is illustrated in Figure 6.2. Since the linked list tracks which variables were created when we will refer to this list as the **varhistory**.

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

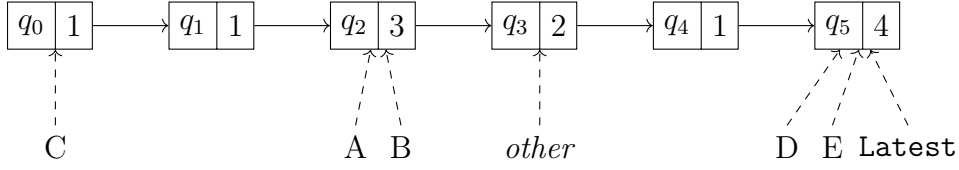


Figure 6.3: **varhistory** after creating child nodes D and E of node B.

Continuing this scenario, let **GCG** deem the column q_2 unnecessary and remove it from the *RMP*. Since there may be external references to this variable, which in this case there are, we do not remove the element in the list holding q_2 . Instead, we mark it as deleted. Next, we would like to solve node C. For this, we must deactivate node B, and activate node 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.

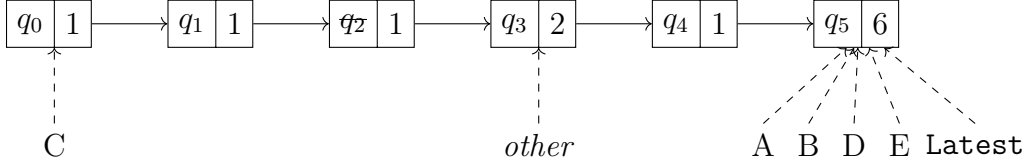


Figure 6.4: **varhistory** after deletion of column q_2 and deactivation of node B.

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 forward the pointer of C one by one until reaching 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 to 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 is updated. As soon as this reference count reaches zero, the element can be safely freed. This ensures that only necessary variables, those that still need to be synchronized across the entire B&B-tree, are kept in memory. This is illustrated in Figure 6.5.

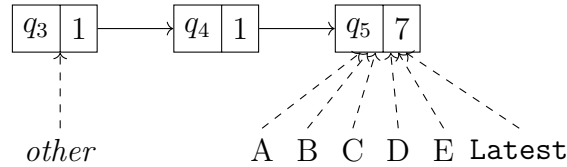


Figure 6.5: **varhistory** after activation of node C.

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

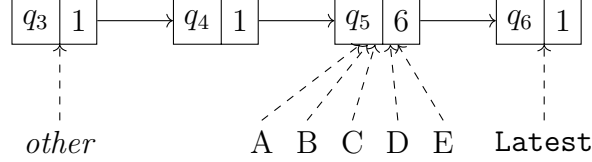


Figure 6.6: **varhistory** after generating column q_6 in node C.

This approach is highly efficient as it updates the reference count of elements in the list and passes only new variables to the branching rule. The deletion of variables is not an issue, as we simply do not pass them to the branching rules. Once all external references have seen some variable q_i , i.e., its reference count reaches zero, we can automatically free the memory of the element in the list holding q_i . This minimizes 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 [13].

6.1.3 History Tracking using Unrolled Linked Lists Approach

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

We can optimize by unrolling the linked list into blocks storing a fixed number of columns, with each block having its own reference count. These blocks are linked together, forming a list of blocks. The references to the **varhistory** point to the blocks and hold an offset within the block. This way, each reference still refers to some q_i , enabling pointer forwarding one column at a time. A new variable is added to the tail block if space allows; otherwise, a new block is allocated. This concept is illustrated in Figure 6.7.

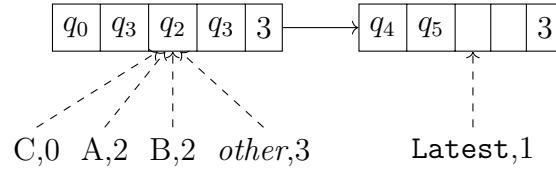


Figure 6.7: `varhistory` of Figure 6.2 unrolled into blocks of size 4.

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

6.2 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] Nathan Chappell. “Minkowski-Weyl Theorem”. In: (2019).
- [5] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.
- [6] George B Dantzig and Mukund N Thapa. *The simplex method*. Springer, 1997.
- [7] Nikolaos Ploskas and Nikolaos Samaras. “Pivoting rules for the revised simplex algorithm”. In: *Yugoslav Journal of Operations Research* 24.3 (2014), pp. 321–332.
- [8] Jii Matouek and Bernd Gärtner. *Understanding and using linear programming*. Vol. 1. Springer, 2007.
- [9] Marcel Schmickerath. “Experiments on Vanderbeck’s generic Branch-and-Price scheme”. In: (2012).
- [10] Gerald Gamrath. “Generic branch-cut-and-price”. MA thesis. 2010.
- [11] Jonas T Witt. “Separation of Generic Cutting Planes in Branch-and-Price”. PhD thesis. Master’s thesis. RWTH Aachen University, 2013.
- [12] Michael Bastubbe, Marco E Lübbecke, and Jonas T Witt. “A Computational Investigation on the Strength of Dantzig-Wolfe Reformulations”. In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.
- [13] Chantal Reinartz Groba. “TODO”. In: (2024).

- [14] Artur Pessoa et al. “In-out separation and column generation stabilization by dual price smoothing”. In: *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*. Springer. 2013, pp. 354–365.
- [15] Artur Pessoa et al. “Automation and combination of linear-programming based stabilization techniques in column generation”. In: *INFORMS Journal on Computing* 30.2 (2018), pp. 339–360.
- [16] Tobias Achterberg, Thorsten Koch, and Alexander Martin. “Branching rules revisited”. In: *Operations Research Letters* 33.1 (2005), pp. 42–54.