MASTER'S THESIS

An Implementation of the Branch-and-Price Algorithm Applied to Opportunistic Maintenance Planning

DAVID FRIBERG

Department of Mathematical Sciences
Division of Mathematics

CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Gothenburg, Sweden 2015

Thesis for the Degree of Master of Science

An Implementation of the Branch-and-Price Algorithm Applied to Opportunistic Maintenance Planning

David Friberg

Department of Mathematical Sciences
Division of Mathematics
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden
Gothenburg, December 2015

Abstract

In the following thesis we march our way through the basic Linear Programming (LP) and LP duality theory, in order to cover the subject of column generation (CG) in the context of Dantzig-Wolfe decomposition. We proceed to thoroughly cover the LP-based Branch-and-Bound (BAB) algorithm for solving mixed integer linear programs (MILP), and by making use of the CG concept, we extend BAB into the more sophisticated Branch-and-Price (BAP) algorithm for solving MILP:s. With the theory attained, we implement a basic problem-specific BAP algorithm for solving instances of the opportunistic replacement problem (ORP), applied to a case study of the maintenance planning for the RM12 aircraft engine. Finally, we refine the basic algorithm by identifying and studying key factors in the BAP implementation; CG refinement, LP solver type, node picking, and branching rules for the BAB/BAP framework.

Keywords: mixed integer linear programming, mixed binary linear programming, column generation, Dantzig-Wolfe decomposition, branch-and-bound, branch-and-price, opportunistic replacement problem, opportunistic maintenance planning, weighted column generation, pseudocost branching

Acknowledgements

I would like to thank my supervisor Ann-Brith Strömberg not only for her valuable insights in the technical and theoretical aspects of this work, but also for for her great patience and encouragement in my finalizing of the thesis. I am particularly grateful for her thorough review of the thesis; in addition to resulting in a better final product, the review process also teached me various important aspects of scientific writing.

I would also like to thank Adam Wojciechowski for helping out with the initial column generation implementation in AMPL, bringing insights that was of great help for grasping the practical application of the algorithm, an experience that proved valuable when migrating the project to C++.

Finally I would like to thank my wife Elin for her never-ending encouragament and persuasion in the finishing of the thesis.

Contents

List of abbreviations **1**

1 Introduction **2**

2 An Overview of Linear and Integer Linear Programming **4**
2.1 A Linear Programming Problem 5
2.2 Extension to Integer Linear Programming 6
2.3 The Lagrangian Dual Problem and Lagrangian Duality 8
2.4 A Method for Solving Linear Programs: The Simplex Method 10

3 Dantzig-Wolfe Decomposition and Column Generation for Linear Programming **13**
3.1 General Column Generation 14
3.2 The Dantzig-Wolfe Decomposition Principle 16
3.3 Extension to Integer Programming 20

4 Methods for Attaining Integer Valued Solutions from Linear Programming Based Column Generation **24**
4.1 The Branch-and-Bound Algorithm 25
4.1.1 An illustrating example 30
4.2 BAB in the Context of Dantzig-Wolfe Decomposition and Column Generation 35
4.3 Dynamic Update of the Column Pool: Branch-and-Price 38

5 A Case Study: the Opportunistic Replacement Problem **42**
5.1 Background 43
5.2 Mathematical Formulation 46
5.3 Reformulation into a Column Generation Problem 49
5.3.1 Preparations 50
5.3.2 The RMP and its associated subproblems 53

6 Implementing a Problem-Specific Branch-and-Price Algorithm 55
6.1 Choosing an External LP Solver and Deciding on a Compliant and Appropriate Programming API . . . 56
6.2 Initial Set-Up: Algorithmic Specifications Necessary for Implementing a Basic BAP Framework . . . 57
6.3 Implementation of a Problem-Specific BAP Algorithm Using a High-Level Programming Language . . . 64

7 Tests and Results: Improving the Basic Branch-and-Price Implementation 66
7.1 Initial Set-Up: Building a Basic BAP Framework and Identifying Algorithmic Key Factors Prune to Possible Enchancement . . . 67
7.2 First Modification: Dealing With the Tailing-Off Effect Using Weighted Dantzig-Wolfe Decomposition . . . 69
7.3 Second Modification: How to Choose Nodes from the Node Tree. 74
7.4 Third Modification: Choosing the Primal or Dual Simplex Method as the LP Solver . . . 81
7.5 Fourth Modification: Introducing a Sophisticated Va-riable Branching Rule—Pseudocost Branching . . . 84
7.6 Results: Comparing Final Setups with CPLEX MIP Solver . . . 90

8 Conclusions and future work 97

List of abbreviations

API	Application Programming Interface
BAB	Branch-and-Bound
BAP	Branch-and-Price
BFS	Basic Feasible Solution
BLP	Binary Linear Programming
CG	Column Generation
ILP	Integer Linear Programming
LP	Linear Programming
MBLP	Mixed Binary Linear Programming
MILP	Mixed Integer Linear Programming
MP	Master Problem
OR	Operations Research
ORP	Opportunistic Replacement Problem
PMSPIC	Preventive Maintenance Scheduling Problem with Interval Costs
RMP	Restricted Master Problem
SORP	Stochastic Opportunistic Replacement Problem

Chapter 1

Introduction

The following thesis investigates the possibility of making use of the Linear Programming (LP)-based branch-and-price (BAP) algorithm for solving instances of the Opportunistic Replacement Problem (ORP), as the problem is presented in [1]. The thesis roughly contain three parts. The first part covers the theory necessary to grasp and make use of the BAP algorithm, Chapters 2 through 4. The second part, Chapters 5 and 6, presents the case study and adapts it to BAP-friendly form, and discusses the necessary steps to be taken prior to implementing a basic BAP algorithm applied to the instances of the case study. Finally, Chapters 7 and 8 cover testing and ways to improve the algorithm—based on the results of solving instances of the case study—of which some are implemented and others are discussed as future work.

Chapter 2 cover general LP theory—including LP duality—and its extension to Integer Linear Programming (ILP), and briefly presents the well-known simplex method for solving LP problems. Chapter 3 proceeds the theory of column generation and Dantzig-Wolfe decomposition, a method and a representation intertwined in use for solving size-wise difficult large LP:s, in the sense that their initial form cannot be directly attacked with the simplex method. While the Dantzig-Wolfe decomposition is a method for re-constructing LP problems, Chapter 3 also presents the equivalent representation form for ILP problems; discretization. Finally, Chapter 4 thoroughly presents the branch-and-bound (BAB) algorithm for solving ILP problems, making use of LP techniques. The algorithm is thereafter extended to the BAP algorithm, combining BAB with CG and Dantzig-Wolfe decomposition (discretization).

Chapter 5 introduces the case study, the opportunistic maintenance planning for the RM12 aircraft engine, which can be expressed as an ORP, and covers the background as well as its mathematical formulation as a Mixed Binary Linear Programming (MBLP) problem, as presented in [2]. The MBLP is thereafter re-formulated using discretization, into the form used in the problem-specific BAP implementation,

constructed with the purpose to solve instances of the problem. Chapter 6 proceeds to specify details and concepts needed in order to proceed with implementing a basic, although fully functioning, BAP algorithm in the high-level programming language C++; from scratch, making use, however, of the LP solver from the commercial optimization software CPLEX [3], specifically via CPLEX:s C++ Application Programming Interface (API) Concert C++.

Chapter 7 defines a number of tests of possible improvements for the basic BAP implementation, with the purpose of improving the algorithm by focusing on specific details, as type of LP solver, node picking rule for the processing of BAP subproblems (nodes), and branching rule for the purpose of deciding which variables to choose to branch upon when expanding the BAP subproblem list (tree). The results of the final BAP implementation is thereafter compared with the corresponding results of the CPLEX MIP Solver [3]. Finally, Chapter 8 discusses lessons learned from the testing in Chapter 7 as well as additional possible improvements that should be interesting subjects of future work, in the case where the BAP implementation were to be further developed, including also possible interesting extensions of the case study.

Chapter 2

An Overview of Linear and Integer Linear Programming

Linear programming (LP) encompasses the field of optimization where one wants to find the best, i.e., optimal, solution to some mathematical optimization model whose variables are continuous and whose objective and constraint functions are linear. A strength of linear programming is its ease of modeling and the existence of efficient solution algorithms, which is related to the fact that all linear programs are convex, with a concave or convex objective function, depending whether the problem is a minimization or maximization problem, respectively. While the limitation to linear modeling might seem as a first-glance obvious weakness, LP is widely used and has gained a strong position in practical optimization, ever since G.B. Dantzig developed the simplex method for solving linear programs in 1947 [4]. For a reader with a deeper interest in the the various historic events leading to Dantzig's discovery, see [5][1].

We assume that the reader is somewhat familiar with optimization theory. This chapter will only briefly introduce linear programming and some of its basic concepts. The first three first sections cover general theory and Section 2.4 proceed with summarizing the simplex method for solving linear programs. For the reader interested in a more thorough passage of the subject, there exists a vast amount of literature on the topic, of which the textbooks [4, 6] are two good examples.

[1]Apart from covering the road to Dantzig's most famous contribution to mathematics, the article generally encompass an interesting, detailed, but concise biographical account of the mathematician's professional life and of his achievements beyond that of the simplex method.

2.1 A Linear Programming Problem

Consider the model to

$$
\begin{array}{llcccccccccr}
\underset{x}{\text{minimize}} & z_{\text{LP}} = & c_1x_1 & + & c_2x_2 & + & \dots & + & c_nx_n, & & & \text{(2.1a)}\\
\text{subject to} & & a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n \geq & b_1, & & \text{(2.1b)}\\
 & & a_{21}x_1 & + & a_{22}x_2 & & \dots & & a_{2n}x_n \geq & b_2, & & \text{(2.1c)}\\
 & & \vdots & & \vdots & & & & \vdots & \vdots, & & \text{(2.1d)}\\
 & & a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n \geq & b_m, & & \text{(2.1e)}\\
 & & x_1 & , & x_2 & , & \dots & , & x_n \geq & 0, & & \text{(2.1f)}
\end{array}
$$

which is a linear minimization problem on canonical form. Canonical form will be explained shortly, but first we shall look at the different components of the problem above. The equation (2.1a) describes the *objective function*, the value of which, z_{LP}, is to be minimized. It contains a set of *decision variables* $x_1, \dots, x_n$—the values of which are to be determined—as well as corresponding constants $c_1, \dots, c_n \in \mathbb{R}$. The inequalities (2.1b–2.1f) describe the *constraints*, which any valid solution to the problem must abide. Apart from the decision variables, the constraints contain a number of constants b_i, $a_{ij} \in \mathbb{R}$, $i = 1, \dots, m$, $j = 1, \dots, n$. A valid solution, that is, a valid combination of variable values $(x_1, \dots, x_n)$, is commonly called a *feasible solution*, or a feasible point, and the set in $\mathbb{R}^n$ of all feasible points abiding the constraints (2.1b–2.1f) defines the feasible region for the problem. The feasible solution which correspond to the lowest (or in the case of a maximization problem, the highest) objective function value z_{LP} in (2.1a), is called the *optimal solution* $(x_1^*, \dots, x_n^*)$ of (2.1), which is said to solve (2.1), with corresponding *optimal objective function value* z_{LP}^*.

To simplify presentation, we introduce the vectors $\boldsymbol{x}$, $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ and the matrix $A \in \mathbb{R}^{m \times n}$, with

$$
\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.
$$

The problem (2.1) can now be rewritten, using matrix notation, as that to

$$
\begin{array}{llrr}
\underset{x}{\text{minimize}} & z_{\text{LP}} = & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, & \text{(2.2a)}\\
\text{subject to} & & A\boldsymbol{x} \geq \boldsymbol{b}, & \text{(2.2b)}\\
 & & \boldsymbol{x} \geq \boldsymbol{0}^n. & \text{(2.2c)}
\end{array}
$$

Now, a linear *minimization* problem is said to be on canonical form if all constraints—with the exception of the compulsory non-negativity restriction on the decision variables—are inequality constraints of the type *greater than or equal to*. Conversely, any linear *maximization* problem, an LP problem in which the objective function is to be maximized, is said to be on canonical form if all constraints are inequality constraints of the type *less than or equal to*. Consequently, the problems $\min_{\boldsymbol{x}} \left\{ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \mid A\boldsymbol{x} \geq \boldsymbol{b},\, \boldsymbol{x} \geq \boldsymbol{0} \right\}$ and $\max_{\boldsymbol{x}} \left\{ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b},\, \boldsymbol{x} \geq \boldsymbol{0} \right\}$ are general examples of a minimization and maximization problem, respectively, on canonical form. We have chosen to also attach the non-negativity of the decision variables ($\boldsymbol{x} \geq \boldsymbol{0}$) in our definition of canonical form, something that some authors include while other leave it out. Furthermore, a linear programming problem is said to be on *standard form* if the objective function is to be minimized, all variables are non-negative, the constraints are equalities and the right-hand-side for each constraint is non-negative, i.e., $\min_{\boldsymbol{x}} \left\{ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \mid A\boldsymbol{x} = \boldsymbol{b},\, \boldsymbol{x} \geq \boldsymbol{0},\, \boldsymbol{b} \geq \boldsymbol{0} \right\}$ is the general form of an LP problem on standard form, a form that is useful in practice. An example is the simplex method for solving linear programs—which is briefly explained in the last section of this chapter—which operates on LP's that are on this form. Any linear program can be transformed into the standard form using some elementary operations, such as introducing slack variables and so on. For details, see any of the textbooks [4, 6].

We return to the model (2.2). The feasible region defined by (2.2b–2.2c) for this general LP problem is given by $X^{\mathrm{LP}} = \{\boldsymbol{x} \in \mathbb{R}^n_+ \mid A\boldsymbol{x} \geq \boldsymbol{b}\}$, which defines a subset of $\mathbb{R}^n$, being a polyhedron [6, p. 47]. This property of linear programs is utilized by the simplex method, and is in fact a key participant in the naming of the method[2] [7].

LP problems are known to be solvable in polynomial time [6, p. 238], and most linear programs are in practice solvable by the well-used simplex method abiding this complexity. There exists problem cases, however, confirming that the worst-case time complexity of the simplex method is exponential [8, p. 304].

We proceed by extending LP into integer linear programming (ILP), where the decision variables of the linear program are constrained to integer values.

2.2 Extension to Integer Linear Programming

A common use of linear programs is the case when the decision variables are restricted to integer values. E.g. when modeling some real-life situation in which several yes/no decisions are to be made, constraining the decision variables to binary values is a natural modeling principle. For a situation where the goal of the model-

[2] Bounded polyhedra are closely related to a generalized form in geometry, denoted *simplex*.

ing is to count some commodities and represent these numbers by decision variables, fractional values might be unwanted w.r.t. the situation modeled, and integer requirements on the variables are naturally enforced.

We consider the LP minimization problem (2.2), and add integer requirements for all decision variables $x_1, \ldots, x_n$, attaining the integer (linear) program (ILP)

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad z_{\text{ILP}} = \quad \boldsymbol{c}^{\text{T}}\boldsymbol{x}, \tag{2.3a}$$

$$\text{subject to} \quad A\boldsymbol{x} \geq \boldsymbol{b}, \tag{2.3b}$$

$$\boldsymbol{x} \in \mathbb{Z}_{+}^{n}. \tag{2.3c}$$

The feasible region of this program, $X^{\text{ILP}} = \{\boldsymbol{x} \in \mathbb{Z}_{+}^{n} \mid A\boldsymbol{x} \geq \boldsymbol{b}\}$, is a subset of the feasible region of the corresponding linear program (2.2), since $X^{\text{ILP}} = X^{\text{LP}} \cap \mathbb{Z}^{n}$. Therefore, as we consider a minimization problem, the optimal objective function value z_{LP}^{*} of the linear program provides a lower bound on the optimal objective function value z_{ILP}^{*} of the integer linear program (2.3). That is, $z_{\text{LP}}^{*} \leq z_{\text{ILP}}^{*}$. The LP (2.2) is referred to as the continuous relaxation of the ILP (2.3).

A special case of integer linear programming is when decision variables are restricted to 0/1 values only, usually used when modeling some yes/no decisions, as one of the examples mentioned in the beginning of this section. These kinds of problems are referred to as binary linear programming (BLP) problems. Apart from pure integer linear programming, another common modeling class is mixed integer linear programming (MILP). In these models, a subset of the decision variables are allowed to take continuous values, while the rest are enforced to integrality.

In contrast to the complexity of LP problems—solvable in polynomial time—ILP problems are generally *NP-hard* [9, pp. 81–84], which means that they are at least as hard as the hardest problems in the fundamental complexity class NP (abbreviation for *nondeterministic polynomial time*) in computational complexity theory. The hardest problems in NP are denoted *NP complete* problems, problems for which there exist no *known*[3] polynomial-time solution algorithm. Consequently, there exist no known polynomial-time algorithm for solving the NP-hard ILP problems.

Methods for obtaining optimal or near-optimal solutions for integer programs will be discussed in Chapter 4, all of which make use of the continuous relaxation of the ILP problem at hand.

[3]Note the emphasis on *'known'* here. There exists no known polynomial-time algorithm for solving NP complete problems, but neither does there exist any proof that there cannot be such a polynomial-time algorithm, even if it is generally suspected that that is the case.

2.3 The Lagrangian Dual Problem and Lagrangian Duality

An important concept in linear programming is the linear programming dual problem. We shall derive the LP dual using the tools of Lagrangian duality, and the associated Lagrangian dual problem. We briefly describe Lagrangian duality in the context of linear programming.

Consider the LP minimization problem (2.2). We refer to this problem as the primal problem. For an arbitrary vector $\boldsymbol{\mu} \in \mathbb{R}^m$, we define the Lagrangian function as

$$L(\boldsymbol{x}, \boldsymbol{\mu}) := \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{\mu}^{\mathrm{T}}(\boldsymbol{b} - A\boldsymbol{x}). \tag{2.4}$$

Further, we define the Lagrangian dual function as

$$\theta_{\mathrm{LP}}(\boldsymbol{\mu}) := \min_{\boldsymbol{x} \geq \boldsymbol{0}^n} L(\boldsymbol{x}, \boldsymbol{\mu}) = \min_{\boldsymbol{x} \geq \boldsymbol{0}^n} \left\{ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{\mu}^{\mathrm{T}}(\boldsymbol{b} - A\boldsymbol{x}) \right\} = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{\mu} + \min_{\boldsymbol{x} \geq \boldsymbol{0}^n} \left\{ (\boldsymbol{c}^{\mathrm{T}} - \boldsymbol{\mu}^{\mathrm{T}}A)\boldsymbol{x} \right\}. \tag{2.5}$$

For any $\boldsymbol{\mu} \geq \boldsymbol{0}^m$, the Lagrangian dual function is called the Lagrangian relaxation of (2.2), and it's optimal solution in $\boldsymbol{x}$ yields a lower bound on the optimal solution of the primal problem. We realize this from the following discussion, throughout which it's assumed that $\boldsymbol{\mu} \geq \boldsymbol{0}^m$. Assume that $\boldsymbol{x}^*$ solves (2.2) with optimal objective function value $z^*_{\mathrm{LP}} = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^*$, and assume that $\bar{\boldsymbol{x}}_{\boldsymbol{\mu}}$ solves the minimization in (2.5). From (2.2b) we have that $\boldsymbol{b} - A\boldsymbol{x}^* \leq \boldsymbol{0}^m$, and hence, we can derive the following bounds

$$\begin{aligned} \theta_{\mathrm{LP}}(\boldsymbol{\mu}) &= L(\bar{\boldsymbol{x}}_{\boldsymbol{\mu}}, \boldsymbol{\mu}) = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{\mu} + (\boldsymbol{c}^{\mathrm{T}} - \boldsymbol{\mu}^{\mathrm{T}}A)\bar{\boldsymbol{x}}_{\boldsymbol{\mu}} \\ &\leq \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^* + \boldsymbol{\mu}^{\mathrm{T}}(\boldsymbol{b} - A\boldsymbol{x}^*) = z^*_{\mathrm{LP}} + \underbrace{\boldsymbol{\mu}^{\mathrm{T}}}_{\geq \boldsymbol{0}^m} \underbrace{(\boldsymbol{b} - A\boldsymbol{x}^*)}_{\leq \boldsymbol{0}^m} \leq z^*_{\mathrm{LP}}. \end{aligned} \tag{2.6}$$

Thereby, for *any* fixed $\boldsymbol{\mu} \geq \boldsymbol{0}^m$, via the optimal solution $\bar{\boldsymbol{x}}_{\boldsymbol{\mu}}$ to (2.5), $L(\bar{\boldsymbol{x}}_{\boldsymbol{\mu}}, \boldsymbol{\mu})$ provides a lower bound on the primal optimal objective function value z^*_{LP}.

We now define the Lagrangian dual problem—in which we want to find the non-negative value of $\boldsymbol{\mu}$ for which $\min\limits_{\boldsymbol{x} \geq \boldsymbol{0}^n} L(\boldsymbol{x}, \boldsymbol{\mu})$ yields the tightest bound on z^*_{LP}—as to

$$\begin{aligned} \underset{\boldsymbol{\mu}}{\text{maximize}} \quad & \theta_{\mathrm{LP}}(\boldsymbol{\mu}) := \quad \boldsymbol{b}^{\mathrm{T}}\boldsymbol{\mu} + \min_{\boldsymbol{x} \geq \boldsymbol{0}^n} \left\{ (\boldsymbol{c}^{\mathrm{T}} - \boldsymbol{\mu}^{\mathrm{T}}A)\boldsymbol{x} \right\}, && \text{(2.7a)} \\ \text{subject to} \quad & \boldsymbol{\mu} \geq \boldsymbol{0}^m. && \text{(2.7b)} \end{aligned}$$

However, if any component of the vector $\boldsymbol{c}^{\mathrm{T}} - \boldsymbol{\mu}^{\mathrm{T}}A$ is negative, the minimization term in the objective function (2.7a) will equal $-\infty$, and such a solution will never be

optimal in the maximization problem with regard to $\boldsymbol{\mu}$. This motivates the inclusion of the additional constraint $\boldsymbol{c} - A^{\mathrm{T}}\boldsymbol{\mu} \geq \mathbf{0}^n$ to the program (2.7), in which case the second term in the objective function vanishes. We have arrived at the linear programming dual problem of (2.2), namely to

$$\underset{\boldsymbol{\mu}}{\text{maximize}} \quad \theta_{\text{LP}} = \boldsymbol{b}^{\mathrm{T}}\boldsymbol{\mu}, \tag{2.8a}$$

$$\text{subject to} \quad A^{\mathrm{T}}\boldsymbol{\mu} \leq \boldsymbol{c}, \tag{2.8b}$$

$$\boldsymbol{\mu} \geq \mathbf{0}^m. \tag{2.8c}$$

We refer to the decision variables $\boldsymbol{\mu} \in \mathbb{R}^m$ of the dual problem as a dual vector of the primal problem (2.2). The dual vector plays an important role in the simplex method, presented in next section, as well as in the column generation process, presented in Chapter 3. From the dual bounds (2.6) we see that for the optimal objective function value of (2.8), θ^*_{LP}, it holds that $\theta^*_{\text{LP}} \leq z^*_{\text{LP}}$. We shall soon see, however, that we can replace this inequality by equality.

First, we present a result on the necessary and sufficient conditions for global optimality for linear programs, as stated in [6, p. 252].

Theorem 1 (necessary and sufficient conditions for global optimality for LP) *For $\boldsymbol{x} \in \mathbb{R}^n$ to be an optimal solution to the linear program (2.2), it is both necessary and sufficient that (a), (b) and (c) hold, with*

(a) $\boldsymbol{x}$ is a feasible solution to (2.2);

(b) $\boldsymbol{\mu} \in \mathbb{R}^m$ is a feasible solution to (2.8); and

(c) the primal-dual pair $(\boldsymbol{x}, \boldsymbol{\mu})$ satisfies the complementarity conditions

$$x_j(c_j - \boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{A}_{\cdot j}) = 0, \qquad j = 1, \ldots, n, \tag{2.9a}$$

$$\mu_i(\boldsymbol{A}_{i\cdot}\boldsymbol{x} - b_i) = 0, \qquad i = 1, \ldots, m, \tag{2.9b}$$

where $\boldsymbol{A}_{\cdot j}$ is the jth column of the constraint matrix A, and $\boldsymbol{A}_{i\cdot}$ the ith row of A.

We return to the dual bounds (2.6), and furthermore assume that $\bar{\boldsymbol{\mu}}$ solves the dual problem (2.8) with optimal objective function value $\theta^*_{\text{LP}} = \boldsymbol{b}^{\mathrm{T}}\bar{\boldsymbol{\mu}}^*$. We realize that as a direct consequence of Theorem 1, it holds for a primal-dual pair $(\boldsymbol{x}^*, \bar{\boldsymbol{\mu}})$ that $\theta^*_{\text{LP}} = \boldsymbol{b}^{\mathrm{T}}\bar{\boldsymbol{\mu}} = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}^* = z^*_{\text{LP}}$, i.e., there is no gap between the optimal dual objective function value and the optimal primal objective function value. This is an important general property of linear programming problems, or even stronger, a general property of convex programming problems[4], among which LP programs are included.

[4]For readers unfamiliar with the concept of convexity and convex optimization programs—a key concept in the theory of optimization—see e.g. [6].

We move on to review the simplex method, a method to find solutions to linear programs fulfilling the optimality conditions (a)–(c) in Theorem 1, i.e., to find optimal solutions to linear programs.

2.4 A Method for Solving Linear Programs: The Simplex Method

Consider the LP problem (2.2), now given in standard form, as to

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad z_{\text{LP}} = \boldsymbol{c}^{\text{T}}\boldsymbol{x}, \tag{2.10a}$$

$$\text{subject to} \quad A\boldsymbol{x} = \boldsymbol{b}, \tag{2.10b}$$

$$\boldsymbol{x} \geq \boldsymbol{0}^n, \tag{2.10c}$$

where $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{c} \in \mathbb{R}^n$ and $\boldsymbol{b} \geq \boldsymbol{0}^m$. As mentioned in Section 2.1, the feasible region of a linear program is a polyhedron. Next we present two results that states that if there exists a finite optimal solution to (2.10), then there exists an optimal solution among the extreme points of the polyhedron describing the feasible region of the problem. The Representation Theorem and a theorem regarding the existence and properties of optimal solutions of LP programs are given as stated in [6, pp. 218–219].

Theorem 2 (Representation Theorem) *Let $P := \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{b};\ \boldsymbol{x} \geq \boldsymbol{0}^n\}$ and $Q := \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_k\}$ be the set of extreme points of P. Further, let $C := \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{0}^m;\ \boldsymbol{x} \geq \boldsymbol{0}^n\}$ and $R := \{\tilde{\boldsymbol{p}}_1, \dots, \tilde{\boldsymbol{p}}_l\}$ be the set of extreme directions of C.*

(a) P is nonempty and bounded if and only if Q is nonempty (and finite) and R is empty.

(b) P is unbounded if and only if R is nonempty (and finite).

(c) Every $\boldsymbol{x} \in P$ can be represented as the sum of a convex combination of the points in Q and a non-negative linear combination of the points in R, that is,

$$\boldsymbol{x} = \sum_{q=1}^{k} \lambda_q \boldsymbol{p}_q + \sum_{r=1}^{l} \tilde{\lambda}_r \tilde{\boldsymbol{p}}_r \tag{2.11}$$

for some $\lambda_1, \dots, \lambda_k \geq 0$ such that $\sum_{q=1}^{k} \lambda_q = 1$ and $\tilde{\lambda}_1, \dots, \tilde{\lambda}_l \geq 0$.

Theorem 3 (existence and properties of optimal solutions in LP) *Let the sets P, Q and R be defined as in Theorem 2 and consider the linear program to*

$$\underset{\boldsymbol{x}}{\text{minimize}} \quad z = \boldsymbol{c}^{\text{T}}\boldsymbol{x}, \tag{2.12a}$$

$$\text{subject to} \quad \boldsymbol{x} \in P. \tag{2.12b}$$

(a) This problem has a finite optimal solution if and only if P is nonempty and z is lower bounded on P, that is, if P is nonempty and $\boldsymbol{c}^{\mathrm{T}}\tilde{\boldsymbol{p}}_r \geq 0$ for all $\tilde{\boldsymbol{p}}_r \in R$.

(b) If the problem has a finite optimal solution, then there exists an optimal solution among the extreme points.

In the simplex method, the geometric notion of extreme points is represented by the algebraic concept of basic feasible solutions (BFS). We won't go into details here, but it can be shown that a point in the feasible region of a problem is a BFS if and only if it is an extreme point of P [4, pp. 89–90]. Hence, by traversing the BFS's of the problem, one traverses the extreme points where—from the statement (b) in Theorem 3—a finite optimal solution can be found, if one exists. In view of the LP problem (2.10), these BFS's are constructed by letting $n - m$ variables equal zero—*non-basic* variables—and letting the remaining m *basic* variables construct the BFS at hand, i.e. the extreme point we want to represent. The traversing between different BFS's is realized by exchanging, or pricing out, a basic variable for a non-basic one, using the concept of *reduced cost* of a variable. The reduced cost can be said to be a measure for the cost—with regard to the objective function—to increase the variable by a small amount. For a minimization problem, we want to find non-basic variables with non-positive reduced cost, eligible to enter the basic variables. Conversely, for a maximization problem, we look for non-basic variables with non-negative reduced cost. For details, see e.g. [4, 6]. We summarize the simplex algorithm below, referring to the optimality conditions stated in Theorem 1.

The algorithm initially finds a BFS to the primal problem, such that the condition (a) of Theorem 1 holds. By construction of the method, each BFS also produces a corresponding dual solution, which, however, may not be feasible. Moreover, by construction, every BFS, or more specific its corresponding primal-dual pair $(\boldsymbol{x}, \boldsymbol{\mu})$, satisfies the complementarity conditions (2.9), i.e. the condition (c) of Theorem 1 also holds. It can be shown, however, that only an optimal BFS—optimal extreme point—satisfies the dual feasibility condition (b). So to summarize, the simplex method starts from a primal feasible basis (representing a BFS) and can be viewed as to travel through dual infeasible solutions until dual feasibility is reached. We refer to this method as the primal simplex method.

The dual simplex method

A variation of the primal simplex method is the dual simplex method. This method is also adapted to the optimality conditions in Theorem 1, but seen from the *dual point of view*. In linear programming, the dual of the dual is the primal, so the terms primal

and dual are just relative, depending on what we consider to be the original problem. In the dual simplex algorithm, we start with a dual feasible basis and travel through primal infeasible bases until primal feasibility is reached.

When to use which of the two methods is quite problem-specific, but there are some situations when one generally could choose one method in preference to the other. Let's say we have some LP problem $\mathcal{P}_1^{\text{LP}}$ and solve it using the simplex algorithm, obtaining an optimal primal and dual vector, $\boldsymbol{x}_1^*$ and $\boldsymbol{\mu}_1^*$, respectively. Now, if we have an associated—associated in the sense that parts of its variables, constraints and objective function can be used to fully describe $\mathcal{P}_1^{\text{LP}}$—linear program, say $\mathcal{P}_2^{\text{LP}}$, we might want to make use of the known solution to $\mathcal{P}_1^{\text{LP}}$ when solving $\mathcal{P}_2^{\text{LP}}$, if possible.

If $\mathcal{P}_2^{\text{LP}}$ is attained by adding *new variables* to original problem $\mathcal{P}_1^{\text{LP}}$, then the existing optimal vector $\boldsymbol{x}_1^*$ will be feasible also in $\mathcal{P}_2^{\text{LP}}$, and this vector could be used as an initial BFS. As new variables in the primal corresponds to new constraints or *cuts* in the dual, the existing dual vector $\boldsymbol{\mu}_1^*$ may not be feasible in the new problem $\mathcal{P}_2^{\text{LP}}$. In such a case, choosing the primal simplex method over the dual simplex method could be sensible.

If, on the other hand, $\mathcal{P}_2^{\text{LP}}$ is attained by adding *new constraints*, or cuts, to the original problem $\mathcal{P}_1^{\text{LP}}$, the situation would be reversed, and the existing dual solution $\boldsymbol{\mu}_1^*$ would still be feasible in (the dual of) $\mathcal{P}_2^{\text{LP}}$, while the existing primal solution $\boldsymbol{x}_1^*$ may not. In this case, we might choose the dual simplex method in preference to the primal method, using $\boldsymbol{\mu}_1^*$ as an initial BFS. We will return to this discussion when presenting the BAB algorithm in Chapter 4.

Chapter 3

Dantzig-Wolfe Decomposition and Column Generation for Linear Programming

In the early post-war years during which the simplex were introduced, the new field of Operations Research (OR) began to grow rapidly. Linear programming was to become a central part of this field, and by such, its intimate relationship with methods for solving large systems of linear equations. It's appropriate to clarify the meaning of the term *"large"* in this context. In 1947, the solving of Stiegler's diet problem—a minimization problem containing 77 variables embedded in nine inequality constraints—took nine clerks 120 clerk-days of work, using hand-operated desk calculators. At the time, this was considered as a very large problem, the first *"large-scale"* LP problem ever solved using the simplex method. This period, however, coincided with the development—fueled by the importance of advancing cryptology during World War II—of the first electronic computers, and the capacity to solve large-scale linear systems, and hence the maximum size for which a LP was solvable, increased rapidly. A few years into the '50s, linear programs with a couple of hundred constraints was solvable. One of many linear programming pioneers, Alex Orden (as cited in [10]) roughly estimates that the maximum number of constraints in a solvable LP has grown by a factor 10 each decade. Comparatively a good estimation, as in the beginning of the '00s, the largest LP ever reported to be solved contained 12.5 million constraints and 25 million variables [11, 12, 13].

Now, bringing to an end the reminiscing of the youth of linear programming, indeed OR and LP has developed hand in hand with computers. But true to form, mathematicians engaged in the theory of linear programming have not been idly waiting for subsequent increases in computer performance to attack seemingly in-

tractable LP problems. There have been several developments of techniques to handle size-wise difficult LP's, based on the idea of manipulating the original problem into a more solver-friendly form. One such method is *column generation* (CG), and the closely related *Dantzig-Wolfe decomposition* principle.

In this chapter, we introduce the general idea of column generation, followed by an overview of the Dantzig-Wolfe decomposition principle. Finally, Section 3.3 will relate column generation to integer programming, the main area in which CG will be put to use in the scope of this thesis. Indeed, even if column generation was initially intended for pure linear programs it eventually became an indispensable tool in the context of integer programming. As we shall see in Chapter 4, the method plays an important role in the BAP algorithm for solving ILP/MILP problems.

For a more exhaustive introduction to the subject, see [14, 15].

3.1 General Column Generation

Consider the linear program to

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{MP}} = \sum_{i \in I} c_i \lambda_i, & \text{(3.1a)} \\ \text{subject to} \quad & \sum_{i \in I} \boldsymbol{a}_i \lambda_i \geq \boldsymbol{b}, & \text{(3.1b)} \\ & \lambda_i \geq 0, \quad i \in I, & \text{(3.1c)} \end{aligned}$$

where $\boldsymbol{a}_i$, $\boldsymbol{b} \in \mathbb{R}^m$, c_i, $\lambda_i \in \mathbb{R}$ for $i \in I = \{1, \ldots, n\}$, and n is the number of variables in the program. We will refer to this problem as the *master problem* (MP), with an optimal objective function value z^*_{MP}, and assume that the number n of variables is much larger than the number m of constraints in (3.1b). Associated with each variable λ_i is a constraint coefficient column $\boldsymbol{a}_i$, appearing in the constraints (3.1b), aswell as an objective function cost coefficient c_i, appearing in the objective function (3.1a). We will refer to $[c_i \;\; \boldsymbol{a}_i^{\text{T}}]^{\text{T}}$, $i \in I = \{1, \ldots, n\}$, as *coefficient columns* or simply *columns* of the master problem, where each column is associated with one of the MP variables. Consequently, we will sometimes refer to the set of variables as the *column pool* of MP.

In many practical situations (3.1) may describe a LP with a huge number of variables λ_i, and explicitly enumerating them all (along with their associated columns $[c_i \;\; \boldsymbol{a}_i^{\text{T}}]^{\text{T}}$) may be computationally intractable, i.e. too large a task for computer resources available to complete. Even just storing all columns may be impossible, yet alone trying to solve the associated LP. However, in any basic feasible solution to the problem (3.1) most of the variables will be non-basic, as $n = |I| \gg m$, and hence set to zero. This means—in theory—that to solve the problem, we need only to consider

a relatively small subset $I' \subset I$ of the variables, thereby considering an LP in which we regain tractability. We substitute I by I' in (3.1), resulting in the following *restricted master problem* (RMP) to

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{RMP}} = \sum_{i \in I'} c_i \lambda_i, && (3.2a) \\ \text{subject to} \quad & \sum_{i \in I'} \boldsymbol{a}_i \lambda_i \geq \boldsymbol{b}, && (3.2b) \\ & \lambda_i \geq 0, \quad i \in I'. && (3.2c) \end{aligned}$$

Let $\bar{z}_{\text{RMP}}$ denote the optimal objective function value to RMP. Given any feasible solution $\boldsymbol{\lambda}_{\text{RMP}} \in \mathbb{R}^{|I'|}$ to (3.2) we can—as we're considering a subset of the variables in MP—construct a solution $[\boldsymbol{\lambda}_{\text{RMP}}^{\text{T}}, \ (\mathbf{0}^{|I \setminus I'|})^{\text{T}}]^{\text{T}}$ that is feasible in the master problem (3.1). Hence, the optimal objective function value to the RMP gives an upper bound on the MP optimal objective function value, i.e., $z_{\text{MP}}^* \leq \bar{z}_{\text{RMP}}$. Furthermore, as both programs contain the same number of constraints, we realize that another important connection between the MP and the RMP is that any feasible dual solution $\boldsymbol{\mu} \in \mathbb{R}^m$ to the RMP is a feasible dual solution also in the master problem. Hence, an optimal dual solution to the RMP yields information that can be usable also from the viewpoint of the full master program.

Now, the general idea of column generation is to iteratively add *improving* columns[1] to the RMP, until its optimal solution, $\bar{\boldsymbol{\lambda}}_{\text{RMP}}$, is optimal also in the MP, i.e., $\boldsymbol{\lambda}_{\text{MP}}^* = [\bar{\boldsymbol{\lambda}}_{\text{RMP}}^{\text{T}}, \ (\mathbf{0}^{|I \setminus I'|})^{\text{T}}]^{\text{T}}$. As explained in Section 2.4, in the process of traversing the extreme points of the problem's feasible region in the simplex method—realized in practice by iteratively examining basic feasible solutions—non-basic variables with non-positive *reduced costs* are iteratively entered into the basis, exchanged for priced-out variables, until an optimal basis is found. In column generation, given a dual optimal solution $\bar{\boldsymbol{\mu}} \in \mathbb{R}^m$ to the RMP, the analogous procedure for finding prospective variables—variables that are not yet included in the subset I'—is achieved by solving the *subproblem*

$$\bar{c}^* := \underset{i \in I}{\text{minimize}} \left\{ c_i - \bar{\boldsymbol{\mu}}^{\text{T}} \boldsymbol{a}_i \right\}. \tag{3.3}$$

If $\bar{c}^* < 0$, we add to the RMP column pool I' the corresponding optimal solution $[c_{i^*} \ \boldsymbol{a}_{i^*}^{\text{T}}]^{\text{T}}$ as a new coefficient column, along with an associated variable λ_{i^*}. The RMP is thereafter re-solved and the procedure is repeated iteratively until $\bar{c}^* \geq 0$, in which

[1]Improving columns (or variables; each new column $[c_i \ \boldsymbol{a}_i^{\text{T}}]^{\text{T}}$ is associated with a new variable λ_i, and vice versa) in the sense that the RMP optimal objective function value will steadily improve as generated columns are added to the RMP column pool I'.

case no more improving columns can be added and we've reached a RMP optimal solution that is optimal also in the MP.

Solving the subproblem (3.3) directly seems no better than just attacking the full MP (3.1) using the simplex algorithm, as we still have to consider the full size of I. However—as we shall see, for example, in the next section—in most purviews of column generation in practice, the elements of I are not explicitly enumerated, but instead implicitly described as feasible points $\boldsymbol{a} \in \mathcal{A}$ in an optimization problem, and hence the process of finding columns with negative reduced cost is handled implicitly. We write the implicit column generating subproblem as to

$$\bar{c}^* := \underset{\boldsymbol{a} \in \mathcal{A}}{\text{minimize}} \left\{ c(\boldsymbol{a}) - \bar{\boldsymbol{\mu}}^{\mathrm{T}} \boldsymbol{a} \right\}, \tag{3.4}$$

where $c(\boldsymbol{a}) = \text{cost}(\boldsymbol{a})$.

Now, as each improving column $[c(\boldsymbol{a}_i) \ \boldsymbol{a}_i^{\mathrm{T}}]^{\mathrm{T}}$ is added to the RMP accompanied by the weight variable λ_i, we can—in addition to the iteratively decreasing upper bound on the MP optimal objective function value z^*_{MP}—in most cases derive also a lower bound for z^*_{MP}. During any iteration, given the RMP optimal objective function value $\bar{z}_{\mathrm{RMP}}$ and the smallest reduced cost $\bar{c}^*$, if we know an upper bound on the sum of the column weight variables λ_i for an optimal solution of the MP, $\sum_{i \in I} \lambda_i \leq \beta$, we also have lower bound on z^*_{MP}

$$\bar{z}_{\mathrm{RMP}} + \beta \bar{c}^* \leq z^*_{\mathrm{MP}}, \tag{3.5}$$

as we can't possibly improve $\bar{z}_{\mathrm{RMP}}$ any more than β times the smallest reduced cost.

We wrap up with a note on complexity. Assuming cycling is dealt with—a phenomenom where, in the simplex method, the same basic and non-basic variables keeps getting exchanged, leading to an infinite loop, comparable to generating the same columns over and over in CG—column generation inherits the benefits of the simplex method. The algorithm terminates, in practice, at an MP optimal solution in polynomial time, as long as the subproblem is solvable abiding the same complexity, which is naturally the case when the subproblem is a pure linear program [15].

3.2 The Dantzig-Wolfe Decomposition Principle

Consider the linear program to

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & z_{\mathrm{LP}} = \boldsymbol{c}^{\mathrm{T}} \boldsymbol{x}, && \text{(3.6a)} \\
\text{subject to} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, && \text{(3.6b)} \\
& D\boldsymbol{x} \geq \boldsymbol{d}, && \text{(3.6c)} \\
& \boldsymbol{x} \geq \boldsymbol{0}^n, && \text{(3.6d)}
\end{aligned}$$

where $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, $\boldsymbol{d} \in \mathbb{R}^p$, $A \in \mathbb{R}^{m \times n}$ and $D \in \mathbb{R}^{p \times n}$. Let $P := \{\boldsymbol{x} \in \mathbb{R}^n \mid D\boldsymbol{x} \geq \boldsymbol{d};\ \boldsymbol{x} \geq \boldsymbol{0}^n\}$ and assume that the polyhedron P is bounded. From Theorem 2 we know that every $\boldsymbol{x} \in P$ can be represented as the sum of a convex combination of its extreme points $\boldsymbol{p}_q \in Q$ and a non-negative linear combination of the extreme directions $\tilde{\boldsymbol{p}}_r \in R$ of P, that is,

$$\boldsymbol{x} = \sum_{q \in Q} \lambda_q \boldsymbol{p}_q + \sum_{r \in R} \tilde{\lambda}_r \tilde{\boldsymbol{p}}_r, \tag{3.7}$$

for some $\lambda_q \geq 0$, $q \in Q$, such that $\sum_{q \in Q} \lambda_q = 1$ and $\tilde{\lambda}_r \geq 0$, $r \in R$. As we have assumed that P is bounded, $R = \emptyset$ holds, and (3.7) simplifies to $\boldsymbol{x} = \sum_{q \in Q} \lambda_q \boldsymbol{p}_q$. We substitute this expression for each $\boldsymbol{x}$ in (3.6), which—due to the construction of the $\boldsymbol{x}$ variable in (3.7)—makes the constraints (3.6c–3.6d) redundant. We've now reached a linear program with decision variables $\lambda_q \geq 0$, $q \in Q$, given as

$$\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{MP}} = \sum_{q \in Q} \boldsymbol{c}^{\text{T}} \boldsymbol{p}_q \lambda_q, && \text{(3.8a)}\\
\text{subject to} \quad & \sum_{q \in Q} A \boldsymbol{p}_q \lambda_q \geq \boldsymbol{b}, && \text{(3.8b)}\\
& \sum_{q \in Q} \lambda_q = 1, && \text{(3.8c)}\\
& \lambda_q \geq 0, \quad q \in Q. && \text{(3.8d)}
\end{aligned}$$

We recognize that this problem is expressed in the same way as the column generation MP (3.1) introduced in Section 3.1[2], introduced in the previous section, with the addition of the *convexity constraint* (3.8c). The convexity constraint make certain that we use a convex combination of the extreme points $\{\boldsymbol{p}_q\}_{q \in Q}$ of P, implicitly ensuring that the constraints (3.6c–3.6d) of the original LP are not violated. Henceforth, we will refer to (3.8) as the MP in Dantzig-Wolfe decomposition.

Now, as explained in Section 3.1, the MP—with an optimal objective value z^*_{MP}—may be too large to solve directly using the simplex method, due to computational and storage limitations. So instead of considering the full set Q in (3.8), that is, enumerating all extreme points $\{\boldsymbol{p}_q\}_{q \in Q}$ of P, we consider a subset $Q' \subset Q$ of the extreme

[2]Applying the linear transformations $c_q = \boldsymbol{c}^T \boldsymbol{p}_q$ and $\boldsymbol{a}_q = A\boldsymbol{p}_q$, $q \in Q$ and further omitting the convexity constraint (3.8c) transforms (3.8) into the general column generation MP (3.1).

points, and substitute Q by Q' in (3.8). This gives us the following RMP to

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{RMP}} = \sum_{q \in Q'} \boldsymbol{c}^{\mathrm{T}} \boldsymbol{p}_q \lambda_q, && (3.9a) \\ \text{subject to} \quad & \sum_{q \in Q'} A \boldsymbol{p}_q \lambda_q \geq \boldsymbol{b}, && (3.9b) \\ & \sum_{q \in Q'} \lambda_q = 1, && (3.9c) \\ & \lambda_q \geq 0, \quad q \in Q'. && (3.9d) \end{aligned}$$

As before, we let $\bar{z}_{\text{RMP}}$ denote the optimal objective function value to the RMP, which provides an upper bound on the MP optimal objective function value, i.e., $z^*_{\text{MP}} \leq \bar{z}_{\text{RMP}}$. Now, given a dual optimal solution $(\bar{\boldsymbol{\mu}}, \bar{v})$ to the RMP, where $\bar{\boldsymbol{\mu}}$ corresponds to the constraints (3.9b) and $\bar{v}$ to the convexity constraint (3.9c), we solve the *subproblem*

$$\bar{c}^* := \underset{\boldsymbol{x} \in P}{\text{minimize}} \left\{ (\boldsymbol{c}^{\mathrm{T}} - \bar{\boldsymbol{\mu}}^{\mathrm{T}} A)\boldsymbol{x} - \bar{v} \right\}. \tag{3.10}$$

As long as the reduced cost is negative, i.e., $\bar{c}^* < 0$, we add the corresponding subproblem optimal solution $\bar{\boldsymbol{x}}$ as a column to the column pool Q', along with its associated objective function cost coefficient $\boldsymbol{c}^{\mathrm{T}}\bar{\boldsymbol{x}}$. From LP theory, each such column is naturally—as (3.10) is a linear program with a bounded feasible region P—another extreme point $\boldsymbol{p}_q$ of P added to the subset of extreme points Q' of P.

From the general lower bound on the MP optimal objective function value, given by (3.5), we can construct a lower bound on the Dantzig-Wolfe MP optimal objective function value z^*_{MP} to (3.8). We note, however, that due to additional convexity constraints (3.8c) we can replace β in (3.5) by 1, yielding the bounds

$$\bar{z}_{\text{RMP}} + \bar{c}^* \leq z^*_{\text{MP}} \leq \bar{z}_{\text{RMP}}. \tag{3.11}$$

We proceed to a special case of DW decomposition, where the constraint matrix D in the original problem (3.6) have a special structure.

A block-diagonal structure of the constraint matrix

We now consider the case where the constraint matrix D in (3.6) has a block-diagonal structure. We write the associated constant vectors $\boldsymbol{d}$ and $\boldsymbol{c}$ and the decision variables vector $\boldsymbol{x}$ on a decomposed form, connected with the special structure of D

$$D = \begin{bmatrix} D^1 & 0 & 0 & \dots & 0 \\ 0 & D^2 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & D^K \end{bmatrix}, \quad \boldsymbol{d} = \begin{bmatrix} \boldsymbol{d}^1 \\ \boldsymbol{d}^2 \\ \vdots \\ \boldsymbol{d}^K \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}^1 \\ \boldsymbol{c}^2 \\ \vdots \\ \boldsymbol{c}^K \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}^1 \\ \boldsymbol{x}^2 \\ \vdots \\ \boldsymbol{x}^K \end{bmatrix},$$

with the additional constraints matrix A decompositioned accordingly, $A = [A^1\, A^2 \ldots A^\kappa]$, where $A^k \in \mathbb{R}^{m \times |\boldsymbol{x}^k|}$, $k = 1, \ldots, \kappa$. We can now write the LP (3.6) as

$$\begin{aligned}
\min_{\boldsymbol{x}^1,\ldots,\boldsymbol{x}^\kappa}.\quad & z_{\mathrm{LP}} = (\boldsymbol{c}^1)^{\mathrm{T}}\boldsymbol{x}^1 + (\boldsymbol{c}^2)^{\mathrm{T}}\boldsymbol{x}^2 + \ldots + (\boldsymbol{c}^\kappa)^{\mathrm{T}}\boldsymbol{x}^\kappa, && (3.12a)\\
\text{s. t.}\quad & A^1\boldsymbol{x}^1 + A^2\boldsymbol{x}^2 + \ldots + A^\kappa\boldsymbol{x}^\kappa \geq \boldsymbol{b}, && (3.12b)\\
& D^1\boldsymbol{x}^1 \geq \boldsymbol{d}^1, && (3.12c)\\
& D^2\boldsymbol{x}^2 \geq \boldsymbol{d}^2, && (3.12d)\\
& \ddots \quad \vdots && (3.12e)\\
& D^\kappa\boldsymbol{x}^\kappa \geq \boldsymbol{d}^\kappa, && (3.12f)\\
& [(\boldsymbol{x}^1)^{\mathrm{T}}\ (\boldsymbol{x}^2)^{\mathrm{T}} \ldots (\boldsymbol{x}^\kappa)^{\mathrm{T}}]^{\mathrm{T}} \geq \boldsymbol{0}^n. && (3.12g)
\end{aligned}$$

We see that only the constraints (3.12b) contained in the A matrix connects variables from the different subsets $\{\boldsymbol{x}^k\}$, $k = 1, ..., \kappa$, whereas each of the constraints (3.12c–3.12f) can be represented as independent polytopes $P^k := \{\boldsymbol{x}^k \in \mathbb{R}^{|\boldsymbol{x}^k|} \mid D^k\boldsymbol{x}^k \geq \boldsymbol{d}^k;\ \boldsymbol{x}^k \geq \boldsymbol{0}^{|\boldsymbol{x}^k|}\}$, $k = 1, \ldots, \kappa$. From the initial assumption of the boundedness of P, each P^k is bounded, and so, as in the previous section, we can implicitly represent elements in these polytopes as convex combinations of the extreme points of each polytope, i.e., for each $k \in \{1, \ldots, \kappa\}$,

$$\boldsymbol{x}^k = \sum_{q \in Q^k} \lambda_q^k \boldsymbol{p}_q^k, \qquad \sum_{q \in Q^k} \lambda_q^k = 1, \qquad \boldsymbol{\lambda}_q^k \geq \boldsymbol{0}^{|Q^k|}. \tag{3.13}$$

Let $K := \{1, \ldots, \kappa\}$. Following the Dantzig-Wolfe decomposition procedure we've just covered, we substitute the expression of each $\boldsymbol{x}^k$, $k \in K$, into the LP (3.12), yielding the following MP to

$$\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{minimize}}\quad & z_{\mathrm{MP}} = \sum_{k \in K}\sum_{q \in Q^k} (\boldsymbol{c}^k)^{\mathrm{T}}\boldsymbol{p}_q^k\lambda_q^k, && (3.14a)\\
\text{subject to}\quad & \sum_{k \in K}\sum_{q \in Q^k} A^k\boldsymbol{p}_q^k\lambda_q^k \geq \boldsymbol{b}, && (3.14b)\\
& \sum_{q \in Q^k} \lambda_q^k = 1, \quad k \in K, && (3.14c)\\
& \lambda_q^k \geq 0, \quad k \in K, q \in Q. && (3.14d)
\end{aligned}$$

For each subsystem $k \in K$, we consider only a subset $\bar{Q}^k \subset Q^k$ of the MP variables to construct an RMP. Given a dual optimal solution $(\bar{\boldsymbol{\mu}}, \bar{v}^1, \ldots, \bar{v}^\kappa)$ to the RMP—where $\bar{\boldsymbol{\mu}}$ as before corresponds to the connecting constraints (3.14b) and $\bar{v}^k$, $k \in K$, correspond to the κ separate convexity constraints (3.14c)—new columns are generated to the RMP column pools $\bar{Q}^k$ by solving κ different subproblems to

$$\bar{c}^{k*} := \underset{\boldsymbol{x}^k \in P^k}{\text{minimize}} \left\{ \left((\boldsymbol{c}^k)^{\mathrm{T}} - \bar{\boldsymbol{\mu}}^{\mathrm{T}} A \right) \boldsymbol{x}^k - \bar{v}^k \right\}, \quad k \in K. \tag{3.15}$$

The column generation procedure for solving the RMP followed by solving the κ subproblems is repeated until $\bar{c}^{k*} \geq 0$, for $k \in K$, whence the process is terminated.

From the bounds on the MP optimal objective function value z^*_{MP}, in the case of a single subproblem, given by (3.11), we can deduce analogous bounds for the case of multiple subproblems. Given an iteration's current RMP optimal objective function value $\bar{z}_{\text{RMP}}$, we can't possibly improve $\bar{z}_{\text{RMP}}$ by more than the sum of the reduced costs from the κ subproblems. That is,

$$\bar{z}_{\text{RMP}} + \sum_{k \in K} \bar{c}^{k*} \leq z^*_{\text{MP}} \leq \bar{z}_{\text{RMP}}. \tag{3.16}$$

A problem with the constraint structure of (3.12) is said to have a *block-angular* full constraint matrix $\left[\begin{smallmatrix} A \\ D \end{smallmatrix}\right]$, and tends to be especially well suited for DW decomposition and column generation [15].

We move on to study how to use Dantzig-Wolfe decomposition in the sense of integer programming.

3.3 Extension to Integer Programming

In this section we will extend the ideas of Dantzig-Wolfe decomposition and column generation to integer programming, starting with general bounded ILP problems, thereafter concluding the section with the special case of binary linear programs.

We once again consider (3.6), now with additional integer requirements on the decision variables, yielding the following ILP problem to

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad z_{\text{ILP}} = \quad & \boldsymbol{c}^T \boldsymbol{x}, && \text{(3.17a)}\\
\text{subject to} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, && \text{(3.17b)}\\
& D\boldsymbol{x} \geq \boldsymbol{d}, && \text{(3.17c)}\\
& \boldsymbol{x} \in \mathbb{Z}^n_+, && \text{(3.17d)}
\end{aligned}$$

where coefficient vectors and matrices are as defined for the problem (3.6) in Section 3.2. Let $P := \{\boldsymbol{x} \in \mathbb{R}^n \mid D\boldsymbol{x} \geq \boldsymbol{d}\}$ and set $X = P \cap \mathbb{Z}^n_+$. As hinted in the introductory remark of this section, we will assume that X is bounded. Now, there exists several different approaches to decompose integer linear programs, but in our context of Dantzig-Wolfe decomposition of linear programs, we will focus on *discretization*, a corresponding decomposition procedure for integer programs. We begin by stating a result as presented in [14, pp. 1011–1012], regarding the representation of points in spaces constructed by the intersection of a linear (convex) region and an integer space.

Theorem 4 (A representation theorem for integer linear programming) *Let* $P := \{\boldsymbol{x} \in \mathbb{R}^n \mid D\boldsymbol{x} \geq \boldsymbol{d}, \boldsymbol{x} \geq \boldsymbol{0}^n\} \neq \emptyset$ *and* $X = P \cap \mathbb{Z}^n_+ \neq \emptyset$*. Then, there exists a finite set of integer points* $\{\boldsymbol{p}_q\}_{q\in Q} \subseteq X$ *and a finite set of integer rays* $\{\tilde{\boldsymbol{p}}_r\}_{r\in R}$ *of* P *such that*

$$X = \left\{ \boldsymbol{x} \in \mathbb{R}^n_+ \,\middle|\, \boldsymbol{x} = \sum_{q\in Q} \boldsymbol{p}_q \lambda_q + \sum_{r\in R} \boldsymbol{p}_r \tilde{\lambda}_r,\ \sum_{q\in Q} \lambda_q = 1,\ \boldsymbol{\lambda} \in \mathbb{Z}^{|Q|}_+,\ \tilde{\boldsymbol{\lambda}} \in \mathbb{Z}^{|R|}_+ \right\}. \tag{3.18}$$

Since we have assumed that X is bounded, the second term in the representation (3.18) of the elements $\boldsymbol{x}$ in X—describing the extreme rays $\{\tilde{\boldsymbol{p}}_r\}_{r\in R}$ of P—will vanish. We reach the trivial result that any $\boldsymbol{x} \in X$ can be represented as *one* of the integer points in X. As we shall see this representation is, however, useful when relaxing the integer requirements on the λ_q variables. We substitute $\boldsymbol{x}$ in (3.17) by the representation (3.18), noting that $R = \emptyset$, yielding the following integer MP to

$$\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{MP}} = \sum_{q\in Q} \boldsymbol{c}^T \boldsymbol{p}_q \lambda_q, && (3.19\text{a})\\
\text{subject to} \quad & \sum_{q\in Q} A\boldsymbol{p}_q \lambda_q \geq \boldsymbol{b}, && (3.19\text{b})\\
& \sum_{q\in Q} \lambda_q = 1, && (3.19\text{c})\\
& \lambda_q \in \mathbb{Z}_+, \quad q \in Q. && (3.19\text{d})
\end{aligned}$$

We perform a continuous relaxation of the integer requirements (3.19d), reaching a linear MP, say $\mathcal{P}^{\text{LP}}_{\text{MP}}$, in appearance identical to the general Dantzig-Wolfe MP (3.8). The difference is that in $\mathcal{P}^{\text{LP}}_{\text{MP}}$ columns $\{\boldsymbol{p}_q\}_{q\in Q}$ describe (integral) points in X, which may be inner points as well as extreme points to the polyhedron P, whereas in the pure linear case (3.8) all columns correspond to extreme points of the feasible region polyhedron P. As explained in the previous section, we consider only subset $Q' \subset Q$ of these columns—yielding a RMP $\mathcal{P}^{\text{LP}}_{\text{RMP}}$—and use a pricing subproblem to generate columns to the RMP. So, given a dual optimal solution $(\bar{\boldsymbol{\mu}}, \bar{v})$ to $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, we solve the *integer* linear subproblem

$$\bar{c}^* := \underset{\boldsymbol{x}\in X}{\text{minimize}} \left\{ (\boldsymbol{c}^T - \bar{\boldsymbol{\mu}}^T A)\boldsymbol{x} - \bar{v} \right\}. \tag{3.20}$$

The subproblem need not always be solved to optimality—even if it's naturally easier to solve than the original ILP (3.17)—but can be used, when possible, to heuristically generate columns with negative reduced cost. When no more improving columns can be added, the column generation process is terminated.

A justified question is how this will help in the solving of the original integer program (3.17). One could of course naïvely hope that the optimal solution to (3.19)

is contained among the generated columns $\{\boldsymbol{p}_q\}_{q\in Q'}$ in $\mathcal{P}^{\text{LP}}_{\text{RMP}}$. Indeed, finding the *best* column—best with regard to the *integer* RMP[3] $\mathcal{P}^{\text{ILP}}_{\text{MP}}$—from the subset column pool Q' would indeed seem as an easier task than solving the original program, and as we shall see in the next chapter, this can readily be done using the BAB algorithm. However, this best column, if it even exists, is generally not optimal in (3.19). In fact, there's even the possibility that among all generated columns none is feasible with regard to the integer RMP. Consequently, even if this procedure for some problem instances could be a way to acquire relatively good integer feasible solutions fast, it's not the main purpose of the discretization and column generation process for integer programs. Instead we realize the method's advantage when considering the following property.

Let z^*_{ILP} be the optimal objective function value of the original ILP (3.17), and let $z^{(\text{cont})*}_{\text{ILP}}$ be the optimal objective function value of its continuous relaxation. Similarly, let z^*_{MP} be the optimal objective function value of the integer MP (3.19), and $z^{(\text{cont})*}_{\text{MP}}$ be the optimal objective function value its continuous relaxation. We define the *integrality gaps* $\Gamma^{(\text{cont})}_{\text{ILP}} = z^*_{\text{ILP}} - z^{(\text{cont})*}_{\text{ILP}}$ and $\Gamma^{(\text{cont})}_{\text{MP}} = z^*_{\text{MP}} - z^{(\text{cont})*}_{\text{MP}}$, for which it holds that[4] $\Gamma^{(\text{cont})}_{\text{MP}}, \Gamma^{(\text{cont})}_{\text{ILP}} \geq 0$. The integrality gap for the integer MP, $\Gamma^{(\text{cont})}_{\text{MP}}$, is generally smaller than the corresponding gap for the original ILP, $\Gamma^{(\text{cont})}_{\text{ILP}}$, i.e., generally $\Gamma^{(\text{cont})}_{\text{MP}} < \Gamma^{(\text{cont})}_{\text{ILP}}$ [16]. We say that the integer MP (3.19) have a *tighter continuous relaxation*, or *tighter LP relaxation*[5], than has the original ILP (3.17). As we shall see in Chapter 4, a tight LP relaxation of an integer program is an ideal setup for the BAB algorithm. Furthermore, with discretizations' natural connection to column generation, the method's main strength comes into light when considering the continuous relaxation of the integer RMP $\mathcal{P}^{\text{ILP}}_{\text{RMP}}$ within a BAB framework, commonly named *ILP column generation* or *branch-and-price* (BAP). We will return to this subject in Chapter 4.

We wrap up this section with a note on the special case of binary integer programming problems.

A note on the special case of binary linear programming

Consider the ILP program (3.17), but assume that the integrality constraint (3.17d) has been replaced by the tighter constraint $\boldsymbol{x} \in \{0,1\}^n$. As before, let $P := \{\boldsymbol{x} \in$

[3]Referring to the LP RMP $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ with integer requirements of the λ_q variables restored, $\mathcal{P}^{\text{ILP}}_{\text{RMP}}$.

[4]Recall that the continuous relaxation of an integer minimization problem has an optimal objective function value that is less or equal to the optimal objective function value of the associated integer problem; see Section 2.2.

[5]Some literature consistently refer to the continuous relaxation of an ILP as its *LP relaxation*. In the scope of this thesis, however, the intended relaxation is always referred to as the *continuous relaxation* of an integer linear program.

$\mathbb{R}^n \mid D\boldsymbol{x} \geq \boldsymbol{d}\}$ and now set $X = P \cap \{0,1\}^n$. Clearly X is bounded, and moreover—as X is constructed as an intersection between a linear region and the n-dimensional binary space, $\{0,1\}^n$ or $\mathbb{B}^n$—any point in X is an extreme point of the convex hull[6] of X, $\text{conv}(X)$. Moreover, naturally any extreme point of $\text{conv}(X)$ is integral. Hence, any integer valued point in the the continuous relaxation of X, say $X^{(\text{cont})} = P \cap [0,1]^n$, is an extreme point of $\text{conv}(X)$.

If we perform the discretization procedure described in this section on such a BLP problem, we will attain pure linear RMP, $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, with an associated BLP subproblem, $\mathcal{P}^{\text{BLP}}_{\text{Sub}}$, that have quite nice properties. We consider our discussion above and look at the continuous relaxation of $\mathcal{P}^{\text{BLP}}_{\text{Sub}}$—denote it $\mathcal{P}^{\text{LP}}_{\text{Sub}}$—in which we replace the feasible region X by $X^{(\text{cont})}$. Ideally $X^{(\text{cont})}$ would equal $\text{conv}(X)$, in which case the LP optimal solution would be optimal also in the original binary subproblem. This is generally not the case, so we would like to add some cuts (constraints) $\tilde{D}\boldsymbol{x} \geq \tilde{\boldsymbol{d}}$ to $X^{(\text{cont})}$ such that $X^{(\text{cont})} \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \tilde{D}\boldsymbol{x} \geq \tilde{\boldsymbol{d}}\}$ contains exactly $\text{conv}(X)$, whereafter the LP optimal solution given this region—readily obtained using e.g. the simplex method—provides us with the aforementioned binary subproblem optimal solution as well. In practice it suffices that $X^{(\text{cont})} \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \tilde{D}\boldsymbol{x} \geq \tilde{\boldsymbol{d}}\}$ exactly resembles $\text{conv}(X)$ only in a subregion of it, a region containing a finite number of extreme points of $\text{conv}(X)$[7]. One method to find these cuts, one that is especially suited for binary linear programs, is the BAB algorithm.

The essence of this end note is that if applying the discretization procedure on a BLP (or a mixed binary linear program (MBLP)), we will attain a BLP subproblem that is quite suited to be solved using LP based techniques. We proceed to Chapter 4, to investigate two such techniques, the LP based BAB algorithm, and the advanced BAP algorithm.

[6] The convex hull for any set of points X is the smallest convex set that includes X.

[7] Remember, due to boundedness of binary programming and the convexity property of it's continuous relaxation, any optimal solution to $\mathcal{P}^{\text{LP}}_{\text{Sub}}$ constrained to $\text{conv}(X)$ will reside within a finite set of (possibly degenerate) extreme points of $\text{conv}(X)$.

Chapter 4

Methods for Attaining Integer Valued Solutions from Linear Programming Based Column Generation

Following the introduction of linear programming and the simplex method in the late '40s, it was not before long that the mathematical discipline of discrete optimization was born, most notably branched into combinatorial optimization and integer programming. Topics that would later be identified and categorized as members of discrete optimization had been independently studied in the preceding decades—shortest spanning tree, optimum assignment, the travelling salesman problem and so on—but it was only now that they naturally gathered under the same discipline [17]. Consequently, the common interest of the subject grew, and with increased research came the desire and hope for a general automatic routine for solving discrete optimization problems, comparable with the simplex method for solving pure linear programs. In 1960 A. H. Land and A. G. Doig published their work on a proposal for such an algorithm, one that is nowadays famously referred to as the *branch-and-bound* (BAB) algorithm [18]. A few decades later—following by the combination of BAB more advanced linear programming techniques—the more refined *branch-and-price* (BAP) algorithm was born, most notably by the work of C. Barnhart et al. [19] and F. Vanderbeck and L. A. Wolsey [20].

In this chapter we will review these two famous algorithms for finding optimal or near-optimal solutions to integer linear programs, focusing on 0–1 programs. In Section 4.1 we present the BAB algorithm in the sense of solving a general BLP problem. The following section covers the subject in the context of column generation and Dantzig-Wolfe decomposition. Section 4.3 proceeds by introducing the advanced BAP algorithm—a direct offspring of the BAB algorithm—in which column genera-

tion holds a key position.

For an overview of the BAB algorithm, as well as an excellent passage of integer programming in general, see [9]. A good summary of the BAP algorithm is given in [19], and as the principal difference between the BAB and the BAP algorithms is the introduction of column generation in the latter, we refer the reader to previously proposed literature on the subject of CG; [14, 15]. Finally, for an overview of branching rules[1]—a key parameter with regard to the efficiency of the BAB/BAP algorithms—see [21].

4.1 The Branch-and-Bound Algorithm

Consider the BLP to

$$\begin{aligned} \underset{\boldsymbol{x}}{\text{minimize}} \quad z_{\text{BLP}} = \quad & \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, && \text{(4.1a)} \\ \text{subject to} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, && \text{(4.1b)} \\ & \boldsymbol{x} \in \{0,1\}^n, && \text{(4.1c)} \end{aligned}$$

where $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, and $A \in \mathbb{R}^{m \times n}$, and let $\boldsymbol{x}^*_{\text{BLP}}$ and z^*_{BLP} denote the optimal solution and associated optimal objective function value, respectively, to (4.1).

Now, ignoring any infeasibility with regard to the constraints (4.1b), we have (at most) 2^n possible solutions to the problem, *candidate solutions*. One method to solve the problem would be to simply test each candidate solution for feasibility; calculating the objective function value for each such candidate; and finally identifying the optimum as the solution(s) corresponding to the lowest objective function value among all feasible candidate solutions. In practice, however, this is generally not possible, as the number of candidate solutions grow exponentially with the number n of decision variables in the program.

BAB is an algorithm for finding the optimal solution of (4.1) without explicitly investigating every candidate solution, instead discarding these en masse, using different criteria to test, identify, and cast away groups of ineffectual solutions. The general idea is to divide the original problem into subproblems by imposing a single constraint on one of the variables of the original problem, partitioning the feasible region of (4.1) into two[2] subregions whose union equals the original region, and then iteratively repeat the process for the subproblems generated. In the context of the

[1]Branching rules with regard to choosing variables for branching, a concept that will be introduced in the first section of this chapter.

[2]We present the BAB algorithm in the context of BLP, in which case each partitioning step results in two subregions. For general ILP, the analogous step can result in several subregions.

notion of groups of candidate solutions, presented above, the set of feasible solutions to such a subproblem describes a group of candidate solutions. The algorithm repeatedly makes use of the continuous relaxation of (4.1), in which the binary constraint (4.1c) has been continuously relaxed, i.e., replaced by the constraint $\boldsymbol{x} \in [0,1]^n$. By solving (or discovering the infeasibility of) the continuous relaxation of each subproblem, we can create the criteria needed to decide whether to throw away—*prune*—a subproblem, or to continue investigating it by partitioning—*branching*—its feasible region further. We return to these terms shortly.

Denote (4.1) by $\mathcal{P}^{\mathrm{BLP}}$ and let $\mathcal{P}_S^{\mathrm{LP}}$ denote its continuous relaxation, where $T := \{\boldsymbol{x} \in \{0,1\}^n \mid A\boldsymbol{x} \geq \boldsymbol{b}\}$ and $S := \{\boldsymbol{x} \in [0,1]^n \mid A\boldsymbol{x} \geq \boldsymbol{b}\}$ represents the feasible region of $\mathcal{P}^{\mathrm{BLP}}$ and $\mathcal{P}_S^{\mathrm{LP}}$, respectively. Assume that T is non-empty.

Now, let $\boldsymbol{x}^*_{\mathrm{LP(S)}}$ be the optimal solution of $\mathcal{P}_S^{\mathrm{LP}}$ and assume that $\boldsymbol{x}^*_{\mathrm{LP}(S)} \notin \{0,1\}^n$, i.e., the solution is fractional in the sense that the value of at least one of the decision variables $\tilde{x}_i := (x^*_{\mathrm{LP(S)}})_i$, $i = 1,\ldots,n$, is fractional, i.e., $\exists\, \tilde{x}_i \in (0,1)$. Without loss of generality, we choose one such fractional variable, say $\tilde{x}_1$, and refer to this as the *branching variable*. The partitioning of the feasible region of $\mathcal{P}_S^{\mathrm{LP}}$ into two sub-regions can then be performed by imposing a *down cut*, $\tilde{x}_1 \leq 0$, and an *up cut*, $\tilde{x}_1 \geq 1$ on the branching variable, according to

$$S_0 := \{\boldsymbol{x} \in [0,1]^n \mid A\boldsymbol{x} \geq \boldsymbol{b} \mid \tilde{x}_1 \leq 0\}, \tag{4.2a}$$

$$S_1 := \{\boldsymbol{x} \in [0,1]^n \mid A\boldsymbol{x} \geq \boldsymbol{b} \mid \tilde{x}_1 \geq 1\}, \tag{4.2b}$$

$$T_0 := \{\boldsymbol{x} \in \{0,1\}^n \mid A\boldsymbol{x} \geq \boldsymbol{b} \mid \tilde{x}_1 \leq 0\}, \tag{4.2c}$$

$$T_1 := \{\boldsymbol{x} \in \{0,1\}^n \mid A\boldsymbol{x} \geq \boldsymbol{b} \mid \tilde{x}_1 \geq 1\}. \tag{4.2d}$$

If replacing the feasible region S of $\mathcal{P}_S^{\mathrm{LP}}$ by each of these sub-regions, S_0 and S_1, we generate two LP subproblems; denote these by $\mathcal{P}_{S_0}^{\mathrm{LP}}$ and $\mathcal{P}_{S_1}^{\mathrm{LP}}$. These subproblems have the property that the union of the regions of the related[3] binary subproblems of $\mathcal{P}^{\mathrm{BLP}}$—denote these by $\mathcal{P}_{T_0}^{\mathrm{BLP}}$ and $\mathcal{P}_{T_1}^{\mathrm{BLP}}$—equals T, i.e., $T = T_0 \cup T_1$, and $T_0 \cap T_1 = \emptyset$. Naturally $\min_{\boldsymbol{x}\in T} \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} = \min_{i\in\{0,1\}} \left\{\min_{\boldsymbol{x}\in T_i} \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}\right\}$, i.e., the optimization of T can be split up into separate optimizations over the partitions of T.

The essence of this branching process is that the candidate solutions of (4.1), described by T, has been split up into two separate groups. We say that the *depth* d of each of these groups is 1, $d = 1$, as exactly one constraint has been added with regard to the original problem $\mathcal{P}^{\mathrm{BLP}}$ (and its continuous relaxation $\mathcal{P}_S^{\mathrm{LP}}$), where we note that the original problem naturally has depth 0, $d = 0$, and in the context of BAB we refer to it as our *root note*. The groups can be further divided using the same method iteratively, i.e., identifying fractional variables $\tilde{x}_j$, $j = 1,\ldots,n$, in the optimal solutions

[3]Related in the sense that the same single constraints have been added to the original BLP (4.1).

of the associated LP problems, and branching upon these variables by further adding up and down cuts. Each group which has not been further divided is called a *leaf* (or *active node*), and each additional branching of such a leaf increases the depth of the subsequently generated leaves. The leaves make up the BAB *tree*. Naturally the maximum depth of any leaf in the tree equals n—the number of decision variables—as each branching operation possibly[4] adds one up cut and one down cut only to variables which has not previously been branched upon in the active *branch* at hand, which runs from the root node to the active leaf. Note also that for a certain depth, say $\tilde{n} \geq 2$ (i.e., possibly more than two leaves are at this depth, and hence more than one predecessor leaf to these exists), the branching variables connecting each of the possibly numerous leaves at depth $\tilde{n}$ to its predecessor in the tree are not necessarily the same variables, as the choice of a branching variable is performed locally at each branched leaf. Details of this process will follow.

Now, return to the term *pruning*. In equation (4.2) above we considered the specific cases of up and down cuts from the root note to leaves of depth $d = 1$. We now consider any arbitrary leaf, say p, of depth $d \geq 1$ (i.e., not the root node) in an BAB tree derived from (4.1), with associated LP and BLP feasible regions $S(p)$ and $T(p)$, respectively, and associated subproblems $\mathcal{P}^{\text{LP}}_{S(p)}$ and $\mathcal{P}^{\text{BLP}}_{T(p)}$, respectively. The notation $S(p)$ (and $T(p)$) is introduced specifically to represent the region(s) of arbitrary leaves, and would, for a non-arbitrary leaf, translate to the notation used in (4.2) above, e.g. $S(\tilde{p}) = S_{0110}$ (and $T(\tilde{p}) = T_{0110}$) for a specific leaf $\tilde{p}$ of depth $d = 4$, in some specific BAB execution. Note that $S(p) \subseteq S$ and $T(p) \subseteq T$, and naturally, $T(p) \subseteq S(p)$. Furthermore, let $\boldsymbol{x}^*_{\text{LP}(S(p))}$ and $\boldsymbol{x}^*_{\text{BLP}(T(p))}$ be the optimal solutions of $\mathcal{P}^{\text{LP}}_{S(p)}$ and $\mathcal{P}^{\text{BLP}}_{T(p)}$, respectively. Naturally, if $\boldsymbol{x}^*_{\text{LP}(S(p))} \in \{0,1\}^n$, $\boldsymbol{x}^*_{\text{LP}(S(p))}$ is feasible and hence optimal also in $\mathcal{P}^{\text{BLP}}_{T(p)}$, with $\boldsymbol{x}^*_{\text{LP}(S(p))} = \boldsymbol{x}^*_{\text{BLP}(T(p))}$.

Now, let $z_{\text{BLP}(T(p))}$ be the optimal objective function value of $\mathcal{P}^{\text{BLP}}_{T(p)}$, and let $\bar{z}_{\text{BLP}}$ be, for any given iteration in the BAB tree, the currently best upper bound on the optimal objective function value of (4.1), implying that $z^*_{\text{BLP}} \leq \bar{z}_{\text{BLP}}$ holds. We can state the following facts:

- $\boldsymbol{c}^{\text{T}}\boldsymbol{x}^*_{\text{LP}(S(p))}$ is a *lower bound* on the optimal objective function value of $\mathcal{P}^{\text{BLP}}_{T(p)}$.
- As mentioned above, if $\boldsymbol{x}^*_{\text{LP}(S(p))} \in \{0,1\}^n$, then $\boldsymbol{c}^{\text{T}}\boldsymbol{x}^*_{\text{LP}(S(p))} = \boldsymbol{c}^{\text{T}}\boldsymbol{x}^*_{\text{BLP}(T(p))} = z_{\text{BLP}(T(p))}$, i.e., $\boldsymbol{x}^*_{\text{LP}(S(p))}$ solves $\mathcal{P}^{\text{BLP}}_{T(p)}$. This naturally means that $z_{\text{BLP}(T(p))}$ is an *upper bound* on the optimal objective function value of (4.1), $\bar{z}_{\text{BLP}}$. For each

[4]As mentioned previously, an initial criteria for branching of any leaf is that the optimal solution of the associated LP problem must contain at least one variable with fractional value, hence representing a non-feasible solution in the related BLP subproblem.

such case, the best upper bound—as computed by heuristics or by prior leaf evaluations—is updated according to $\bar{z}_{\text{BLP}} := \min\{\bar{z}_{\text{BLP}}, z_{\text{BLP}(T(p))}\}$.

- If $\mathcal{P}^{\text{LP}}_{S(p)}$ if infeasible, any further branching of the leaf p is pointless, as it follows that any generated subproblems is also infeasible.

These statements can be used to describe the three different ways to perform pruning during the BAB process:

1. Naturally, if for any leaf p $\mathcal{P}^{\text{LP}}_{S(p)}$ is infeasible, then so is $\mathcal{P}^{\text{BLP}}_{T(p)}$. In such case, any further partitioning of $T(p)$ is pointless, and hence *prune the leaf by infeasibility*.

2. If any leaf, say p, is not pruned due to infeasibility, but have an LP optimal objective function value that is greater than the current[5] upper bound on the original BLP (4.1), then the optimal solution of $\mathcal{P}^{\text{BLP}}_{T(p)}$ cannot possible yield an objective function value below the current upper bound. Consequently, if $\boldsymbol{c}^{\text{T}}\boldsymbol{x}^{*}_{\text{LP}(S(p))} \geq \bar{z}_{\text{BLP}}$, *prune the leaf by bound*.

3. If any leaf, say p, is not pruned by bound, and the LP optimal solution is binary valued, $\boldsymbol{x}^{*}_{\text{LP}(S(p))} \in \{0,1\}^n$, then the associated objective function value, $\boldsymbol{c}^{\text{T}}\boldsymbol{x}^{*}_{\text{LP}(S(p))} = \boldsymbol{c}^{\text{T}}\boldsymbol{x}^{*}_{\text{BLP}(T(p))}$, will—by the definition of the criteria for pruning by bound, i.e. 2.—improve the best upper bound, i.e., $\min\{\bar{z}_{\text{BLP}}, \boldsymbol{c}^{\text{T}}\boldsymbol{x}^{*}_{\text{LP}(S(p))}\} = \boldsymbol{c}^{\text{T}}\boldsymbol{x}^{*}_{\text{LP}(S(p))}$. Consequently, update the best upper bound and *prune the leaf by optimality*.

The LP-based BAB described in this section—specifically for BLPs—is presented in pseudocode in Algorithm 1, where the BLP is assumed to be a minimization problem, and where C is used to denote the *subproblem pool*, containing all active leaves or subproblems. Note that the *prune*(p) functions hold no actual meaning, and are added simply for ease of presentation[6].

We move on to present a simple yet illustrating example of the algorithm in practice.

[5] *Current* meaning the upper bound at the time of processing the leaf p. Naturally this upper bound can be updated several times during the BAB process, as leaves are visited that have binary valued LP optimal solutions.

[6] Subproblems (leaves) are removed from the subproblem pool before they are processed, and are hence pruned implicitly if they are not branched; the denotation of pruning is simply a statement to return to the start of the "while" statement on row 1 in Algorithm 1.

Algorithm 1 LP-based Branch and Bound for Binary Linear Programs

1: **procedure** LP-BAB($\mathcal{P}^{\text{BLP}}$) ▷ **INPUT**: BLP $\mathcal{P}^{\text{BLP}}$
2: $\mathcal{P}_S^{\text{LP}} \leftarrow relax(\mathcal{P}^{\text{BLP}})$ ▷ Continuously relax $\mathcal{P}^{\text{BLP}}$
3: $\overline{z}_{\text{BLP}} \leftarrow +\infty$ ▷ Set initial upper bound on input BLP
4: $\overline{\boldsymbol{x}}_{\text{BLP}} \leftarrow \emptyset$ ▷ Initialize best feasible solution: empty
5: $initialize(C)$ ▷ Initialize the subproblem pool to be empty
6: $add(C, \mathcal{P}_S^{\text{LP}})$ ▷ Add root node to subproblem pool
7: **while** $C \neq \emptyset$ **do** ▷ Continue as long as pool is non-empty
8: $p \leftarrow choose(C, \mathcal{P}_i^{\text{LP}})$ ▷ Choose a problem p from the pool
9: $remove(C, p)$ ▷ Remove this problem from the pool
10: **if** $(z_p^*, \boldsymbol{x}_p^*) \leftarrow solve(p)$ **then** ▷ Solve p, continue if feasible
11: **if** $z_p^* \geq \overline{z}_{\text{BLP}}$ **then**
12: $prune(p)$ ▷ p pruned by bound
13: **else**
14: **if** $\boldsymbol{x}_p^* \in \{0,1\}^n$ **then**
15: $\overline{z}_{\text{BLP}} \leftarrow z_p^*$ ▷ Update upper bound
16: $\overline{\boldsymbol{x}}_{\text{BLP}} \leftarrow \boldsymbol{x}_p^*$ ▷ Update best incumbent BLP solution
17: $prune(p)$ ▷ p pruned by optimality
18: **else**
19: $(p_1, p_2) \leftarrow branch(p)$ ▷ Branch p into two subproblems
20: $C \leftarrow (p_1, p_2)$ ▷ Add subproblems to pool
21: **end if**
22: **end if**
23: **else**
24: $prune(p)$ ▷ p pruned by infeasibility
25: **end if**
26: **end while**
27: **if** $\overline{x}_{\text{BLP}} \neq \emptyset$ **then**
28: **return** $(\overline{z}_{\text{BLP}}, \overline{\boldsymbol{x}}_{\text{BLP}})$ ▷ **OUTPUT**: optimal solution to $\mathcal{P}^{\text{BLP}}$
29: **else**
30: **return** $NULL$ ▷ **OUTPUT**: $\mathcal{P}^{\text{BLP}}$ is infeasible
31: **end if**
32: **end procedure**

4.1.1 An illustrating example

Consider the following instance of the BLP (4.1), where $\boldsymbol{x} \in \{0,1\}^3$, to

$$
\begin{aligned}
\underset{\boldsymbol{x}=(x_1,x_2,x_3)}{\text{minimize}} \quad z_{\text{BLP}} = \; & x_1 + \tfrac{2}{3}x_2 + \tfrac{5}{6}x_3, && \text{(4.3a)}\\
\text{subject to} \quad & x_1 - \tfrac{1}{2}x_2 \geq 0, && \text{(4.3b)}\\
& \tfrac{1}{3}x_1 + x_2 - \tfrac{1}{2}x_3 \geq 0, && \text{(4.3c)}\\
& x_1 + x_2 + x_3 \geq 1, && \text{(4.3d)}\\
& x_1, \; x_2, \; x_3 \in \{0,1\}. && \text{(4.3e)}
\end{aligned}
$$

Denote (4.3) $\mathcal{P}_T^{\text{BLP}}$, with feasible region T, described by the constraints (4.3b–4.3e). The optimal solution of $\mathcal{P}_T^{\text{BLP}}$ is $(x_1, x_2, x_3) = (1, 0, 0)$, with optimal objective function value $z^*_{\text{BLP}} = 1$. Example 1 below shows, step by step, how this solution can be reached using the BAB procedure presented previously in this subsection. If one so prefer, the pseudocode in Algorithm 1 is a good companion while covering the example.

Example 1 (An illustrating BAB example)

`init:` Continuously relax the binary constraints (4.3e) of $\mathcal{P}_T^{\text{BLP}}$ to $\boldsymbol{x} \in [0,1]^3$, and denote the resulting LP as $\mathcal{P}_S^{\text{LP}}$, with feasible region S defined by the constraints (4.3b–4.3d) and $\boldsymbol{x} \in [0,1]^3$. Initialize the BAB subproblem pool C by $\mathcal{P}_S^{\text{LP}}$. Set the upper bound to $+\infty$ and the best incumbent solution to $\mathcal{P}_T^{\text{BLP}}$ as empty:

$$
\begin{cases}
C & = \{\mathcal{P}_S^{\text{LP}}\},\\
\bar{z}_{\text{BLP}} & = +\infty,\\
\bar{\boldsymbol{x}}_{\text{BLP}} & = \emptyset.
\end{cases}
$$

The current setup is visualised in Figure 4.1(a).

`it01:` C is non-empty: choose a subproblem, $\mathcal{P}_S^{\text{LP}}$, from C.

`a:` Remove $\mathcal{P}_S^{\text{LP}}$ from C.

`b:` Solve $\mathcal{P}_S^{\text{LP}}$. Feasible? Yes: proceed (no pruning by infeasibility).

$$
\begin{aligned}
z^*_{\text{LP}(S)} &= 7/9,\\
\boldsymbol{x}^*_{\text{LP}(S)} &= (1/3, 2/3, 0).
\end{aligned}
$$

`c:` Is $z^*_{\text{LP}(S)} \geq \bar{z}_{\text{BLP}}$? No: proceed (no pruning by bound).

`d:` Is $\boldsymbol{x}^*_{\text{LP}(S)}$ binary-valued? No: proceed (no pruning by optimality).

`e:` Branch $\mathcal{P}_S^{\text{LP}}$: *choose*[a] a fractional variable in $\boldsymbol{x}^*_{\text{LP}(S)}$ to branch upon. E.g., choose $x_2 = (x^*_{\text{LP}(S)})_i\big|_{i=2} = 2/3$ as branching variable $\tilde{x}_1$, i.e., $\tilde{x}_1 = x_2$. Branching on $\tilde{x}_1$

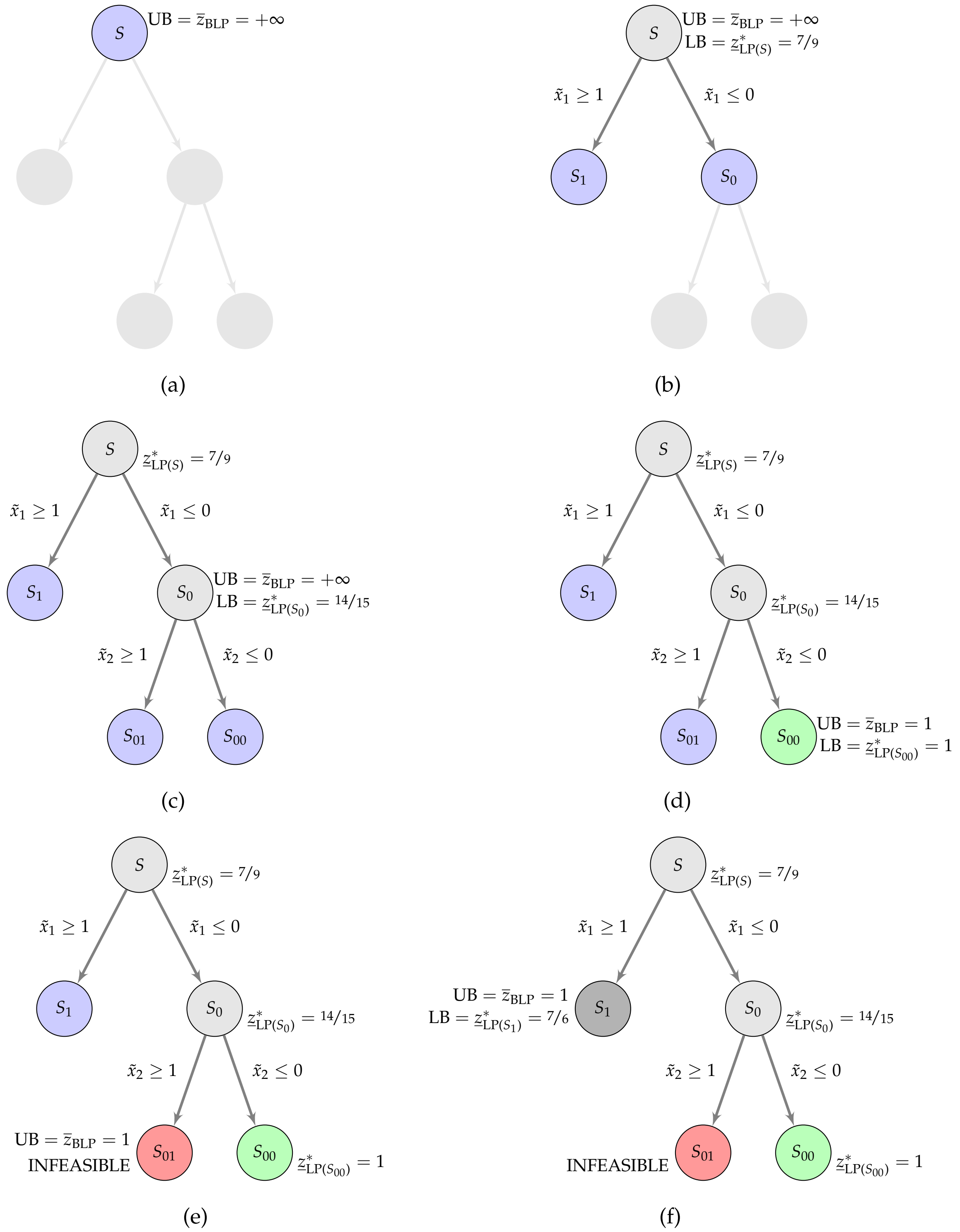


FIGURE 4.1: A BAB example; with **(a)** initialization, introducing the *root node* with associated LP, $\mathcal{P}^{\text{LP}}_S$, **(b)** solving $\mathcal{P}^{\text{LP}}_S$ and *branching* on a *branching variable* $\tilde{x}_1$, **(c)** solving $\mathcal{P}^{\text{LP}}_{S_0}$ and branching on $\tilde{x}_2$, **(d)** solving $\mathcal{P}^{\text{LP}}_{S_{00}}$ and *pruning by optimality*, as $\boldsymbol{x}^*_{\text{LP}(S_{00})} \in \{0,1\}^n$, **(e)** solving $\mathcal{P}^{\text{LP}}_{S_{01}}$ and *pruning by infeasibility*, **(f)** solving $\mathcal{P}^{\text{LP}}_{S_1}$ and *pruning by bound*, as $z^*_{\text{LP}(S_1)} \geq \bar{z}_{\text{BLP}}$.

yields the two following subproblems

$$\begin{aligned}\mathcal{P}^{\text{LP}}_{S_0}: \quad & S_0 := \{S \mid \tilde{x}_1 \leq 0\} = \{S \mid x_2 \leq 0\},\\ \mathcal{P}^{\text{LP}}_{S_1}: \quad & S_1 := \{S \mid \tilde{x}_1 \geq 1\} = \{S \mid x_2 \geq 1\}.\end{aligned}$$

`f:` Add $\mathcal{P}^{\text{LP}}_{S_0}$ and $\mathcal{P}^{\text{LP}}_{S_1}$ to C.

The BAB stats after this iteration are

$$\begin{cases} C & = \{\mathcal{P}^{\text{LP}}_{S_0}, \mathcal{P}^{\text{LP}}_{S_1}\},\\ \overline{z}_{\text{BLP}} & = +\infty,\\ \overline{\boldsymbol{x}}_{\text{BLP}} & = \emptyset.\end{cases}$$

The current setup is visualised in Figure 4.1(b).

`it02:` C is non-empty: choose[b] a subproblem, $\mathcal{P}^{\text{LP}}_{S_0}$, from C.

`a:` Remove $\mathcal{P}^{\text{LP}}_{S_0}$ from C.

`b:` Solve $\mathcal{P}^{\text{LP}}_{S_0}$. Feasible? Yes: proceed (no pruning by infeasibility).

$$\begin{aligned} z^*_{\text{LP}(S_0)} &= {}^{14}/_{15},\\ \boldsymbol{x}^*_{\text{LP}(S_0)} &= ({}^{3}/_{5}, 0, {}^{2}/_{5}).\end{aligned}$$

`c:` Is $z^*_{\text{LP}(S_0)} \geq \overline{z}_{\text{BLP}}$? No: proceed (no pruning by bound).

`d:` Is $\boldsymbol{x}^*_{\text{LP}(S_0)}$ binary-valued? No: proceed (no pruning by optimality).

`e:` Branch $\mathcal{P}^{\text{LP}}_{S_0}$: choose a fractional variable in $\boldsymbol{x}^*_{\text{LP}(S_0)}$ to branch upon. E.g., choose $x_3 = (x^*_{\text{LP}(S_0)})_i\big|_{i=3} = {}^{2}/_{5}$) as branching variable $\tilde{x}_2$, i.e., $\tilde{x}_2 = x_3$. Branching on $\tilde{x}_2$ yields the two following subproblems

$$\begin{aligned}\mathcal{P}^{\text{LP}}_{S_{00}}: \quad & S_{00} := \{S_0 \mid \tilde{x}_2 \leq 0\} = \{S \mid \tilde{x}_1 \leq 0 \mid \tilde{x}_2 \leq 0\} = \{S \mid x_2 \leq 0 \mid x_3 \leq 0\},\\ \mathcal{P}^{\text{LP}}_{S_{01}}: \quad & S_{01} := \{S_0 \mid \tilde{x}_2 \geq 1\} = \{S \mid \tilde{x}_1 \leq 0 \mid \tilde{x}_2 \geq 1\} = \{S \mid x_2 \leq 0 \mid x_3 \geq 1\}.\end{aligned}$$

`f:` Add $\mathcal{P}^{\text{LP}}_{S_{00}}$ and $\mathcal{P}^{\text{LP}}_{S_{01}}$ to C.

The BAB stats after this iteration are

$$\begin{cases} C & = \{\mathcal{P}^{\text{LP}}_{S_1}, \mathcal{P}^{\text{LP}}_{S_{00}}, \mathcal{P}^{\text{LP}}_{S_{01}}\},\\ \overline{z}_{\text{BLP}} & = +\infty,\\ \overline{\boldsymbol{x}}_{\text{BLP}} & = \emptyset.\end{cases}$$

The current setup is visualised in Figure 4.1(c).

`it03:` C is non-empty: choose a subproblem $\mathcal{P}^{\text{LP}}_{S_{00}}$.

`a`: Remove $\mathcal{P}^{\text{LP}}_{S_{00}}$ from C.

`b`: Solve $\mathcal{P}^{\text{LP}}_{S_{00}}$. Feasible? Yes: proceed (no pruning by infeasibility).

$$z^*_{\text{LP}(S_{00})} = 1,$$
$$\boldsymbol{x}^*_{\text{LP}(S_{00})} = (1, 0, 0).$$

`c`: Is $z^*_{\text{LP}(S_{00})} \geq \bar{z}_{\text{BLP}}$? No: proceed (no pruning by bound).

`d`: Is $\boldsymbol{x}^*_{\text{LP}(S_{00})}$ binary-valued? Yes: *prune by optimality*.

`d'`: Update upper bound and best incumbent solution.

$$\bar{z}_{\text{BLP}} = z^*_{\text{LP}(S_{00})} = 1,$$
$$\bar{\boldsymbol{x}}_{\text{BLP}} = \boldsymbol{x}^*_{\text{LP}(S_{00})} = (1, 0, 0).$$

The BAB stats after this iteration are

$$\begin{cases} C & = \{\mathcal{P}^{\text{LP}}_{S_1}, \mathcal{P}^{\text{LP}}_{S_{01}}\}, \\ \bar{z}_{\text{BLP}} & = 1, \\ \bar{\boldsymbol{x}}_{\text{BLP}} & = (1, 0, 0). \end{cases}$$

The current setup is visualised in Figure 4.1(d).

`it04`: C is non-empty: choose a subproblem $\mathcal{P}^{\text{LP}}_{S_{01}}$.

`a`: Remove $\mathcal{P}^{\text{LP}}_{S_{01}}$ from C.

`b`: Solve $\mathcal{P}^{\text{LP}}_{S_{01}}$. Feasible? No: *prune by infeasibility* (note that for fixed $\tilde{x}_1 = x_2 = 0$ and $\tilde{x}_2 = x_3 = 1$, the constraint (4.3c) can never be fulfilled, as $x_1 \in [0, 1]$ in $\mathcal{P}^{\text{LP}}_{S_{01}}$).

The BAB stats after this iteration are

$$\begin{cases} C & = \{\mathcal{P}^{\text{LP}}_{S_1}\}, \\ \bar{z}_{\text{BLP}} & = 1, \\ \bar{\boldsymbol{x}}_{\text{BLP}} & = (1, 0, 0). \end{cases}$$

The current setup is visualised in Figure 4.1(e).

`it05`: C is non-empty, but contain only one subproblem, $\mathcal{P}^{\text{LP}}_{S_1}$.

`a`: Remove $\mathcal{P}^{\text{LP}}_{S_1}$ from C.

`b`: Solve $\mathcal{P}^{\text{LP}}_{S_1}$. Feasible? Yes: proceed (no pruning by infeasibility).

$$z^*_{\text{LP}(S_1)} = 7/6,$$
$$\boldsymbol{x}^*_{\text{LP}(S_1)} = (1/2, 1, 0).$$

C: Is $z^*_{\mathrm{LP}(S_1)} \geq \bar{z}_{\mathrm{BLP}}$? Yes: *prune by bound*.

The BAB stats after this iteration are

$$\begin{cases} C &= \emptyset, \\ \bar{z}_{\mathrm{BLP}} &= 1, \\ \bar{\boldsymbol{x}}_{\mathrm{BLP}} &= (1,0,0). \end{cases}$$

As $C = \emptyset$, the BAB algorithm is terminated. Moreover, as $\bar{\boldsymbol{x}}_{\mathrm{BLP}} \neq \emptyset$, the best incumbent solution $\bar{\boldsymbol{x}}_{\mathrm{BLP}}$ solves $\mathcal{P}^{\mathrm{BLP}}_T$, i.e., (4.3), with optimal objective function value $\bar{z}_{\mathrm{BLP}}$.

$$\mathrm{BAB}(\mathcal{P}^{\mathrm{BLP}}_T) \Rightarrow \begin{cases} z^*_{\mathrm{BLP}} &= \bar{z}_{\mathrm{BLP}} = 1, \\ \boldsymbol{x}^*_{\mathrm{BLP}} &= \bar{\boldsymbol{x}}_{\mathrm{BLP}} = (1,0,0). \end{cases} \tag{4.4}$$

The final setup is visualised in Figure 4.1(f), which now describes a complete instance of a BAB *tree* for (4.3).

[a] As will be discussed in detail in Chapter 6, and further studied in Chapter 7, the choice of branching variable can have a great effect on the overall performance of the BAB algorithm. For the scope of this example, however, consider the choice of branching variable—among fractional variables—as random.

[b] In the second iteration, the choice of subproblem from C is no longer unambiguous. In this specific example, we've chosen to process the subproblem that relates to the down cut to the variable branched in it01. As will be studied in later chapters, the way to choose subproblems from the subproblem pool can, much like the choice of branching variable, play an important role in the improvement of the general BAB algorithm. This notion of a so-called subproblem or *node/leaf picking rule* is discussed in Chapters 6 and 7. For now, to cover the scope of this example, however, consider the simple random subproblem picking as our method of choice.

A note on complexity

We recall that, ignoring feasibility w.r.t. the constraints (4.1b), there exists 2^n candidate solutions to the general BLP (4.1). Hence, even though BAB eventually converges—assuming that the computer resources at hand are sufficient—to an optimal solution, in the worst-case scenario[7] it does so in exponential time.

Now, the BAB performance can be enhanced by using smart branching rules and

[7] Theoretically, the branching process *could* lead to the full enumeration of all the candidate solutions of the original BLP (4.1); if all the LP optimal solutions for all leafs for which $d < n$ are fractional—i.e., no pruning by optimality—and where no leaves for $d < n$ are pruned due to bound or infeasibility. This scenario is depicted graphically in Figure 4.2, for $n = 3$, i.e., the full enumeration of $2^n = 8$ candidate solutions. In practice, however, clever in-algorithm rules ascertain that at least a great part of leaves that—when expanded—contain only unfruitful candidate solutions can be pruned at a shallow depth in the BAB tree, in so discarding the associated solutions en masse.

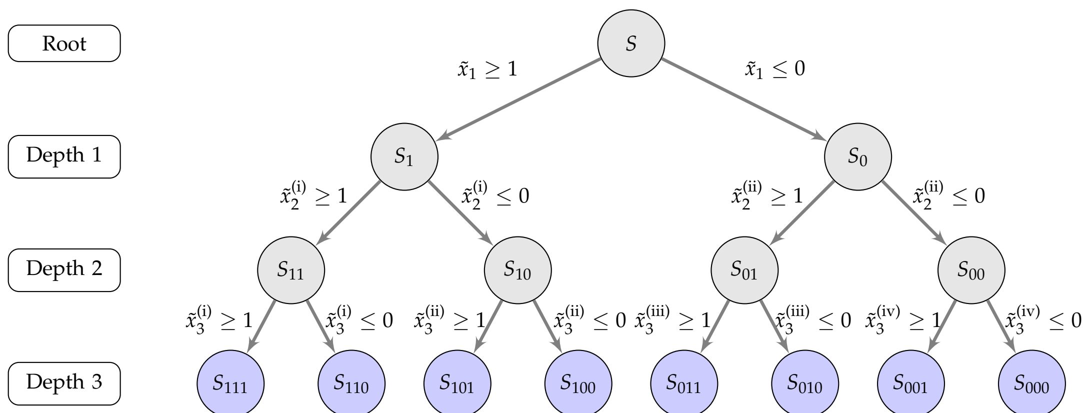


FIGURE 4.2: BAB, full enumeration of subsets, for $n = 3$ in (4.1), i.e., $\mathcal{P}^{\text{BLP}}$ with $\boldsymbol{x} \in \{0,1\}^3$. Note that the branching variables for leaves at same depth are not necessarily the same, e.g. $\tilde{x}_2^{(i)}$ and $\tilde{x}_2^{(ii)}$ may be different variables.

likewise good methods to pick nodes in the BAB tree, resulting in a BAB algorithm that run reasonably fast on average. If n is very large, however, then the continuous relaxation of (4.1) might in itself be a challenge to solve, which is an apparent issue, as the BAB algorithm iteratively solves a closely related LP. In such cases, we would like, if possible, to simplify the LP *before* we implement it in a BAB framework. In order to accomplish this, we move on to BAB in the context of DW decomposition and column generation.

4.2 BAB in the Context of Dantzig-Wolfe Decomposition and Column Generation

We continue to focus of the special case of ILP that is BLP. In the end of the previous section, we noted that a limiting factor for the practical usefulness of the BAB algorithm is the size of the BLP, as a very large BLP yields a likewise large LP relaxation. If the latter program is intractable due to its sheer size, naturally the BAB approach will be intractable as well. With the theory covered in Chapter 3 fresh in mind, we have, however, tools that can be combined with the BAB algorithm to reach a tractable solution process, even for a seemingly intractable ILP.

Now, recall the Dantzig-Wolfe decomposition principle covered in the previous chapter, specifically the discretization procedure in Section 3.3. For all notation—coefficient vectors and matrices, variables, sets, and so on—not explicitly defined in this section, we refer to Section 3.3. We consider the ILP (3.17) and replace the integrality requirement $\boldsymbol{x} \in \mathbb{Z}_+^n$ with the binearity requirement $\boldsymbol{x} \in \{0,1\}^n$, to attain

the BLP to

$$
\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad z_{\text{BLP}} = \quad & \boldsymbol{c}^T \boldsymbol{x}, && \text{(4.5a)}\\
\text{subject to} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, && \text{(4.5b)}\\
& D\boldsymbol{x} \geq \boldsymbol{d}, && \text{(4.5c)}\\
& \boldsymbol{x} \in \{0,1\}^n, && \text{(4.5d)}
\end{aligned}
$$

Analogously to Section 3.3, apply the result of Theorem 4 to (4.5), to attain the following binary MP to

$$
\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{minimize}} \quad z_{\text{MP}} = \quad & \sum_{q\in Q} \boldsymbol{c}^T \boldsymbol{p}_q \lambda_q, && \text{(4.6a)}\\
\text{subject to} \quad & \sum_{q\in Q} A\boldsymbol{p}_q \lambda_q \geq \boldsymbol{b}, && \text{(4.6b)}\\
& \sum_{q\in Q} \lambda_q = 1, && \text{(4.6c)}\\
& \lambda_q \in \{0,1\}, \quad q \in Q. && \text{(4.6d)}
\end{aligned}
$$

Now, as is described in Section 3.3, if X—as defined in Theorem 4—is bounded, a trivial result of Theorem 4 is that any $\boldsymbol{x} \in X$ can be represented as one of the integer points in X; i.e., the set of integer points—or set of columns—$\{\boldsymbol{p}_q\}_{q\in Q}$ in (4.6) is the set of X itself. With this in mind, the problem (4.6) boils down to finding a column $\boldsymbol{p}_q$, $q \in Q$, with lowest associated objective function value in (4.6a), and which also fulfils the constraints (4.6b)[8].

With the purpose of applying the BAB algorithm to solve (4.6), it is quite apparent that there is no merit in trying to branch direcly upon the MP variables λ_q, $q \in Q$, as each step in the algorithm would reduce to testing, one at a time, a single candidate solution for objective function value and feasibility. Due to the convexity constraint (4.6c), a leaf with an up cut on, say, variable $\lambda_{\tilde{q}}$, for some $\tilde{q} \in Q$ ($\lambda_{\tilde{q}} \geq 1$, leaf S_{*1} in Figure 4.3(a)), would never be further expanded; the added up cut results in $\lambda_{\tilde{q}}$ fixed to a value of 1, $\lambda_{\tilde{q}} = 1$, in which case the constraint (4.6c) combined with the non-negativity of the λ_q variables will constrain all other decision variables to the value of 0. Hence, in such a leaf, only a single set of decision variables are feasible w.r.t. the constraints (4.6c), namely $\widetilde{\boldsymbol{\lambda}} = (0,\ldots,\lambda_{\tilde{q}},\ldots,0)^{\text{T}}$, and this single possible candidate solution will be pruned either by infeasibility w.r.t. the constraints (4.6b), or, as $\widetilde{\boldsymbol{\lambda}}^{\text{T}} \in \{0,1\}^{|Q|}$, pruned by integrality. This means that the BAB tree would grow a single, deep branch of down cuts—growing to the right in Figure 4.3(a)—where for

[8] In this simple interpretation of the problem, the constraints (4.6c) and (4.6d) are already implicitly fulfilled since we endevour to find "a (single) column".

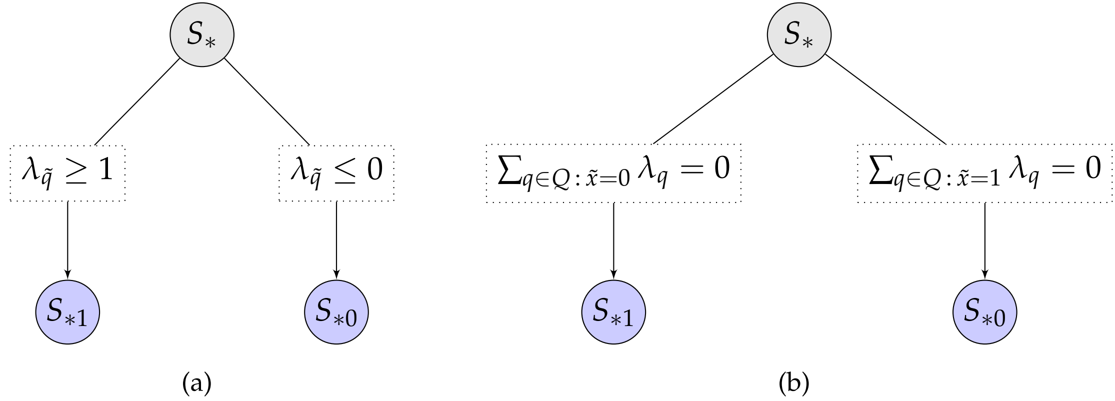


FIGURE 4.3: BAB, branching on (a) the MP variables, λ_q, $q \in Q$, and (b) the original variables $x_i \in \boldsymbol{x} \in \mathbb{R}^n$.

each depth, a single candidate solution is tested in the up cut branch. In practice, it's possible that after sufficiently many candidate solutions have been excluded, i.e., sufficiently many subsequent down cuts have been applied to the LP relaxation of (4.6), the LP optimal solution would be binary valued, and the ever-growing down cut branch could be pruned due to integrality. The theoretical maximum depth of the single down cut branch, however, amass to 2^n, in comparison with the maximum depth n for a BAB algorithm applied to the original problem (4.5). This means that one of the main strengths of BAB—discarding candidate solutions en masse—will be unutilized.

A more sensible approach is to look back at the original variables $\boldsymbol{x} \in \mathbb{R}^n$ of (4.5), and expand the BAB tree of (4.6) by branching on these, or specifically upon their representation by the MP variables λ_q, $q \in Q$. Consider, at any point in the BAB process of solving (4.6), the optimal solution of the LP relaxation of (4.6) with appended cuts, say $\boldsymbol{\lambda}^* = (\lambda_1^*, ..., \lambda_{|Q|}^*)^{\mathrm{T}} \in [0,1]^{|Q|}$. The corresponding solution[9] in the LP relaxation of the original problem (4.6) (with same set of appended cuts), say $\overline{\boldsymbol{x}}$, is constructed by $\overline{\boldsymbol{x}} = (\overline{x}_1, ..., \overline{x}_n) = \sum_{q \in Q} \boldsymbol{p}_q \lambda_q^*$, and given that $\boldsymbol{\lambda}^* \notin \{0,1\}^{|Q|}$, due to the convexity constraint (4.6c) there must exist some λ_i^*, $i = 1, .., |Q|$, with fractional values, which means that there must likewise exist some $\overline{x}_i$, $i = 1, .., n$, with fractional values. Instead of branching on a fractional MP variable, we can choose a corresponding fractional original variable, say $\tilde{x}$, and branch on this original variable by removing all columns $\boldsymbol{p}_q$, $q \in Q$, where the given variable has the value 0 or 1, depending on whether we want to branch up or down. Mathematically, we do this by imposing the constraint $\sum_{q \in Q\,:\,\tilde{x}=0} \lambda_q = 0$ for an up cut, or $\sum_{q \in Q\,:\,\tilde{x}=1} \lambda_q = 0$ for a

[9]Note that $\overline{x}$ is not necessarily optimal in its associated LP problem, as the MP generally has a tigher LP relaxation than the original problem, as is described in Section 3.3.

down cut; see Figure 4.3(b). With this method, we can apply the BAB algorithm to the MP (4.6) while maintaining an important limiting property of the original problem (4.5) w.r.t. BAB; we still branch (implicitly) on the original variables $x \in \mathbb{R}^n$, giving a maximum tree depth of n, as compared to the maximum tree depth of 2^n when branching directly on the MP variables λ_q, $q \in Q$. An additional advantage of applying BAB to the MP (4.6), rather than the original problem (4.5), is—as previously covered in Section 3.3—that the former generally have a tighter continuous relaxation than the latter, i.e., a smaller integrality gap between the optimal objective function value of (4.6) and its continuous relaxation. Refer to Section 3.3 for details.

A smaller integrality gap value between a BLP and its LP relexation is naturally as good reason as any as to why one should modify a problem prior to applying the BAB algorithm to it, as a smaller integrality gap will directly affect of how quickly branches of non-optimal candidate solutions can be discarded by pruning by bound. This is not the main reason, however, in the context of BAB, for us to apply the Dantzig-Wolfe decomposition principle on problem (4.5). Indeed, the BAB algorithm seems just as hopeless applied to the MP as to the original problem, given that the original problem is uncomfortably large. Our possible scepticism to the combined Dantzig-Wolfe BAB procedure will, however, shortly be cleared out, as we proceed to reduce the MP (4.6) to the RMP counterpart, in so joining BAB with column generation, reaching one of the key concepts used of this thesis.

4.3 Dynamic Update of the Column Pool: Branch-and-Price

We return to the BLP (4.5), and assume that it is of a *considerable size*. In our context, this term has the purpose of describing a problem for which the problem's continous relaxation, an LP problem, is a challenge itself to solve directly, given the available computer resources at hand. However, with the knowledge obtained through the past chapters, we have the tools to readily attack such a problem.

With the preparations above, it is not a far-fetched idea to consider the use of column generation and an RMP to iteratively solve large difficult LP:s in a BAB context. Consequently, we considered the RMP counterpart of (4.5), or specifically, the RMP

counterpart of the continous relaxation of the MP (4.5), i.e., the problem to

$$\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & z_{\text{RMP}} = \sum_{q\in Q'} \boldsymbol{c}^T \boldsymbol{p}_q \lambda_q, && \text{(4.7a)}\\
\text{subject to} \quad & \sum_{q\in Q'} A\boldsymbol{p}_q \lambda_q \geq \boldsymbol{b}, && \text{(4.7b)}\\
& \sum_{q\in Q'} \lambda_q = 1, && \text{(4.7c)}\\
& \lambda_q \in [0,1], \quad q \in Q'. && \text{(4.7d)}
\end{aligned}$$

Where, as before, $Q' \subset Q$. As described in the final section of Chapter 3, (4.7) is solved for an initial set of columns Q'—it is not a difficult task to heuristically construct a feasible set of columns to initialize Q'—and thereafter solve the corresponding BLP pricing subproblem(s) (see (3.20)) to generate improving columns to the RMP column pool Q'. When no more improving columns can be found, (4.7) is solved and we proceed within the BAB algorithm; by branching on a fractional original variable as described above and depicted in Figure 4.3(b). This is repeated iteratively as is described in detail for BAB in the previous section.

This hybrid between BAB and column generation is called *Branch-and-Price* (BAP), where the pricing part self-evidently refers to the iteratively solved pricing subproblem in the column generation process. BAP naturally inherits all the treats of the BAB algorithm, but with the additional strength that large LPs—the optimal solution of which generally contains a vast amount non-basic columns—can be solved with a relative ease, as is described in detail in Chapter 3.

We finalize the chapter by a schematic overview of the BAP algorithm—for a general BLP problem such as (4.5) (ILP analogously)—in the form of a flowchart. Let $\mathcal{P}^{\text{BLP}}$ describe the minimization BLP to be solved, and let $\mathcal{P}^{\text{LP}}_{\text{MP}}$ describe the continous relaxation of the MP form of $\mathcal{P}^{\text{BLP}}$, with the associated RMP[10] $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, and the RMP column pool Q'. Furthermore, let $\bar{z}_{\text{BLP}}$ describe—at any iteration in the BAP process—the current upper bound on the optimal objective function value of $\mathcal{P}^{\text{BLP}}$, with corresponding best incumbent binary valued solution $\bar{\boldsymbol{x}}^{\text{BLP}}$ of the original variables[11] of $\mathcal{P}^{\text{BLP}}$. Denote the BAP subproblem pool by C, and let $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ be the original root problem added to C at initialization. Assuming that $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ is branched upon, a number of variations of $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ (appended cuts) will be added to C, and many of these will be explicitly processed during the BAP process[12]. Let $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ denote the active

[10]For simplicity, we assume that there is only one pricing subproblem to $\mathcal{P}^{\text{LP}}_{\text{RMP}}$. For several subproblems the process is analogous.

[11]The values of the original variables are readily obtained from the RMP variables.

[12]It is naturally possible that an optimal solution is found and verified as optimal while there are still subproblems remaining in C, in which case the remaining problems will not need to be processed.

problem chosen from C at any time during the BAP process, say at the i:th BAB iteration. Given that $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ is feasible (otherwise pruned) let $z^{\text{LP}}_{\text{RMP}(i)(j)}$ and $\boldsymbol{x}^{\text{LP}}_{\text{RMP}(i)(j)}$ denote the corresponding RMP optimal solution and the corresponding original variable solution, respectively, after the j:th column generation iteration in the process of solving $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$. Finally let $\bar{c}^{*}_{(i)(j)}$ be the optimal value (cost value) of the pricing subproblem of $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ during the j:th CG iteration of the i:th BAB iteration.

Figure 4.4 depicts the flowchart that summarizes the BAP algorithm as presented in this chapter, making use of the definitions stated above. This also concludes the background theory part of this thesis, and we move on to the case study onto which a custom-written BAP algorithm eventually will be applied.

A note on column management

After some iterations of the BAP execution, the ever-growing common column pool Q' will most certainly contain columns that, for many subproblems, are not *"good"* columns. For such subproblems these columns will only increase the computational demand for solving the RMP without any actual gain. This is due to the simple reason that different subproblems (different sets of appended cuts) processed out of the subproblem pool C will most probably generate quite different columns, due to the variety of different appended constraints (cuts) at different positions in the BAP tree. This motivates incoorporating some kind of column management that, during the BAP process, dynamically cleans, modifies, and updates the column pool. It is however beyond the scope of this thesis to go into details of such column management, but it should be noted that the effect of such an implementation in the BAP process could have quite an effect on the overall performance of the algorithm in practice, especially for problems that process a large amount of nodes/subproblems in a large BAP tree. Indeed, it could be an interesting parallel study to this thesis to focus on heuristic methods for clever problem-specific column management, not only the removal of obsolete non-basic columns, but also the possible modification of basic ones.

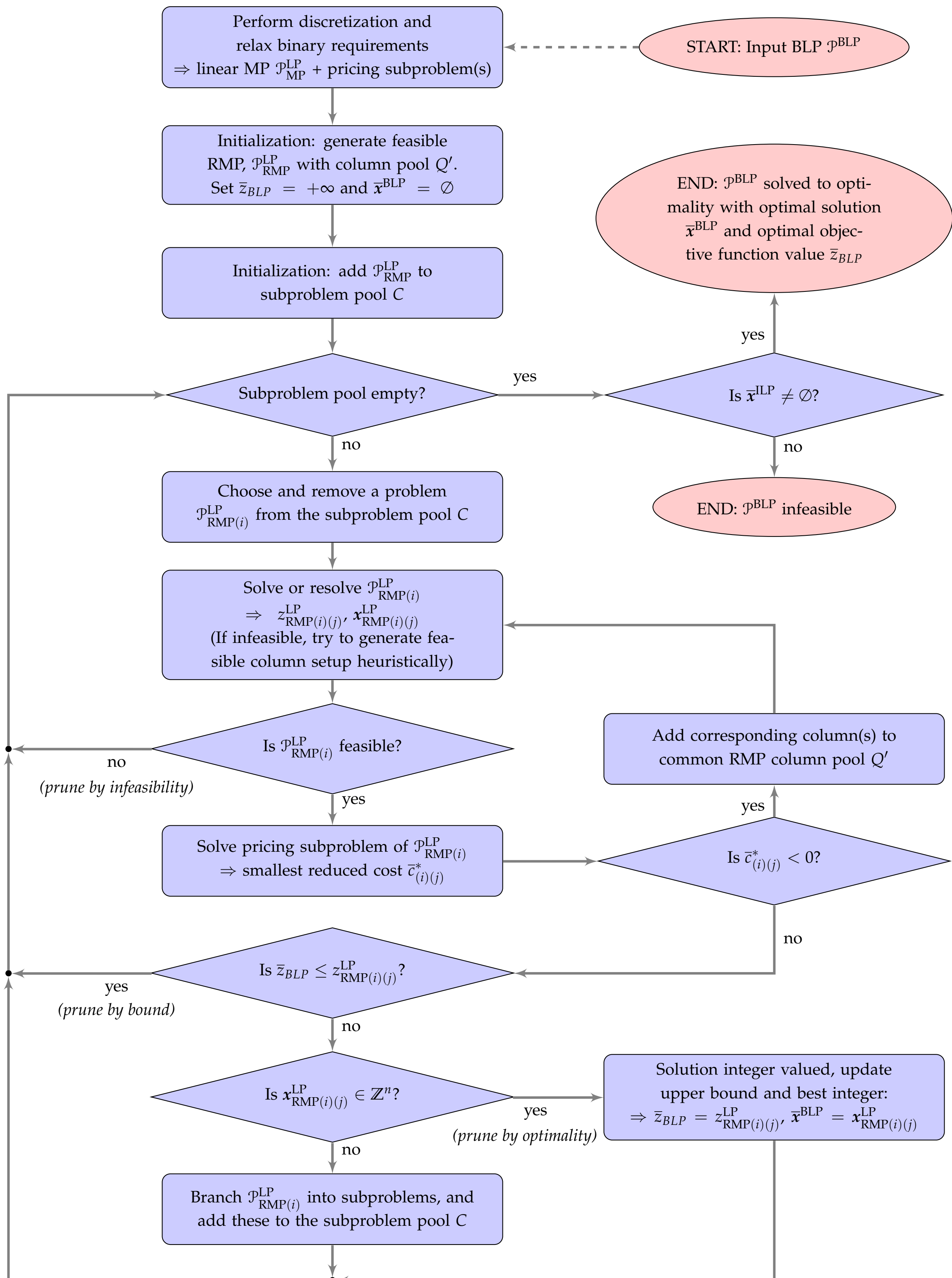


FIGURE 4.4: BAP flowchart.

Chapter 5

A Case Study: the Opportunistic Replacement Problem

In this chapter we study the problem of opportunistic maintenance scheduling—the *opportunistic replacement problem* (ORP)—applied to the maintenance of the RM12 aircraft engine. We begin in the first section by explaining necessary specifications of the RM12 engine, the appearance of the engine and certain limitations that we will impose on the associated ORP. In the following section we state a mathematical optimization model to describe the problem of minimizing the cost of the maintenance schedule, given the prerequisites from the previous section. In the last section, the model—an MBLP—is approached by making use of Dantzig-Wolfe decomposition for integer programs—previously covered in Section 3.3—, discretization. The method is used to derive a column generation set-up, containing an RMP with continuous variables, and a number of associated BLP subproblems. The goal is to implement the achieved column generation problem into a problem-specific BAP framework, with the ultimate objective to solve the original MBLP problem.

The ORP has been extensively studied by the authors of [2], from whose work [1] the original MBLP model presented in this chapter is taken. Particularly [1] covers the optimization of opportunistic replacement activities applied to the case study[1] of the RM12 aircraft engine, whereas [2] is a great reference for the general theory and background of the ORP. Moreover, [24] and [25] considers and thoroughly presents theory as well as results for the stochastic ORP (SORP), a special case which incoorporates stochastic component lives as a way to deal with the naturally non-deterministic nature of these lives. In a more recent study, [26], an extension of the ORP is presented,

[1]For a reader interested in other case studies on the subject, see [22] and [23], for studies of opportunistic maintenance scheduling models of an offshore wind power system and for shaft seals in feed-water pump systems in nuclear power plants, respectively.

laying out—in this context—the idea of *preventive* maintenance within the concept of maintenance planning, introducing the preventive maintenance scheduling problem with interval costs (PMSPIC). The PMSPIC approach is somewhat associated with the SORP approach in the sense that they both try to account for uncertainty in the component lives, but where the latter defines an explicit stochastic schedule, the former instead uses the concept of *risk*. The PMSPIC approach modifies the ORP into a bi-objective problem, with an associated Pareto front consisting of, naturally, a cost measure, but also a risk measure, to be minimized. This is—from the viewpoint of the author of this thesis—very promising, not only from a mathematical viewpoint, but also by considering that *risk* is a measure that can be gauged also by professionals that may not be proficient in mathematics, but have valuable knowledge in other relevant areas, e.g., in the discipline that a specific case study covers. There is always a value in bridging mathematics—especially the qualitative understanding of it—with the practical fields onto which its applied theory is to be employed.

For the purpose of this thesis, however, we now move and and focus solely on the ORP with deterministic component lives, and we proceed to present the background of the case study, as presented in [1] .

5.1 Background

Consider the graphical representation of the RM12 engine, displayed in Figure 5.1. There are seven distinguishable *modules* in the engine, each module consisting of one or several *components*. The fan, compressor, burner, high pressure turbine, low pressure turbine, and after burner modules are all mounted on a common *axle*, whereas the gear box module is mounted as a separate extension from the compressor module.

The components within each module have different *lives*, where a component is said to be *failing* when its life has run out. Without any deeper knowledge about the case study at hand—using only common sense in the context of the importance of functionality of an *aircraft* engine—the main objective of the RM12 engine maintenance planning could speculatively be described as the task of scheduling the maintenance of the engine such that it at no time contains failing components. In the eyes of a student of optimization theory, such a schedule could be said to be feasible, and the task of optimization would then come down to finding the *best* feasible schedule. We are not yet ready to interpret the meaning of *best*—how to rate the quality of a feasible schedule, how to interpret the adjective *opportunistic* to *'maintenance scheduling'*, or according to what actual rules we construct such a feasible schedule; perhaps failing components are allowed in the engine after all—but shall return to the subject

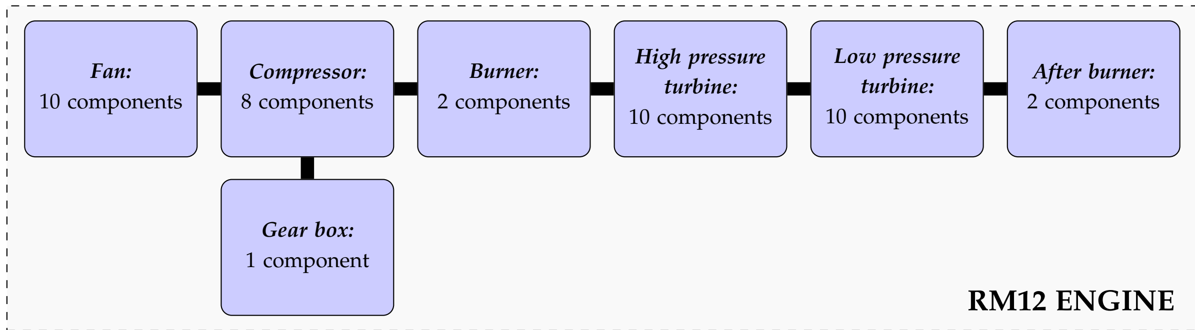


FIGURE 5.1: Graphical representation of the engine and its modules. For this study, dependencies between components *within* each module are ignored.

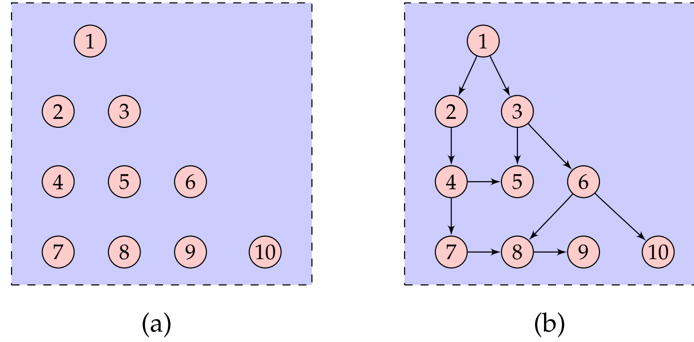
after some preparations. We begin by focusing on the lives of the components.

All components considered in this project are deemed *critical*, i.e., they cannot possibly be allowed to fail before replacement. As such, the lives of each of these components are assumed and set to have a *deterministic* life, calculated using methods to ensure that the vast amount of components of a certain type have at least the life of the corresponding deterministic value. I.e., the deterministic life serves as a high percentage one-sided confidence bound, for a population of specific components.

For maintenance of *non-critical* components, however, one can impose *stochastic* lives, to account for the fact that most components will in fact live longer—a few, however, will live shorter—than a pre-determined deterministic life. Opportunistic maintenance scheduling with stochastic components will not be studied in the scope of this thesis, but for a reader interesting in the concept, see [24] and [25].

Now, in the process of maintaining the engine, we *replace* failing components within the modules, and use each such *maintenance occasion* as an *opportunity* to possibly replace other components that are yet to fail. To replace a component, we need to *remove* the component from its module. We shall assume that, for a given module, any component can be removed without considering the neighboring components, i.e., it is assumed that there are no dependencies between components *within* modules. In reality, there exists dependencies between components within modules, and prior to removing a certain component we might need to remove other components. Figure 5.2 displays an example of a module with eight components, with or without dependencies between the components, where the *intradependency network* in Figure 5.2(b) is to be interpreted as different ways, or routes, to access different components within the module[2]. Note that due to our non-dependency assumption on components within modules, in the scope of this thesis, we will consider all modules to be of the principal

[2]E.g., in the example given in Figure 5.2(b), prior to accessing component 5 for removal, we can either remove components 1, 2 and 4—in the order given—or alternatively components 1 and 3.



(a) (b)

Figure 5.2: Example of a module with eight components, with (a) no dependencies and (b) dependencies between components. Dependencies are here to be interpreted as different ways to access components within a module.

structure illustrated in Figure 5.2(a).

To remove a component from a module, we need to *maintain* the associated module. In order to maintain a module, we need to perform necessary *activities* to be able to release the module from the engine. E.g., to maintain a certain module, prior to the module release we need to perform the activity of removing the module at hand. Prior to performing this activity, however, we might need to perform other activities, such as removing intermediate modules. From the sketch of the engine in Figure 5.1, we define a *dependency network* for the different activities, where we define one activity for the removal of each of the seven modules, as well as one activity for the removal of the axle, resulting in eight activities. Figure 5.3 displays this activity dependency network, in which arcs are to be interpreted as follows. If there is an arc from activity A_1 to activity A_2 in the dependency network, then prior to performing activity A_2 we need to perform activity A_1. E.g., to remove the burner, we first need to remove the high pressure turbine, prior to which we need to remove the low pressure turbine, and so on.

Finally, to maintain any module in the engine, naturally we need to maintain the full engine, i.e., recall the engine for maintenance. For this purpose, we define a number of *possible* maintenance occasions, in which we are allowed to take the engine in for maintenance. These possible maintenance occasions make up our *planning period*, over which we want to find the best possible[3] opportunistic maintenance schedule, i.e., the optimal schedule with respect to the mathematical program presented in the next section. A natural construction for the possible maintenance occasions is a uniform mesh, where each discrete step corresponds to a possible maintenance occasion. The distance between mesh points correspond to some real time measure, e.g., a num-

[3]We return shortly to how to define the *'best possible'* opportunistic maintenance schedule.

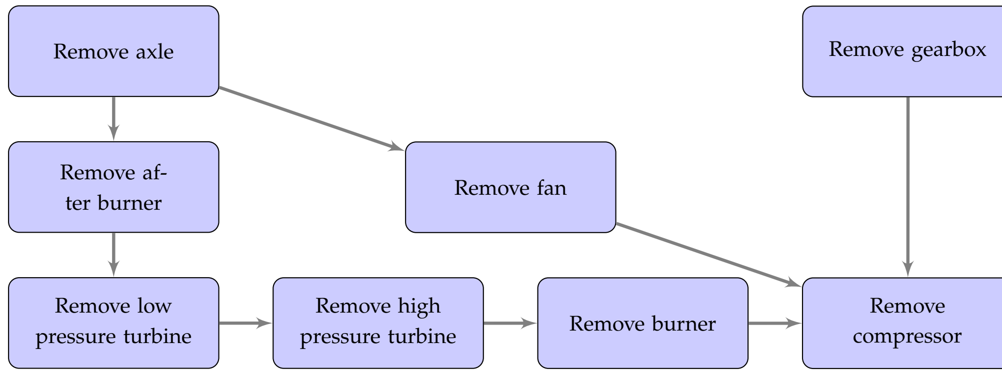


FIGURE 5.3: Dependency network for the set of different activities.

ber of days, and the size of the mesh—the number of discrete steps—consequently corresponds to the length of the planning period.

Naturally there's a cost associated with each of the procedures described above. When replacing a component, there's a cost for the new component and furthermore labour costs for the working hours needed for the procedure, labour costs that also arise when performing different activities. Moreover, there's a relatively high cost related to recalling the full engine for maintenance. Given these costs, we can deduce a measure of the quality of a given maintenance schedule. Our objective is to find the opportunistic maintenance schedule with the *minimum total cost*, under the constraint that the schedule must be valid—i.e., feasible—in the sense that during no time in the planning period does the engine contain components that have been in the engine longer than their deterministic lives.

We now have the necessary prerequisites to state a formal optimization program, describing the problem above. We move on to the next section, to see how the ORP—for the given case study—can be described as an MBLP.

5.2 Mathematical Formulation

We describe the ORP as a formal mathematical optimization problem. To do so, we define a number sets, describing the different key parts in the engine and its maintenance process, such as different modules or activities, as well as the discrete time mesh over the scheduling period. Thereafter, we introduce a number of decision variables describing different decisions regarding the maintenance of the engine, and finally a number of constants are used to express the objective function cost of these decisions, as well as the lives of the intramodule components.

We start by defining the necessary sets, as

$$\mathcal{T} := \text{the set of the discrete time steps } t \in \mathcal{T} \text{ that represent the planning, period, i.e., for a number of } T \geq 2 \text{ time steps, } \mathcal{T} := \{0, \ldots, T-1\}; \quad (5.1a)$$

$$\mathcal{M} := \text{the set of all modules } m \in \mathcal{M} \text{ in the engine;} \quad (5.1b)$$

$$\mathcal{N}^m := \text{the set of all components } i \in \mathcal{N}^m \text{ included in module } m \in \mathcal{M}; \quad (5.1c)$$

$$\mathcal{A} := \text{the set of all activities } n \in \mathcal{A} \text{ necessary to separate the engine into modules;} \quad (5.1d)$$

$$\mathcal{A}^m := \text{the set of all activities } n \in \mathcal{A}^m \text{ necessary to perform directly before releasing module } m \in \mathcal{M}; \quad (5.1e)$$

$$\mathcal{A}(n) := \text{the set of all activities } n' \in \mathcal{A}(n) \text{ from which there are arcs directed toward activity } n \in \mathcal{A} \text{ in the activities, dependency network.} \quad (5.1f)$$

Continuing, we define the following variables:

$$x_{it}^m := \begin{cases} 1, & \text{if component } i \text{ in module } m \text{ is replaced at time } t, \\ 0, & \text{otherwise;} \end{cases} \quad (5.2a)$$

$$w_t := \begin{cases} 1, & \text{if the engine is maintained at time } t, \\ 0, & \text{otherwise;} \end{cases} \quad (5.2b)$$

$$v_{nt} := \begin{cases} 1, & \text{if activity } n \text{ is performed at time } t, \\ 0, & \text{otherwise;} \end{cases} \quad (5.2c)$$

$$z_t^m := \begin{cases} 1, & \text{if module } m \text{ is maintained at time } t, \\ 0, & \text{otherwise,} \end{cases} \quad (5.2d)$$

where $t \in \mathcal{T}$, $i \in \mathcal{N}^m$, $m \in \mathcal{M}$ and $n \in \mathcal{A}$. Finally, we define the necessary constants, connecting decisions with their associated costs as well as lives of intramodule components, as

$c_{it}^m :=$ the cost to replace component i in module m at time t;

$d_t :=$ the cost to recall the engine for maintenance at time t;

$b_{nt} :=$ the cost to perform activity n at time t;

$T_i^m :=$ the deterministic life of component i in module m;

$\overline{T}_i^m :=$ the deterministic remaining life—at the start of the planning period—of the used indvidual of component i in module m;

$\delta_m :=$ a binary parameter, allowing us to study smaller instances of the engine,

where, as above, $t \in \mathcal{T}$, $i \in \mathcal{N}^m$, $m \in \mathcal{M}$ and $n \in \mathcal{A}$. We assume that all cost coefficients are positive and that all component lives are longer than one discrete time step[4], i.e., that $T_i^m \geq 2$. Finally, we assume that all objective function cost constants are unchanged over time, i.e., $c_{it}^m = c_i^m$, $d_t = d$, and $b_{nt} = b_n$, $t \in \mathcal{T}$.

Using these sets, variables and constants, we can write the ORP—with the simplification that no intramodule component dependencies will be taken into consideration—as the following MBLP to

$$\underset{x,z,v,w}{\text{minimize}} \quad \sum_{t=0}^{T-1} \left(\sum_{m \in \mathcal{M}} \left[\sum_{i \in \mathcal{N}^m} c_i^m x_{it}^m \right] + d w_t + \sum_{n \in \mathcal{A}} b_n v_{nt} \right), \tag{5.3a}$$

$$\text{subject to} \quad z_t^m \leq \delta_m, \qquad t = 0, ..., T-1,\ m \in \mathcal{M}, \tag{5.3b}$$

$$\sum_{t=0}^{\overline{T}_i^m} x_{it}^m \geq \delta_m, \qquad i \in \mathcal{N}^m : \overline{T}_i^m \leq T-1,\ m \in \mathcal{M}, \tag{5.3c}$$

$$\sum_{t=l}^{T_i^m+l-1} x_{it}^m \geq \delta_m, \qquad \begin{cases} l = 1, ..., T - T_i^m, \\ i \in \mathcal{N}^m : T_i^m \leq T-1,\ m \in \mathcal{M}, \end{cases} \tag{5.3d}$$

$$x_{it}^m \leq z_t^m \leq w_t, \qquad t = 0, ..., T-1,\ i \in \mathcal{N}^m,\ m \in \mathcal{M}, \tag{5.3e}$$

$$z_t^m \leq \sum_{n \in \mathcal{A}^m} v_{nt}, \qquad t = 0, ..., T-1,\ m \in \mathcal{M}, \tag{5.3f}$$

$$v_{nt} \leq v_{n't}, \qquad t = 0, ..., T-1,\ n' \in \mathcal{A}(n),\ n \in \mathcal{A}, \tag{5.3g}$$

$$x_{it}^m, z_t^m, v_{nt}, w_t \in \{0, 1\}, \qquad \begin{cases} t = 0, ..., T-1, \\ i \in \mathcal{N}^m,\ m \in \mathcal{M},\ n \in \mathcal{A}, \end{cases} \tag{5.3h}$$

where the binary parameter δ_m, $m \in \mathcal{M}$, as mentioned above, allows us to study smaller instances of the RM12 engine model, in which only a subset of the seven original modules are considered. By default, $\delta_m = 1$, $m \in \mathcal{M}$, i.e., including all modules, in which case the constraints (5.3b) become redundant, as the z_t^m variables are allowed to take only binary values, by (5.3h). The objective function, which is to be minimized, describes the cost of the maintenance over the full planning period. The constraints (5.3d) ensure that any new component inserted into the engine is replaced before its deterministic life runs out. The similar constraints (5.3c) make sure that any used components, contained in the engine at the start of the planning period, is replaced before they fail. If a component is to be replaced, then the module in which

[4] If there is a component with life $T_i^m = 1$, we have to maintain the associated module m, and hence the full engine, at all time steps, a scenario in which *opportunities* to replace components in module m arise at *every* time step. This would imply a severe simplification of the model, and hence, without loss of generality, we assume that $T_i^m \geq 2$.

the component is contained must be maintained at the same time/occasion. Furthermore, if, at any time, any module is maintained, then naturally the full engine must be brought in for maintenance. These two properties are enforced by the constraints in (5.3e). The constraints (5.3f) describe that, when a module is to be maintained, the necessary activity must be performed directly before releasing the module. The following constraints (5.3g) ensure that for any activity to be performed, the necessary preceding activities, described in the activities, interdependency network in Figure 5.3, are performed. Finally, the constraints (5.3h) ensure that all decision variables take binary values.

Before proceeding, we realize—by the following discussion—that we can relax, continuously, the integrality of the variables v_{nt} and the w_t. The variables w_t are present in only one constraint—the rightmost constraints of (5.3e)—with a positive coefficient of magnitude 1, *'alone'* with another binary variable, z_t^m, that also have positive coefficient of magnitude 1. The variables v_{nt} reside in similar constraints, while the sum in the constraints (5.3f) actually always contain exactly one v_{nt} variable. Finally, the objective function cost coefficients of w_t and v_{nt} are both positive. Consequently, as long as the variables x_{it}^m and z_t^m are binary, the variables v_{nt} and w_t will take binary values, even if relaxed to take continuous values.

We move on to Section 5.3, where we will approach the program (5.3) making use of the Dantzig-Wolfe decomposition principle, with the goal of transforming the problem into an RMP with associated column generation subproblems.

5.3 Reformulation into a Column Generation Problem

The first step in the decomposition procedure is to identify some kind of special in the constraints, matrix, deciding which constraints to consider as complicated—constraints that will remain in the MP—as well as which constraints that will be used to construct one or several subproblems. We note that there exists two quite obvious ways in which the full constraint matrix can be seen as block-angular, with the blocks defined by either (1) the T time steps or (2) the $|\mathcal{M}|$ modules.

For the first method, when specifying the block-structure in the constraint matrix, we consider all the variables x_{it}^m, z_t^m, v_{nt} and w_t, as they all contain an index in the set $\mathcal{T}$. We realize that the constraints (5.3c–5.3d) can be seen as the complicating constraints—in the sense that they connect variables from different time steps in the same constraint—to be left in the MP (cf. (3.9)), whereas (5.3b) and (5.3e–5.3g) can be seen as the block-constraints to be used when constructing the T subproblems (cf. (3.10)), one for each discrete time step.

For the second method, only the x_{it}^m and z_t^m variables will be considered when identifying the block angular constraint matrix structure, as they are the only variables with an index in the set $\mathcal{M}$. We realize that this means that the v_{nt} and w_t variables will remain in their original shape in the MP, as they can't possibly be part of a subproblem. Hence, any constraints connecting the x_{it}^m and z_t^m variables with the v_{nt} or w_t variables will be seen as complicating. Moreover, constraints containing only the v_{nt} and w_t variables can be seen as a kind of external constraints, left in the MP as they are, with no direct connections to the subproblems. Consequently, the constraints (5.3f) and the rightmost constraints of (5.3e) (i.e., $z_t^m \leq w_t$) will be seen as complicating and remain in the MP along with the external/independent constraints (5.3g). Further, we identify the block-structured constraints as (5.3b–5.3d) along with the leftmost constraints of (5.3e) (i.e., $x_{it}^m \leq z_t^m$), giving rise to $|\mathcal{M}|$ independent subproblems.

We will choose to attack the problem using the latter method, viewing subproblems as module-specific, as each subproblem would then be localized—in the view of the real problem described by the model—in a somewhat independent part of the engine. Of course, module dependencies cannot be ignored, but, in short terms, each subproblem $m \in \mathcal{M}$ generates columns that describe full feasible schedules corresponding to the associated module m.

Before continuing, we make a short note. Assume that we continuously relax the x_{it}^m and z_t^m variables in the program (5.3). We can then, in accordance with Section 3.2, consider the block-structured constraints—divided in module-independent blocks—along with continuously relaxed basic[5] constraints on the variables x_{it}^m and z_t^m as $|\mathcal{M}|$ independent bounded polytopes P^m, $m \in \mathcal{M}$. We will return to these polytopes shortly, when using discretization to represent the variables x_{it}^m and z_t^m.

We cover some preparations before finally presenting the RMP with its associated column generating subproblems.

5.3.1 Preparations

We will state these preparations with regard to the RMP and its associated subproblems. As our interest lies in generating columns to the RMP, we will not explicitly state the actual MP, but directly present the RMP. Furthermore, as our goal is to implement the RMP along with its subproblems into a BAP algorithm, we will consider the continuous relaxation of the achieved RMP, in line with the discussion in the last section of Chapter 3. Hence, our goal is to reach a continuously relaxed RMP with

[5] A continuous relaxtion of the binary constraints $x_{it}^m, z_t^m \in \{0,1\}$ yields the constraints $x_{it}^m, z_t^m \in [0,1]$.

BLP subproblems. For further reference, we denote this RMP by $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, and its $|\mathcal{M}|$ associated subproblems by $\mathcal{P}^{\text{BLP}}_{\text{SP}(m)}$, $m \in \mathcal{M}$. The specific RMP itself, along with the subproblems, will be stated after the preparations.

Now, let A denote the coefficient matrix corresponding to the complicating constraints[6] remaining in the RMP, i.e., A is the constraint matrix associated with constraints (5.3f) and the rightmost constraints of (5.3e) (i.e., $z_t^m \leq w_t$), in the original problem. With this, the complicating constraints can be described on the standard form

$$A\tilde{\boldsymbol{x}} \geq \boldsymbol{b}, \tag{5.4}$$

where the vector $\tilde{\boldsymbol{x}}$ contains *all* the variables in the program, i.e., $\tilde{\boldsymbol{x}} = \left[\boldsymbol{x}^{\text{T}}\ \boldsymbol{z}^{\text{T}}\ \boldsymbol{v}^{\text{T}}\ \boldsymbol{w}^{\text{T}}\right]^{\text{T}}$, with

$$\boldsymbol{x} = (\boldsymbol{x}^m)_{m\in\mathcal{M}},\ \text{where}\ \boldsymbol{x}^m = (x_{it}^m)_{i\in\mathcal{N}^m,\, t\in\{0,\ldots,T-1\}},\ m \in \mathcal{M}, \tag{5.5a}$$

$$\boldsymbol{z} = (\boldsymbol{z}^m)_{m\in\mathcal{M}},\ \text{where}\ \boldsymbol{z}^m = (z_t^m)_{t\in\{0,\ldots,T-1\}},\ m \in \mathcal{M}, \tag{5.5b}$$

$$\boldsymbol{v} = (v_{nt})_{n\in\mathcal{A},\, t\in\{0,\ldots,T-1\}}, \tag{5.5c}$$

$$\boldsymbol{w} = (w_t)_{t\in\{0,\ldots,T-1\}}, \tag{5.5d}$$

and where the vector $\boldsymbol{b}$ contains the corresponding right-hand-sides of the constraints.

The matrix A contains non-zero columns corresponding to the variables $\boldsymbol{v}$, $\boldsymbol{w}$ and $\boldsymbol{z}$. Hence, we consider A in the decomposed form

$$A = [A_x\ A_z\ A_v\ A_w] = [\mathbf{0}\ A_z\ A_v\ A_w]\,, \tag{5.6}$$

where the second equality follows from the fact that A_x contain only zeros, as none of the complicating constraints contain the variables x_{it}^m.

Now, we further decompose the coefficient matrix A_z as $A_z = \left[A_z^1\ A_z^2\ \ldots\ A_z^m\right]$, matching the $\boldsymbol{z}^m$ variables in (5.5b), $m \in \mathcal{M}$, and express (5.4) using (5.5) and (5.6), yielding

$$A\tilde{\boldsymbol{x}} = A_z\boldsymbol{z} + [A_v\ A_w]\begin{bmatrix}\boldsymbol{v}\\ \boldsymbol{w}\end{bmatrix} = \begin{bmatrix}A_z^1 & \ldots & A_z^m\end{bmatrix}\begin{bmatrix}\boldsymbol{z}^1\\ \vdots\\ \boldsymbol{z}^m\end{bmatrix} + [A_v\ A_w]\begin{bmatrix}\boldsymbol{v}\\ \boldsymbol{w}\end{bmatrix} \geq \boldsymbol{b}. \tag{5.7}$$

Now, for the complicating constraints, any non-zero coefficients corresponding to variables present in the subproblems are contained in the matrices A_z^m, $m \in \mathcal{M}$. We will make use of this shortly, when creating the objective (pricing) functions the RM subproblems.

[6]Note that these constraints *do not* include (5.3g), involving only v_{nt}, which we consider *external to or independent* of the column generation process.

Now, let $X^m = P^m \cap \{0,1\}^{|(x^m)^\mathrm{T}\,(z^m)^\mathrm{T}|}$, $m \in \mathcal{M}$, where P^m is as described above and as follows. Assuming that the binary constraints (5.3h) for the variables x_{it}^m and z_t^m have been continuously relaxed (i.e., yielding $x_{it}^m, z_t^m \in [0,1]$), P^m represents $|\mathcal{M}|$ independent bounded polytopes P^m, $m \in \mathcal{M}$, corresponding to—in addition to the continuously relaxed binary constraints (5.3h)—$|\mathcal{M}|$ module-independent block-structured constraints, consisting of the constraints (5.3b–5.3d) and the leftmost constraints of (5.3e) (i.e., $x_{it}^m \leq z_t^m$). We make use of Theorem 4 of Chapter 3 and use discretization to represent the variables $\boldsymbol{x}^m$ and $\boldsymbol{z}^m$ as follows (cf. (3.18))

$$X^m = \left\{ \begin{bmatrix} \boldsymbol{x}^m \\ \boldsymbol{z}^m \end{bmatrix} \in \mathbb{R}_+^{(|\mathcal{N}^m+1|)T} \;\middle|\; \begin{bmatrix} \boldsymbol{x}^m \\ \boldsymbol{z}^m \end{bmatrix} = \sum_{q \in \overline{Q}^m} \begin{bmatrix} \boldsymbol{p}_{xq}^m \\ \boldsymbol{p}_{zq}^m \end{bmatrix} \lambda_q^m,\ \sum_{q \in \overline{Q}^m} \lambda_q^m = 1,\ \boldsymbol{\lambda}^m \in \mathbb{Z}_+^{|\overline{Q}^m|} \right\}, \tag{5.8}$$

where $m \in \mathcal{M}$ and $\overline{Q}^m$, as stated in Theorem 4, contains the binary points of X^m. Recall now the trivial result that—with the binary requirement on the λ_q^m variables intact—any in point in X^m can be represented as exactly one of the binary points in X^m. This means that each X^m, $m \in \mathcal{M}$, describes the region of the corresponding block constraint in the program (5.3)—for each of the $|\mathcal{M}|$ independent block constraints—given that the binary constraints on the variables x_{it}^m and z_t^m are taken into account. Hence, we can safely replace the variables x_{it}^m and z_t^m in (5.3) using the representation above. We will attain an MBLP with binary variables λ_q^m and continuously relaxed variables v_{nt} and w_t, denoted $\mathcal{P}^{\text{MBLP}}$.

We realize that if we continuously relax the binary requirements on the λ_q^m variables in $\mathcal{P}^{\text{MBLP}}$, we will reach a linear MP—say $\mathcal{P}_{\text{MP}}^{\text{LP}}$—with a set of $|\overline{Q}|$ variables, $\overline{Q} = \overline{Q}^1 \times \cdots \times \overline{Q}^{|\mathcal{M}|}$, with associated coefficient columns $\boldsymbol{p}_q^m = \left[(\boldsymbol{p}_{xq}^m)^\mathrm{T}\ (\boldsymbol{p}_{zq}^m)^\mathrm{T} \right]^\mathrm{T}$, $q \in \overline{Q}^m$, $m \in \mathcal{M}$. As discussed in Chapter 3, $\mathcal{P}_{\text{MP}}^{\text{LP}}$ could possibly contain a huge number of columns, and we are better off considering only a small subset of them. This yields a linear RMP, with associated column pool $Q = Q^1 \times \cdots \times Q^m$, where $Q \subseteq \overline{Q}$ and $Q^m \subseteq \overline{Q}^m$, $m \in \mathcal{M}$. This is the RMP representation of the program (5.3) that we have been striving for, denoted $\mathcal{P}_{\text{RMP}}^{\text{LP}}$, and which we are soon ready to present explicitly.

Now, as for any RMP, we have one or several associated subproblems generating columns to the RMP column pool. In our case, there are $|\mathcal{M}|$ independent BLP subproblems, $\mathcal{P}_{\text{SP}(m)}^{\text{BLP}}$, each generating columns $\boldsymbol{p}_q^m$ to the $|\mathcal{M}|$ column pools Q^m, $m \in \mathcal{M}$. Further, by (5.8), the columns $\boldsymbol{p}_q^m \in Q^m$ are constructed as

$$\boldsymbol{p}_q^m = \begin{bmatrix} \boldsymbol{p}_{xq}^m \\ \boldsymbol{p}_{zq}^m \end{bmatrix}, \tag{5.9}$$

in which, say for a fixed m, each subcolumn $\boldsymbol{p}_{xq}^m$ and $\boldsymbol{p}_{zq}^m$ corresponds to the optimal solution $\boldsymbol{x}^{m*}$ and $\boldsymbol{z}^{m*}$, respectively, produced when solving the reduced cost minimization subproblem $\mathcal{P}_{\text{SP}(m)}^{\text{BLP}}$, i.e., the subproblem which generates the columns $\boldsymbol{p}_q^m$ to column pool Q^m, with $q = 1, \dots, |Q^m|$.

For simplification of the presentation, we introduce the following notation, allowing for the expression of the objective costs on vector form, as

$$\begin{aligned} \boldsymbol{c}_x^m &: \text{ the cost vector for } \boldsymbol{x}^m = (x_{it}^m)_{i \in \mathcal{N}^m,\, t \in \{0,\dots,T-1\}},\ m \in \mathcal{M}; \\ \boldsymbol{c}_v &: \text{ the cost vector for } \boldsymbol{v} = (v_{nt})_{n \in \mathcal{A},\, t \in \{0,\dots,T-1\}}; \\ \boldsymbol{c}_w &: \text{ the cost vector for } \boldsymbol{w} = (w_t)_{t \in \{0,\dots,T-1\}}. \end{aligned}$$

The variables z_t^m do not appear in the objective function and, hence, have no corresponding cost vector.

The *'full'* cost vector, used when calculating the cost of a column $\boldsymbol{p}_q^m$, is given by

$$\boldsymbol{c}^m = \begin{bmatrix} \boldsymbol{c}_x^m \\ \boldsymbol{0}^{|T|} \end{bmatrix}. \tag{5.10}$$

The zero-valued part in the cost vector corresponds to the variables z_t^m, which, as mentioned above, have no cost in the RMP objective function, but are present in the subproblems. The cost of a generated column $\boldsymbol{p}_{q_m}^m$ can then be described as $c_q^m = (\boldsymbol{c}^m)^{\mathrm{T}} \boldsymbol{p}_q^m$, $q \in Q^m$, $m \in \mathcal{M}$.

Recall that the A_z^m submatrices are the ones referred to when calculating the reduced costs in the subproblems, and note that in the RMP, the variables $\boldsymbol{v} = (v_{nt})_{n \in \mathcal{A},\, t \in \{0,\dots,T-1\}}$ appear on the form $\boldsymbol{v}_n$, where $\boldsymbol{v}_n = (v_{nt})_{t \in \{0,\dots,T-1\}}$.

Finally, we denote the dual variables associated with the complicating constraints—i.e., constraints (5.3f) and the rightmost constraints of (5.3e)—by $\bar{\boldsymbol{\mu}}$, and the dual variables associated with the $|\mathcal{M}|$ convexity constraints—arisen when we inserted the representation of the x_{it}^m and z_t^m variables into the program (5.3)—by $\bar{v}^m$, $m \in \mathcal{M}$.

We can now, finally, state the RMP, i.e., $\mathcal{P}_{\text{RMP}}^{\text{LP}}$, and its associated subproblems $\mathcal{P}_{\text{SP}(m)}^{\text{BLP}}$, $m \in \mathcal{M}$.

5.3.2 The RMP and its associated subproblems

Following the discretization procedure described in the last section of Chapter 3, we reach the RMP, $\mathcal{P}_{\text{RMP}}^{\text{LP}}(Q^1, \dots, Q^{|\mathcal{M}|})$, to

$$\underset{\boldsymbol{\lambda}, \boldsymbol{v}, \boldsymbol{w}}{\text{minimize}} \quad \sum_{m \in \mathcal{M}} \left(\sum_{q \in Q^m} \lambda_q^m (\boldsymbol{c}_x^m)^{\mathrm{T}} \boldsymbol{p}_{xq}^m \right) + \boldsymbol{c}_v^{\mathrm{T}} \boldsymbol{v} + \boldsymbol{c}_w^{\mathrm{T}} \boldsymbol{w}, \tag{5.11a}$$

$$
\begin{aligned}
\text{subject to} \quad & \sum_{q \in Q^m} \lambda_q^m \boldsymbol{p}_{zq}^m \leq \boldsymbol{w}, && m \in \mathcal{M}, && (5.11b)\\
& \sum_{q \in Q^m} \lambda_q^m \boldsymbol{p}_{zq}^m \leq \sum_{n \in \mathcal{A}^m} \boldsymbol{v}_n, && m \in \mathcal{M}, && (5.11c)\\
& v_{nt} \leq v_{n't}, && t = 0, ..., T-1,\ n' \in \mathcal{A}(n),\ n \in \mathcal{A}, && (5.11d)\\
& \sum_{q \in Q^m} \lambda_q^m = 1, && m \in \mathcal{M}, && (5.11e)\\
& \lambda_q^m \geq 0, && m \in q_m \in Q^m, \mathcal{M}, && (5.11f)\\
& w_t \geq 0, && t = 0, ..., T-1 && (5.11g)\\
& v_{nt} \geq 0, && t = 0, ..., T-1,\ n \in \mathcal{A}. && (5.11h)
\end{aligned}
$$

Note that the last two terms in the objective function are unchanged[7]—with regard to the original problem (5.3)—-when the decomposition is performed. That is, $\boldsymbol{c}_v^T \boldsymbol{v} + \boldsymbol{c}_w^T \boldsymbol{w} = \sum_{t=0}^{T-1} (d_t w_t + \sum_{n \in \mathcal{A}} b_{nt} v_{nt})$, since these variables are not represented using columns from subproblems.

Now, as explained in Section 5.3.1, we let $\bar{\boldsymbol{\mu}}$ denote the dual variables corresponding to the RMP complicating constraints, that is, (5.11b–5.11c) in the RMP above. Furthermore, let $\bar{v}^m$, $m \in \mathcal{M}$, denote the dual variables of the $|\mathcal{M}|$ convexity constraints (5.11e), respectively. Then, for each $m \in \mathcal{M}$, we have the associated BLP subproblem $\mathcal{P}_{\mathrm{SP}(m)}^{\mathrm{BLP}}(m, \bar{\boldsymbol{\mu}}, \bar{v}^m, A_z^m)$ to

$$
\underset{\boldsymbol{x}^m, \boldsymbol{z}^m}{\text{minimize}} \quad (\boldsymbol{c}_x^m)^{\mathrm{T}} \boldsymbol{x}^m - \bar{\boldsymbol{\mu}}^{\mathrm{T}} A_z^m \boldsymbol{z}^m - \bar{v}^m, \tag{5.12a}
$$

$$
\begin{aligned}
\text{subject to} \quad & z_t^m \leq \delta_m, && t = 0, ..., T-1, && (5.12b)\\
& \sum_{t=0}^{\overline{T}_i^m} x_{it}^m \geq \delta_m, && i \in \mathcal{N}^m : \overline{T}_i^m \leq T-1, && (5.12c)\\
& \sum_{t=l}^{T_i^m + l - 1} x_{it}^m \geq \delta_m, && l = 1, ..., T - T_i^m,\ i \in \mathcal{N}^m : T_i^m \leq T-1, && (5.12d)\\
& x_{it}^m \leq z_t^m, && t = 0, ..., T-1,\ i \in \mathcal{N}^m, && (5.12e)\\
& x_{it}^m, z_t^m \in \{0, 1\}, && t = 0, ..., T-1,\ i \in \mathcal{N}^m. && (5.12f)
\end{aligned}
$$

We now have enough prerequisites to proceed to implementing a problem-specific BAP algorithm for solving the MBLP (5.3), by making use of its discretization reformulation into the RMP (5.11) and the associated pricing subproblems (5.12).

[7]I.e., not rewritten using discretization, and hence not related to the convexity variables λ_q^m.

Chapter 6

Implementing a Problem-Specific Branch-and-Price Algorithm

When wishing to implement a BAP algorithm for a specific problem, we have two distinct choices. Either we make use of an existing BAB/BAP framework, or we construct one from scratch. Even if using an existing framework might sound simpler, none of the options are trivial. Most freely available existing frameworks still require a lot of programming work from the user, as they are imbedded in open-source software, programmed in some high-level language(s). Both options have their own benefits as well as disadvantages.

Existing frameworks are usually of high quality, one that is difficult to compete against with a from-scratch implementation, given the time limitations of a Master thesis project. But the hours spent and the satisfaction and learning possibilities gained from implementing your own framework could probably be seen as more worth than spending the same time trying to learn a framework and it's associated source code of someone else, no matter the quality of its documentation, or computational results. Hence, in this thesis project, the BAP algorithm has been implemented from scratch.

Section 6.1 briefly reviews some existing commercial and non-commercial MILP solvers, as well as frameworks associated with non-commercial software. Section 6.2 covers some segments in the BAP algorithm that needs to be specified before proceeding to implement the algorithm. Finally Section 6.3 concludes the chapter, with a short summary of implementation issues from the programmers point of view, e.g., how to plan and segment different major parts of the programming process.

6.1 Choosing an External LP Solver and Deciding on a Compliant and Appropriate Programming API

In this section we briefly review some existing commercial and non-commercial software for solving mixed integer linear programs. We also briefly cover how to possibly use these softwares either as existing frameworks for implementing a problem-specific BAP algorithm, or simply as LP solvers necessary for implementing the BAP algorithm and its associated framework from scratch. We clarify what we mean by the *'necessary LP solver'* above, and the reason for not considering the construction of such a solver as part of the BAP implementation.

A key component in any decently working MILP solver is a good LP solver, as many of the essential techniques embedded into the MILP solver, such as BAB, BAP, etc., are based upon pure LP solution techniques [27]. To implement an LP solver from scratch could perhaps be the purpose of a Master's thesis with focus on numerical linear algebra rather than optimization, as a principal part in the implementation would be to construct a library of fast matrix and vector operations, where issues as robustness and error propagation would be put into focus. For our purpose, however, we will use an already existing LP solver as to not deviate from the focus of this project, the implementation of the BAP framework. It is also the case in many of the existing branch(-cut)-and-price frameworks that the LP solver used is external, and one often has the choice to use different commercial or non-commercial LP solvers.

Two well-known commercial solvers—generally seen as the two of the top-tier MILP solvers in the world—are CPLEX [3] and Gurobi [28], of which the latter is quite a young company, whereas CPLEX has been around for more than 25 years. It is curiosa that the founders of Gurobi are all closely related to CPLEX; the president of Gurobi, Robert Bixby, was one of the co-founders of CPLEX, and the other two co-founders of Gurobi both held lead positions in the CPLEX R&D team before leaving for Gurobi [29].

For non-commercial software, the COIN-OR Branch-and-Cut MIP Solver, CBC [30], is a good example; an open source solver written in C++. Another well known non-commercial solver is the SCIP (Solving Constraint Integer Programs); an open-source MIP solver containing, among other things, a framework for branch(-cut)-and-price. The framework for SCIP was developed by the Ph.D. student Tobias Achterberg, as a supplement to his well-known Ph.D. thesis on Constraint Integer Programming (CIP) [27]. To continue on the curiosa above, after his Ph.D. work, Achterberg worked in the R&D team of CPLEX, but recently parted from CPLEX and joined the ranks of Gurobi [31]. Indeed a strong competence flow *from* CPLEX *to* Gurobi, and if we're to

trust Gurobi 5.6 benchmarks [32], we could ask ourselves: is CPLEX is starting to lag behind?

Now, returning to *this* thesis, for the purpose of our BAP implementation, we choose the CPLEX LP solver as the LP solver within our BAP framework, mainly due to the reason that the CPLEX is readily available at Chalmers through an academic license. This also makes it a suitable choice of commercial softare to compare the eventual results of our final implementation to.

With this decision, the choice of programming interface will naturally have to comply with the available CPLEX application programming interfaces (API:s). One relatively forthcoming API toward the CPLEX solver is AMPL [33], where a key strength lies in the ease of transferring a mathematical formulation of an optimization program straight-forwardly into a functional programming language, which is also hinted by full name of the API, A Mathematical Programming Language. In this project, the first column generation implementations were created using AMPL, which proved to be an effiecient learning tool, but for building larger frameworks, however, AMPL is quite slow and the language lack the versatility of an object-orientated programming language. Hence, once the basic concept of implementing column generation had been grasped, a passage was made to C++, combining standard C++ with the CPLEX C++ API, Concert Technology 2.9 [34], to build a fully functioning BAP framework.

6.2 Initial Set-Up: Algorithmic Specifications Necessary for Implementing a Basic BAP Framework

In this section we will describe the initial, or *basic*, implementation of the problem-specific BAP, applied to the ORP described in Chapter 5.

We start with some notational issues. We denote the continuously relaxed RMP as $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ and the associated subproblems as $\mathcal{P}^{\text{BLP}}_{\text{SP}(m)}$, $m \in \mathcal{M}$. Further, we will refer to the BAB subproblem pool as the *node list*, and consequently, any problems in this pool will be referred to as *nodes*. All nodes considered, i.e., nodes that are pruned or branched upon, along with nodes remaining in the node list, make up the *node tree*. For any given BAP iteration i, we define the associated active node's RMP as $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ with subproblems $\mathcal{P}^{\text{BLP}}_{\text{SP}(m)(i)}$, $m \in \mathcal{M}$.

Now, in Section4.3 we described how, given a continuously RMP with an associated BLP subproblem, we could solve the original BLP problem—from which the RMP and the pricing subproblem(s) were derived—by implementing column generation dynamically within a BAB framework. This general BAP algorithm was thereafter summarized in a flowchart, as presented in Figure 4.4.

Referring to this flowchart, we consider the most basic form of our growing BAP algorithm as an implementation where all parts of the flowchart are satisfied, if even only by simple methods. We've already covered the transformation of a specific BLP into an MP with associated pricing subproblems, as described on our case study in the previous chapter. Furthermore, even if implementing the basic structure of the BAB framework[1] as well as a functioning column generation might be a bit tricky and rather tedious, it's quite straightforward. Some steps of the algorithm, however, need to be more formally specified before reaching a working basic version of the BAP implementation. We list these steps below.

- *Initialization:* initialize the column pools (several pricing subproblems: several column pools) such that the root problem $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ becomes feasible.
- *Pruning nodes by infeasibility:* in the BAP implementation, heuristically make sure that an infeasible $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ is truly infeasible, with regard to the MP, as there may exist feasible columns that are not yet in the column pool.
- *Node picking rule:* specify a rule for how to choose nodes to remove from the node list.
- *Branching rule:* specify how, during any iteration i, to perform branching on the active problem/node, given a fractional solution to $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$.
- *Solving the subproblems:* as the subproblems describe BLP problems, solving them cannot be considered as trivial; this solution must be specified.

We proceed to specify the implementation of each of the four topics above.

Initialization

As mentioned above, the goal of this step is to initialize the column pool such that the root problem $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ becomes feasible, a task that will be performed heuristically. Further, as the initialization naturally generates only one column per column pool, the achieved feasible initial solution to the $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ also provides the first incumbent solution, feasible also in the *binary* RMP, i.e., an upper bound on the original MBLP's optimal objective value.

We will use a simple heuristic to generate an initial incumbent solution and an associated upper bound, but also add a $\mathbf{1}$-column (containing only 1-entries) to each column pool, to avoid the initial RMP to be too biased towards our initial heuristic

[1] Recall the core idea of BAB: a subproblem pool, or node tree, from which problems/nodes are removed and solved for pruning or branching.

solution. These **1**-columns can be readily used in the root node, but will be gradually ruled out by down cuts[2] as the algorithm proceeds deeper in the node tree.

The initialization heuristic generates initial columns, which correspond to the case in which the maintenance schedule for each separate module is module-locally optimal, i.e., an optimal solution to the case of a one-module ORP[3], for each module $m \in \mathcal{M}$. As we have assumed all costs to be constant in time, the one-module-case with deterministic lives can be readily solved to optimality using a heuristic to replace components as late as possible in the module. For details, see [2, pp. 8–9].

Initialization and heuristics has not been a subject of focus in the scope of this thesis, but rather a component necessary for the BAB/BAP algorithm to function. Hence the lack of formal definitions, particularly for the heuristics used. Tests on the problem instances chosen have shown that a combination of the heuristic for module-locally optimal initial solutions and the addition of the **1**-columns yields the best results, compared to using either of these column pool initializations separately, hence the choice of this combination.

As a side note, intuitively one might think that the *'quality'*—with quality referring to the value of the solution's associated objective function—of the initial incumbent solution is a factor that directly affects the performance of the BAB algorithm, in a sense that *'the better initial incumbent solution, the better BAB performance'*. This is, however, not always the case. In fact, while initial incumbent solutions with *'good'* bounds might improve performance for some problem instances and BAP implementations, for others, the reverse can hold. Assuming that a minimization problem is considered, low initial upper bounds might lead to larger BAB trees, and hence a worse overall performance. The reason for this is that some BAP implementations use node picking heuristics that make use of the current best incumbent solution (and associated upper bound), in which case an initial solution with a low lower bound might very well lead to choosing nodes that end up generating a larger BAP tree than if a *'worse'* initial solution had been used [35].

We move on to explain how to deal with nodes whose associated RMP's are infeasible.

Pruning nodes by infeasibility

Looking back at the RMP (5.11) derived in Chapter 5, it's quite apparent that infeasibility should not be an issue for the RMP. This, since one can always heuristically

[2]Recall, from Section 4.1, that down cuts refer to a binary variable being forced to take the value 0, by incorporating additional constraints to the program at hand.

[3]In the original ORP (5.3), let $f_m = 0$ for all $m \in \mathcal{M} \setminus \{\tilde{m}\}$ and $f_{\tilde{m}} = 1$, to attain the corresponding one-module problem for module $\tilde{m}$.

construct single columns that are feasible in the program, even if they are quite expensive w.r.t. objective value. Deep down in the BAB tree, however, it's theoretically possible that sufficiently many of the subsequent down cuts apply to the same original variable, i.e., concerning the replacement of same single component. This means that they all describe the non-allowance of replacement for sufficiently many consecutive time steps for this single component, where the number of succeeding time steps is larger than the deterministic life of the component. Now, in practice, this is quite an unlikely scenario, and any possible infeasibility of the RMP will most likely be due to a lack of feasible columns, columns which can readily be constructed heuristically.

With this in mind, we choose to solve this potential problem in the simplest possible manner. If, at any iteration i, the associated RMP problem $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ is infeasible, we call a simple but fail-safe heuristic to, if possible, generate columns to add to the RMP such that feasibility is regained. Given a number of down cuts in the active node, we generate columns in which all elements, for which the corresponding variables are *not* present in the down cuts, are given the value 1.

If, after generation, the RMP is still infeasible, the extremly unlikely scenario described above has occured, and the associated node can be pruned by infeasibility. If it is feasible, which is more likely, we can proceed with the column generation procedure.

A node picking rule

The general concept of *choosing* a node from the node list, and when solved, if given a fractional solution, *branching* the active node into two subproblems, i.e., two new nodes to be placed in the node list, is quite easy to grasp. A more thorough understanding of how to actually carry out this concept within a BAP implementation requires a definition of the choice of nodes from the node list, as well as of the choice of variables to branch upon when splitting the active problem node into subproblems.

We will begin by explaining the former, stating a *node picking rule*, and thereafter proceed to the choice of variables to branch upon—in case an active node cannot be pruned—by stating a *branching rule*.

Now, node processing can be summarized as a removal from the node list followed by a solution of the associated RMP to decide on a subsequent pruning or branching. While not depending on our initial heuristic solution to provide a good upper bound, we would like to choose nodes from the node list such that integer valued solutions are found relatively fast, i.e., that the best incumbent solution is quickly improved. In accordance to the theory of BAB for binary programs, we know that each branched node gives rise to two new nodes, one in which a down cut (some $x_i \leq 0$) has been

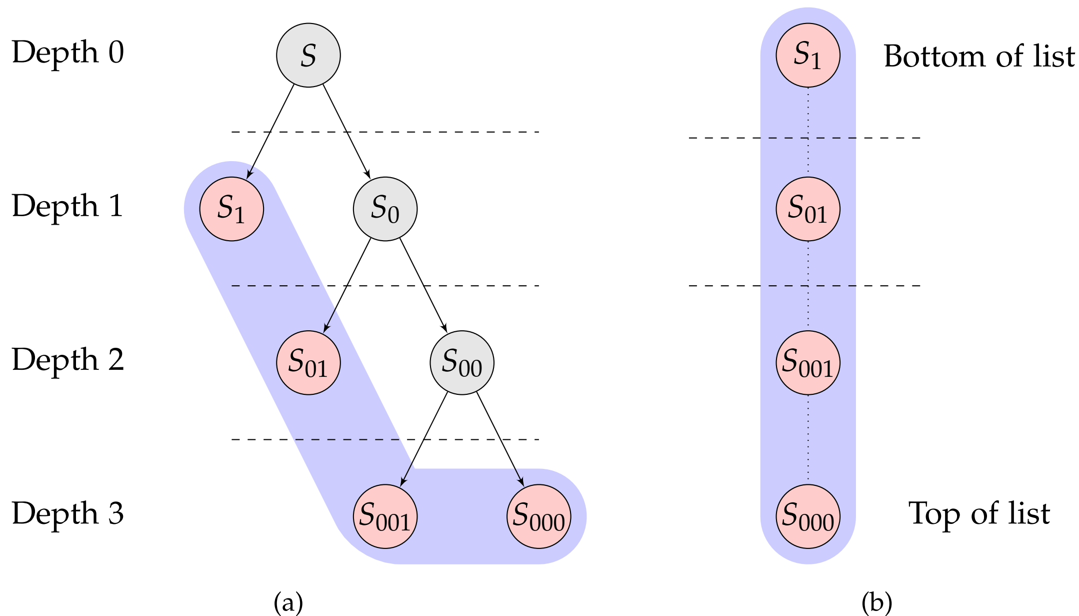


FIGURE 6.1: (b) stacking of nodes into the node list, after (a) three BAB iterations. For each new iteration, the node on top of the node list stack is removed from the list and solved.

appended, and one with an appended up cut (same $x_i \geq 1$). Our basic node picking idea is to move as deep as possible in the node tree, by choosing nodes from the node list that represent the deepest located nodes. This can be done quite easily by considering the node list as a *stacked data structure*, or short, just a *stack*. A stack be thought of visually as a pile of items, that are stacked upon each other. The stack is then characterized by two fundamental operations, *removal* of the top item of the stack, or the *addition* of a new item to the top of the pile. By constructing the node list in this fashion, a so called *Depth-First* (DF) node picking rule [9] is attained and in which the node most recently added to the list will be the one to be removed when a new BAP iteration starts (also known as LIFO: last in-first out). Figure 6.1 visualizes the concept in which the node list or stack is presented after three BAP iterations, in each of which the active node has been branched upon. Note that for each branched node we attain two new subproblems, which are added to the stack in the following order: first add the node containing a new up cut, and thereafter add the node containing a new down cut. This means we will attain a depth-first node picking rule with focus on down cuts, a logical choice as module maintenance occasions should rather be excluded than enforced, due to the fact that the final solution of module maintenance occasions is expected to be rather sparse.

We proceed to state a simple branching rule.

Specifying a branching rule

Now, during any BAP iteration i, if the optimal solution of the associated RMP $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$ is fractional, we have to branch the problem into two subproblems. From Chapter 4, we know that this is done by choosing a variable with a fractional value in the optimal solution, and generating one up cut and one down cut subproblem, by constraining the variable to take the values 1 and 0, respectively. We need to specify, however, exactly how the procedure of *choosing* such a variable is performed. Before introducing such a *branching rule*, we state some notations that will be used in presenting the rule. Note also, as is motivated in Section 4.2, that the branching is applied to the original variables, and not the RMP:s explicit variables.

Let $\bar{\boldsymbol{\lambda}}^{(i)}_{\text{RMP}}$ be the optimal solution to the RMP in the i:th iteration, i.e., $\mathcal{P}^{\text{LP}}_{\text{RMP}(i)}$, and let $\bar{\boldsymbol{z}}^{(i)}$ be the associated vector, representing the values of the variables z^m_t of the original problem (5.3)[4], $t \in \mathcal{T}$, $m \in \mathcal{M}$. Recall that these variables describe whether module m should be maintained at time t ($z^m_t = 1$) or not ($z^m_t = 0$). Further, let $\mathcal{K}$ be a set of candidates eligible to branch on, e.g., in our case, let $\mathcal{K}$ be the set of all fractional z^m_t (*original*) variables—representing module maintenance or not—in the optimal solution, i.e., $\mathcal{K} = \{(m,t) \mid z^m_t \in \bar{\boldsymbol{z}}^{(i)} \,:\, 0 < z^m_t < 1\}$.

Wolsey [9] describes the common branching rule of picking *the most fractional variable*. Noting that all non-continuous variables are binary, this corresponds to choosing the pair of indices in $\mathcal{K}$ that satisfies

$$\arg\max_{(m,t)\in\mathcal{K}} \min\{z^m_t, 1 - z^m_t\} \tag{6.1}$$

When testing this rule in a BAB framework upon small test problems with sparse solutions, however, it performs very poorly, even worse than just randomly picking a candidate from the set $\mathcal{K}$, something that is reported also in [21]. We wont perform any thorough analysis of the poor behavior of this rule, but we discuss shortly a logical explanation of it's performance.

From above, we've chosen to use the DF node picking rule with focus on down cuts, and if using the branching rule above, we will branch on variables with fractional values near 0.5. If we consider the fractional values in the span $(0,1)$ to be a simple measure of the certainty—from the associated LP's point of view—of which binary value the corresponding variable might take in an integer feasible solution, a value of 0.5 would correspond to the minimum amount of certainty. Hence, given our DF node picking strategy focused on down cuts, with the discussion above in mind, the

[4] Recall from Section 5.3 that, given $\bar{\boldsymbol{\lambda}}^{(i)}_{\text{RMP}}$, we can assemble $\bar{z}^{(i)} = \{\bar{z}^{m,(i)}\}_{m\in\mathcal{M}}$ using the relation $\bar{z}^{m,(i)} = \sum_{q\in Q^m} \bar{\lambda}^{(i)}_q \boldsymbol{p}^{m,(i)}_{zq}$, $m \in \mathcal{M}$.

most fractional branching rule does not seem to be a very good choice for the problem at hand. A more logical choice would be to choose variables with fractional values in the span $(0, 0.5]$, as these are—from the LP's local and simple point of view—inclined to take the value 0 in an integer feasible solution.

Algorithm 2 A basic branching rule

1: **procedure** BASIC-BRANCHING-RULE($\mathcal{K}$) ▷ **INPUT**: Set of branching candidates $\mathcal{K}$
2: ▷ Ordering of $\bar{z}_t^{m,(i)} \in \mathcal{K}$: increasing time steps nested within increasing
3: ▷ module numbers, i.e., $z_0^{1,(i)}, z_1^{1,(i)}, ..., z_{T-1}^{1,(i)}, z_0^{2,(i)}, ..., z_{T-1}^{2,(i)}, z_0^{3,(i)}, ..., z_{T-1}^{|\mathcal{M}|,(i)}$,
4: ▷ for all fractional $\bar{z}_t^{m,(i)}$.
5: **for each** $\bar{z}_t^{m,(i)} \in \mathcal{K}$ **do** ▷ Start testing variables of $\mathcal{K}$ in order
6: **if** $\bar{z}_t^{m,(i)} \in (0, 0.5]$ **then**
7: **return** $(m, t, \bar{z}_t^{m,(i)})$ ▷ Return first valid candidate found
8: **exit procedure** ▷ Exit variable picking procedure
9: **end if**
10: **end for each**
11: $(\tilde{m}, \tilde{t}, \tilde{z}) \leftarrow random(\bar{z}_t^{m,(i)} \in \mathcal{K})$ ▷ If not exited: select a random candidate
12: **return** $(\tilde{m}, \tilde{t}, \tilde{z})$ ▷ Return random candidate
13: **end procedure**

We will use the above arguments to state our initial branching rule, to be used in the basic BAP implementation. Given our list of branching candidates $\mathcal{K}$, under the assumption that $\mathcal{K}$ is non-empty, we choose the *first* variable with a value in the span $(0, 0.5]$. To measure what we mean by first, we order the z_t^m variables contained in $\mathcal{K}$ as follows, $z_0^1, z_1^1, ..., z_{T-1}^1, z_0^2, ..., z_{T-1}^2, z_0^3, ..., z_{T-1}^{|\mathcal{M}|}$, i.e., in a nested fashion with the ordered time span $t = 0, ..., T-1$ nested within the ordered module numbering, $m = 1, ..., |\mathcal{M}|$. In the quite unlikely case that none of the branching candidates in $\mathcal{K}$ possesses a value in the span $(0, 0.5]$, we just choose a branching variable randomly, and stack the up cut (≥ 1) on top of node list, in contrast to the default case, in which down cuts at stacked on top of the list. The branching rule is described with commented pseudocode in Algorithm 2.

We move on to describe the final specification of the basic BAP algorithm—how to solve the column generation subproblems—after which we describe pure implementation issues.

Solving the subproblems

In the last section of Chapter 3, we shortly discussed how BLP Dantzig-Wolfe subproblems—attained from a discretization procedure of a (mixed) BLP—possess properties

that would make them suitable for solving using LP based techniques. We covered one such general technique in the previous chapter, i.e., BAB. Indeed, as a simple solution method, the binary subproblems described in equation (5.12) could probably be solved using a continuous relaxation of all binary variables followed by an implementation into a standard BAB network, as their number would be small compared to the original problem (5.3). There exists, however, more advanced and a great deal better methods to solve binary problems like these, ones that also account for the knowledge of the real problem at hand, the fact that each subproblem describe a module-local ORP. This was the subject of the Master's thesis *'Solution approaches for the opportunistic replacement problem: Benders decomposition and Chvátal-Gomory cut generation'* [36].

For the scope of this thesis, however, we shall not plunge any deeper into this subject. Subproblems will be solved using CPLEX MIP Solver, a commercial optimization software for solving general MILP (MBLP) problems. And as expected, as the subproblems have quite a neat form, we will see they are solved relatively quickly by CPLEX. We will bear in mind, however, that a problem-specific method for solving these subproblems could arguably outperform CPLEX' general MIP solver, which would drastically improve the performance of the BAP implementation of this thesis project[5].

With the specifications above, we are ready to proceed to the implementation step of the project. We conclude the chapter with a brief summary the implementation from the programmers point of view.

6.3 Implementation of a Problem-Specific BAP Algorithm Using a High-Level Programming Language

As mentioned in Section 6.1, with the choice of CPLEX as LP solver, the choice of high-level programming language naturally landed on C++, given the well-documented CPLEX C++ API, Concert Technology 2.9 [34]. Here briefly provide an overview of the architecture of the BAP implementation.

Figure 6.2 depicts a programming dependency chart, containing the key programming modules in the BAP implemention associated with this thesis. Existing Concert classes and methods are used to model the RMP and its associated pricing subproblems, as well as the original MBLP model, as we want to solve the latter directly with CPLEX for readily comparison with our BAP framwork. As mentioned above, CPLEX

[5]The solving of the subproblems are nested into each iteration of the column generation procedure, which in itself is nested within the BAB process. Consequently, improving the solution process of the subproblems would have a tremendous effect on the execution time of the BAP procedure.

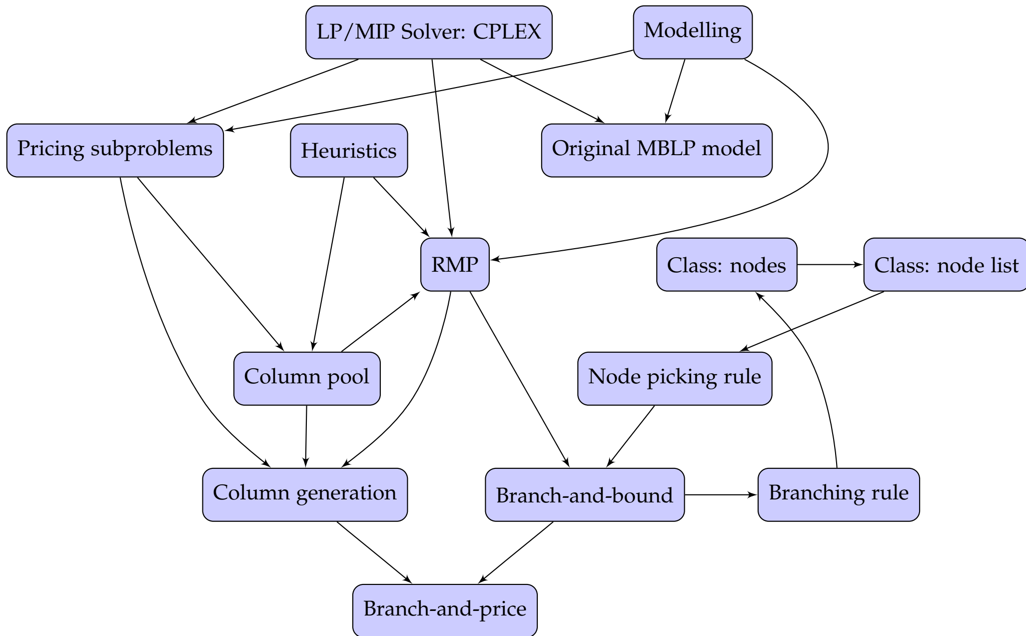


FIGURE 6.2: Programming dependency chart.

is used as the LP solver for the RMP, and also as the MIP solver for the pricing subproblems as well as the original MBLP model. Pricing subproblems and heuristics (initialization and infeasibility fix) add columns to the column pool, common for all BAB subproblems/nodes. The BAB subproblems are stored as node objects from a custom built node class, which most importanly, for each BAB subproblem (node), contains the additional appended cuts with reagard to the original RMP. The node objects are stored in a node list object, a custom built class that represent the active BAB tree. Node objects are created and placed in the node list after a branching rule has decided upon branching the variable for a node processed to be branched upon, and a node picking rule is used to decide which node to extract for processing from the node list. The column generation runs parallel with the BAB process, generating columns on-the-fly when solving the numerous RMP variations in the BAB tree, in so describing a BAP algorithm.

For in-depth details about the implementiaton, our well-commented C++ source code should prove an excellent reference.

Chapter 7

Tests and Results: Improving the Basic Branch-and-Price Implementation

Implementing a BAP algorithm can be seen as an ever-growing process. The number of possible algorithmic components eligible to enhancements is vast, and with the always ongoing development of new associated techniques, one realizes that to reach a *'final'* version of the BAP implementation there's a need to define some restrictions on what possible enhancements that are to be examined when expanding the first basic implementation[1]. In this chapter, we will identify/decide which factors that will be put in the spotlight when trying to improve the basic implementation, and thereafter perform a number of tests to decide whether to keep a proposed improvement or not.

The first section contains a discussion on which improvements that are to be attempted, and the following four sections cover the testing along with implementation decisions, based on the test results. In the final section, we compare the final BAB/BAP implementations with CPLEX MIP Solver [3].

[1]As a reference, Tobias Achterberg's well-known Ph.D. thesis on Constraint Integer Programming (CIP), supplemented by the software SCIP (Solving Constraint Integer Programs), as mentioned in the previous chapter; an open-source MIP solver containing, among other things, a framework for branch(-cut)-and-price [27]. The total project, as of SCIP v. 1.1.0 (released 30 September 2008) contains over 275 000 lines of C code, and is still growing steadily, as a part of the ZIB Optimization Suite.

7.1 Initial Set-Up: Building a Basic BAP Framework and Identifying Algorithmic Key Factors Prune to Possible Enchancement

In this section we discuss the most obvious flaws the initial BAP implementation, and decide on prospective factors of improvements. After this has been done, we present three test problem cases of the ORP—as presented in Chapter 5—which will be used to perform the tests of possible improvements.

We identify the following key factors prune to possible enchancement:

- Test 1, improving the column generation process: As described in the final paragraph of Chapter 4, no column management will be included in the BAP algorithm implemented in the scope of this thesis. Without column management, the column pool will grow rapidly while traversing the BAP tree, and the importance of a good column generator is apparent; we want to try to minimize the number of "bad" columns—columns that will seemingly never be used, but still end up in the column pool, in so increasing the computational complexity of the RMP:s—added to the column pool. For this reason, the first possible improvement step will concern the implementation and subsequent testing on a more sophisticated column generating process, focused on which dual variable values that should be sent to the pricing subproblems of the RMP:s.

- Test 2, examining alternative node picking rules: In Chapter 6, the Depth-First node picking rule was presented and chosen as the default node picking rule for our basic implementation. The second possible improvement step will be testing alternate node picking rules.

- Test 3, choice of type of LP solver: In the end of Chapter 2 we discussed the apparent pros and cons of the primal and dual simplex methods. The third possible improvement step will be testing which LP solver combination to use within the BAP framework.

- Test 4, examining a sophisticated branching rule: Finally, we will implement a sophisticated rule for choosing branching variables and examine whether it outperforms the simple branching variable rule presented as Algorithm 2 in the previous chapter.

For the first test: testing of the possible improvements for the column generation, testing is benchmarked for solving the RMP (5.11) to (LP) optimality. For the second

test: testing rules for node picking, the linear RMP (5.11) is solved to LP optimality, and BAB is thereafter applied to find the best binary valued solution (for the original variables) given the columns generated when solving the RMP. For the third test: testing which LP solver setup to use, the same scenario as for the second test is used. For the final test, introducing a sophisticated branching rule, the testing is benchmarked for the performance of the actual full BAP algorithm as applied to solving the MBLP (5.3), making use of it's discretization form, as derived and presented in Chapter 5.

Three test problem instances

Table 7.1 presents the setup for each of the three test problem instances[2] of the MBLP (5.3). For reference of notation, see Chapter 5. Note that only the minum and maximum deterministic lives for the components of each module are presented in the table, such that $T^m_{\min} \leq T^m_i \leq T^m_{\max}$, $m \in \mathcal{M}$, $i \in \mathcal{N}^m$. Moreover, all components are assumed new at the start planning period, i.e., $T^m_i = \overline{T}^m_i$, $i \in \mathcal{N}^m$, $m \in \mathcal{M}$,.

TABLE 7.1: Three test problem instances A, B and C, in which $|\mathcal{M}| = 7$ for all three instances. The second column describes the time span for each problem instance, and the columns following thereafter describe, for each module and problem instance, the number of components $|\mathcal{N}^m|$ and min/max lives of these, $T^m_{\min}$ and $T^m_{\max}$, respectively.

Instance	Time span	Module $m =$	1	2	3	4	5	6	7
A	$T = 35$	$\lvert\mathcal{N}^m\rvert :=$	5	3	2	7	3	7	1
		$T^m_{\min} :=$	3	5	7	11	13	17	19
		$T^m_{\max} :=$	20	24	23	29	30	31	19
B	$T = 50$	$\lvert\mathcal{N}^m\rvert :=$	7	4	2	8	7	8	1
		$T^m_{\min} :=$	3	5	7	11	13	17	19
		$T^m_{\max} :=$	45	43	23	45	49	49	19
C	$T = 100$	$\lvert\mathcal{N}^m\rvert :=$	8	8	2	9	13	15	1
		$T^m_{\min} :=$	3	5	7	11	13	17	19
		$T^m_{\max} :=$	60	83	23	61	99	93	19

We move on to cover details and results for each of the four tests.

[2]Note that the setup of these are not representations of real ORP instances, but have been constructed by the author of this thesis to serve as three different sized problem instances of the MBLP (5.3) used when testing and improving the BAP implementation.

7.2 First Modification: Dealing With the Tailing-Off Effect Using Weighted Dantzig-Wolfe Decomposition

The slow convergence—commonly called tailing-off effect—of column generation is a subject worth putting in the spotlight. One well-accepted source of this slow convergence is degeneracy in the optimal solutions of the RMP [37].

To deal with the tailing-off effect, we implement a dual stabilizing method, as proposed in [38], *weighted Dantzig-Wolfe decomposition*. Previously, for each iteration in the column generation process (see Subsection 5.3.2), the optimal dual variable values $\bar{\boldsymbol{\mu}}$ (and $\bar{v}$) were sent to the pricing subproblems for pricing of columns, whereas we will now use a convex combination of the active iteration's optimal dual variable values and the *'best dual variable values'* attained so far. We interpret *'best'* as the values yielding the highest lower bound on the optimal value of the MP. We will shortly see that the lower bound from the dual variables is in no way smoothly increasing, but is instead rather noisy—the so called *jo-jo effect* of CG—hence the need for a simple measure of the best dual variable values as the best dual variable values up to and including current iteration (i.e., "best" attained so far).

Let $\bar{\boldsymbol{\mu}}^{k+1}$ and $\bar{\boldsymbol{\mu}}^{\text{best},k}$ be the optimal dual variable values from the RMP in the $(k+1)$st iteration and the best optimal dual variable values from the RMP up to and including the kth iteration, respectively. We denote the dual variable values sent to the subproblems by $\hat{\boldsymbol{\mu}}^{k+1}$, where

$$\hat{\boldsymbol{\mu}}^{k+1} := \frac{1}{w_{k+1}}\bar{\boldsymbol{\mu}}^{k+1} + \frac{w_{k+1}-1}{w_{k+1}}\bar{\boldsymbol{\mu}}^{\text{best},k}, \tag{7.1}$$

$$\bar{\boldsymbol{\mu}}^{\text{best},k} \in \underset{i=1,\dots,k}{\arg\max} \left\{ L(\bar{\boldsymbol{\mu}}^{i}) \right\}, \tag{7.2}$$

and

$$w_{k+1} := \min \left\{ C, \frac{1}{2}(k + \text{ number of improvements of } L(\bar{\boldsymbol{\mu}}^{\text{best},k})) \right\}, \tag{7.3}$$

where $C \geq 2$ is a constant, and $L(\bar{\boldsymbol{\mu}}) := \bar{z}_{\text{RMP}} + \sum_{m\in\mathcal{M}} \bar{c}^{m*}$, i.e., the MP lower bound produced for that given iteration, as presented in the leftmost inequality in inequalities (3.16).

We proceed to compare the original and weighted DW decomposition methods.

Results

Figures 7.1–7.3 show the convergence of the upper and lower bounds for the two column generation implementations, for the test instances A, B, and C, respectively.

Clearly, the introduction of weighted DW results in a faster CG convergence. Moreover, there is an appreciable reduction of the jo-jo effect. Note also that the figures are truncated with regard to the CG tails for the original DW, as these continue even further beyond the presented amount of iterations, before finally terminating.

Hence, with regard to CG for the ORP with subproblems constructed over modules, weighted DW is superior, and will be the method used in the BAB/BAP implementation from this point on.

A note on CG cycling

In the end of Section 3.1 we briefly mentioned CG cycling, a problem possibly appearing for LP problems with degenerate optimal solutions (cf. cycling in the simplex method). When initially testing the basic CG method in the initial BAP framwork—CG applied to several different, although related, instances—it was apparent that for some instances the CG process reached the MP optimal value, but the process seemingly never terminated. At that point in the implementation, no measure had been implemented to deal with cycling. Subsequently, prior to the test described above, a simple anti-cycling measure was implemented, shortly described below.

Let $z^{\text{LP}}_{\text{RMP}(i)}$ denote the RMP objective value after the i:th CG iteration, and define the running mean of the RMP the objective values—at any time during the CG process—over the K latest CG iterations as $z^{\text{LP}}_{\text{RMP},mean(K)} = \frac{1}{K}\sum_{i=N-K+1}^{N} z^{\text{LP}}_{\text{RMP}(i)}$, where N denotes the total number of CG iterations at this point, and $K \in \{1,\ldots,N\}$ is a constant. Now, break the CG process when the *relative improvement* $\hat{\alpha}_{\text{CG}}$ of the RMP objective function value for the K latest CG iterations is less than a constant α_{CG}; the relative improvement $\hat{\alpha}_{\text{CG}}$ is defined as

$$\hat{\alpha}_{\text{CG}} := \frac{\max\limits_{i=N-K+1,\ldots,N}\left\{z^{\text{LP}}_{\text{RMP}(i)}\right\} - \min\limits_{i=N-K+1,\ldots,N}\left\{z^{\text{LP}}_{\text{RMP}(i)}\right\}}{z^{\text{LP}}_{\text{RMP},mean(K)}}, \tag{7.4}$$

where, in the BAP implementation, the constants above are set to $\alpha_{\text{CG}} := 10^{-4}$ and $K := 75$. After each solution of the RMP during the CG process—given that at least K CG iterations have been performed—the criterion $\hat{\alpha}_{\text{CG}} \overset{?}{<} \alpha_{\text{CG}}$ is controlled, and, if true, the current CG process is terminated.

The anti-cycling measure described above worked well in practice, and were incorporated in the basic CG method as well as in the stabilized CG method, prior to the testing included in this section. More rigourous methods are, however, described in the literature, and replacing the measure above by such measures should be considered as future work; this is also mentioned in Chapter 8.

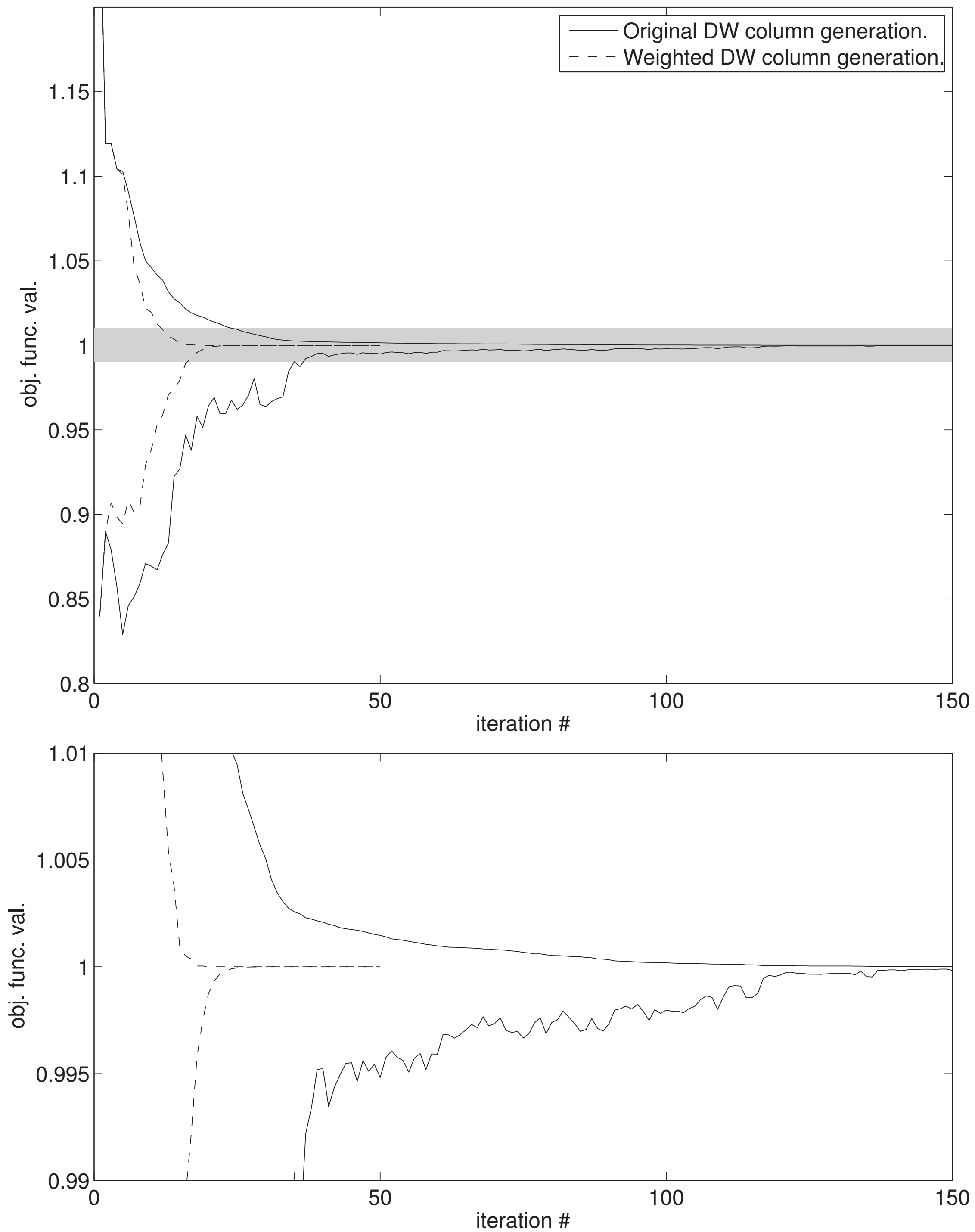


Figure 7.1: Convergence of upper and lower bounds on z_{MP}, using original (solid) and weighted (dashed) DW column generation, for test instance A. The lower diagram shows a close-up of the grey region in the upper one.

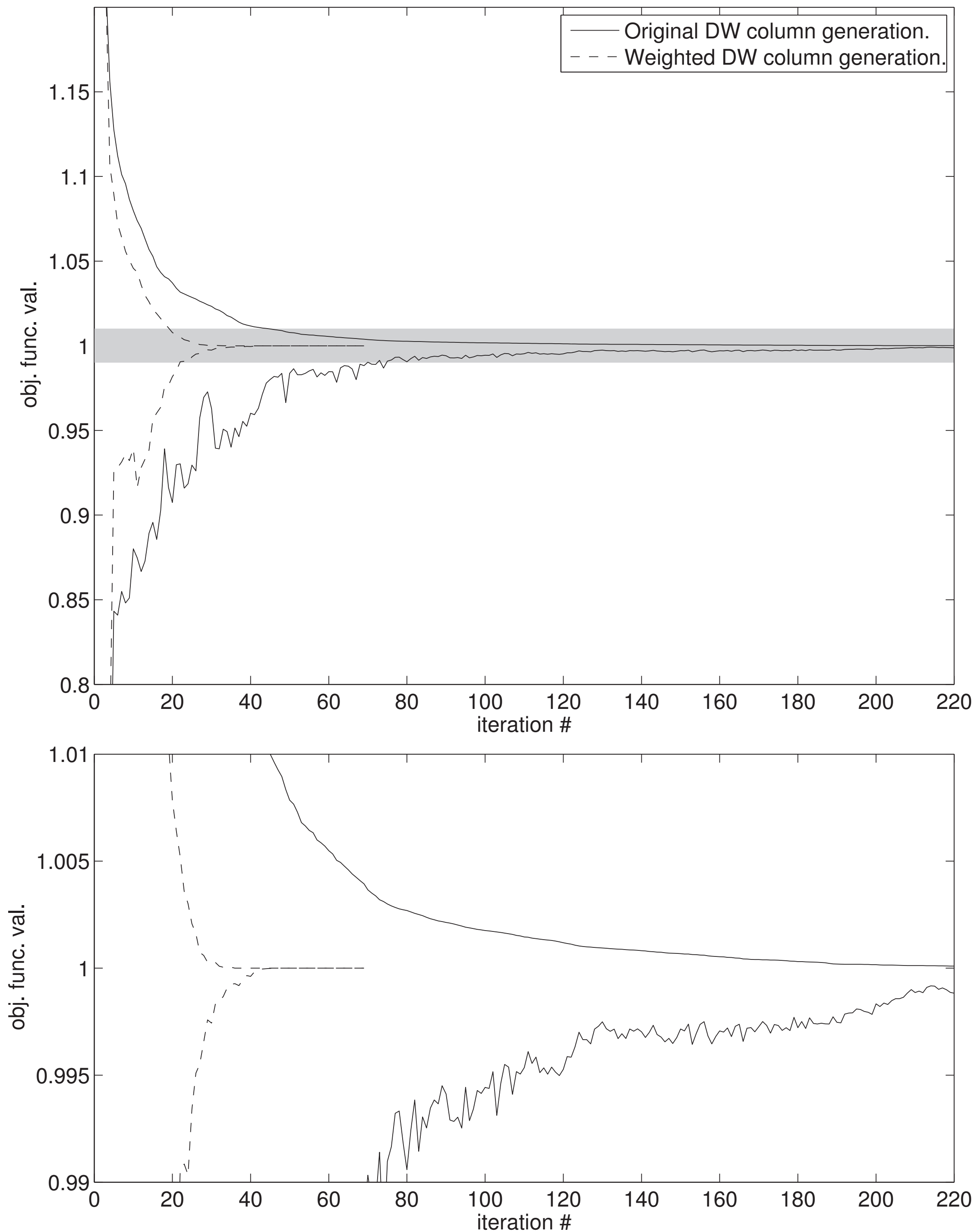


Figure 7.2: Convergence of upper and lower bounds on z_{MP}, using original (solid) and weighted (dashed) DW column generation, for test instance B. The lower diagram shows a close-up of the grey region in the upper one.

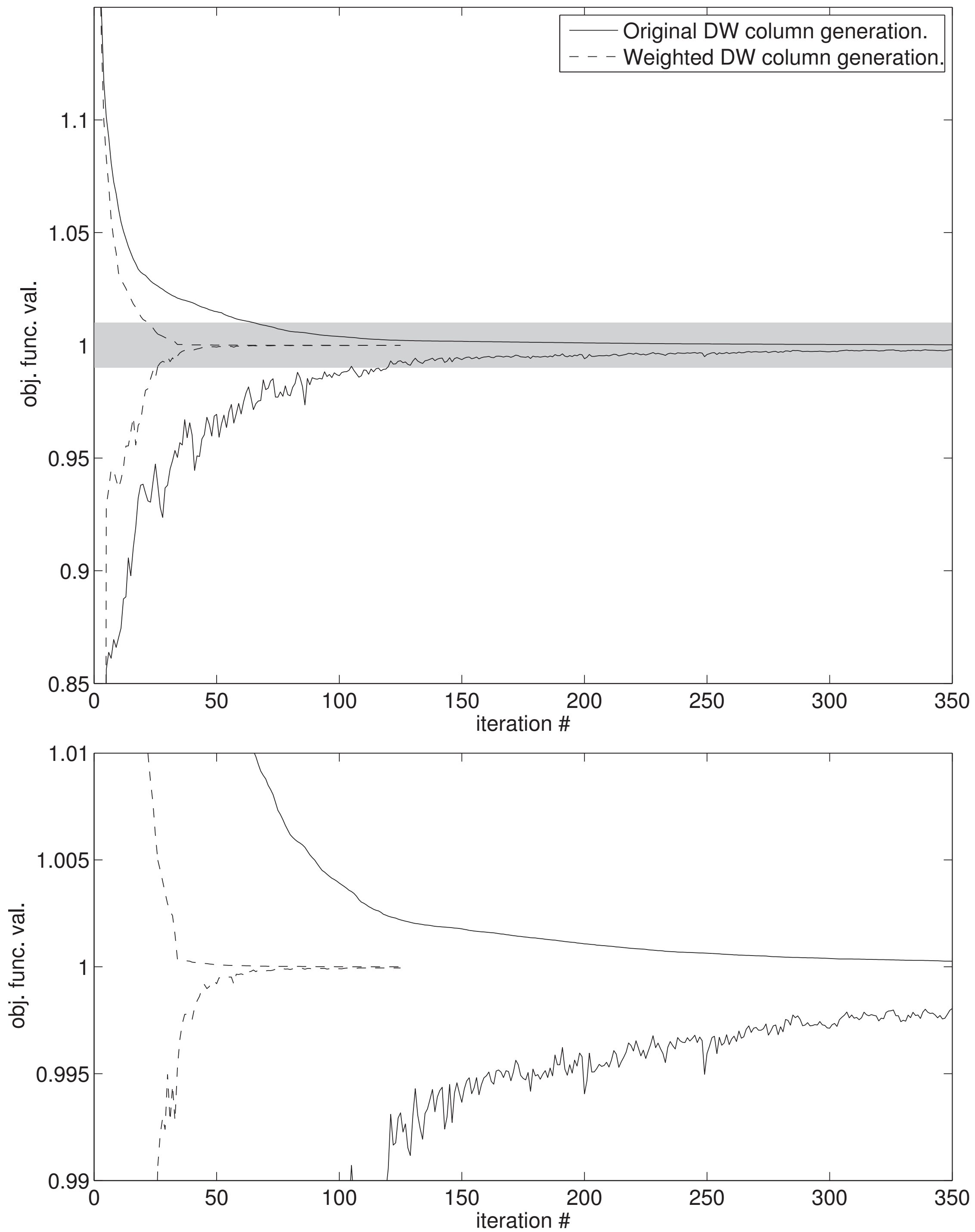


Figure 7.3: Convergence of upper and lower bounds on z_{MP}, using original (solid) and weighted (dashed) DW column generation, for test instance C. The lower diagram shows a close-up of the grey region in the upper one.

7.3 Second Modification: How to Choose Nodes from the Node Tree.

As we are yet to choose LP solver procedure, the choice for the purpose of the testing in this section will be based on the discussion in the end of Section 2.4; the dual simplex method will be used for the column generation as well as for the branching procedure. We will try three different node picking rules, based on two basic node picking strategies. The strategies are described below ([9]).

- *Depth-First (DF) strategy:* In the process of pruning nodes in the BAB tree, it's important to have a good upper bound (in case of a minimization problem), i.e., a good integer feasible solution. This can be achieved by quickly going deep in the node tree, as depth in the tree directly corresponds to the number of up or down cuts appended to the original RMP, and logically, the more cuts in a node, the higher the probability that the associated continuously relaxed RMP solution is integer valued. This strategy was described as our basic node picking rule in Section 6.1, and will be tested here aginst two alternate rules. The DF strategy is generally a good choice initially in the BAB process, with the purpose of finding a reasonably good upper bound fast. The stacking of nodes will be as depicted in Figure 6.1(b), hence focusing the DF strategy on down cut branches.

- *Best-Node First (BNF) strategy:* Each additional node that is processed directly affects the computational cost of the full algorithmic procedure, so naturally we want to avoid visiting nodes in the node list which, in retrospect, should never even have been considered. Using the BNF strategy, we always choose the node with the best (i.e., lowest lower) bound. By doing so, we guarantee never to branch upon a node possessing a lower bound that is larger than the integer optimal objective function value. Another advantage of this strategy is that we will always strive to increase the *tree lower bound*, i.e., the lower bound over all the remaining nodes in the node list. Whenever this tree lower bound becomes equal to or larger than the upper bound—provided by the best incumbent solution—we can terminate the BAB procedure, without actually processing the nodes remaining in the node list (all of which would be pruned by bound, if processed).

We use the two strategies above to create three different node picking rules in our BAB/BAP implementation. The three rules will thereafter be tested within a BAB execution on the three test instances, with rating criteria being *total number of nodes processed/visited*, *number of nodes visited until the upper/lower bound gap is less than* 5%, and *execution time for the BAB procedure*. The three node picking rules to be tested are as follows.

- P_1 : Uses the DF strategy throughout the full BAB procedure, i.e., it always picks the node that is on top of the node list, where the stacking in the node list is constructed such that the top node directly after branching of a node is a down cut.

- P_2 : Uses the BNF strategy throughout the full BAB procedure, i.e., always picks the node from the node list that has the lowest lower bound.

- P_3 : The article [39] mentions that a good node picking strategy should try to combine the DF and the BNF strategies, and shortly mentions *diving methods* and *two phase methods*. A basic diving method performs the DF strategy until a node is pruned specifically due to integrality (i.e., feasibility in the original problem with binary or integer variables) and thereafter backtracks in the tree for a good prospective node to re-dive from, e.g., the node in the tree with the lowest lower bound. It thereafter repeats the DF from this node until next node is pruned specifically due to integrality, iteratively diving from different locations of the tree, each time as deep as needed to find an integer valued solution. The idea of the two-phase methods is to start with DF to find one or a few initial integer feasible solutions, and thereafter to use BNF in an attempt to prove the possible optimality of the best incumbent solution, or at least to more quickly increase the lowest lower bound of all remaining nodes in the node list.

  As the third node picking rule for our testing, we will use the idea of the two-phase method, starting with a diving method until the LB/UB gap is *sufficiently small*, thereafter entering the second phase of trying to close the gap and prove optimality. To specify what we mean by sufficiently small, we define a constant $\alpha = 1.03$, and let the second phase begin when the ratio between the upper bound and tree lower bound is less than α. The third rule is as follows

  1. Use the DF strategy until an integer valued solution is found.
  2. Pick the node with the lowest lower bound in the node list.
  3. Solve the LP corresponding to this node. Is $\bar{z}^{\text{RMP}}/\underline{z}^{\text{tree}} < \alpha$? If yes, go to 6, otherwise proceed to 4.
  4. Is the node pruned? If yes, go to 2, otherwise, proceed to 5.
  5. Use the DF strategy to pick a node. Go to 3.
  6. Use the BNF strategy until BAB/BAP terminates.

  To summarize, the node picking rule finds an initial integer solution using the DF strategy, thereafter it uses the BNF strategy to possibly move away from the

active branch, after which the DF is used as long as active nodes are branched. If any node is pruned, we switch to the BNF strategy for just the next node. If, at any step, the ratio between the upper bound and the tree lower bound is less than α, permanently switch to the BNF strategy, which is used until termination. The somewhat small value of the constant α ($\alpha = 1.03$) means that the diving method will be favoured until we know, de facto, that the best incumbent (integer valued) solution is "quite" good. Only after this point will we try to prove its possible optimality, as well as focus on raising the tree lower bound $\underline{z}^{\text{tree}}$. If the pure BNF strategy is favoured early in the BAB/BAP process, say when the upper bound is still quite poor, there is a risk that the strategy—rather than raising the tree lower bound—mostly expands the tree by branching it at various locations, whereas pruning of branches, by bound, is less likely.

Results

Table 7.2 shows the result of BAB applied to the test instances A, B, and C—specifically, their RMP representations, cf. RMP (5.11) as derived from (5.3)—,using the node picking rules P_1, P_2 and P_3. Prior to applying the BAB algorithm to each instance, CG was used to initialize the column pool for the RMP of each instance, and the goal of the subsequent BAB appliance hence boiled down to finding the *best* column among those available in the initialized pool. In this context, the best column refers to the column, among those available, which is feasible[3] and corresponds to the lowest objective value in the MBLP RMP (5.11). For the remainder of this section, we will (non-rigorously) refer to this best column—given the available initialized columns—as the optimal solution for each test. The results displayed in Table 7.2 have been normalized with the objective function value of the best available column for each problem. The fourth column displays the number of nodes visited before BAB termination (proving *'optimality'* of the best column), and within parantheses, the number of nodes visited before identifying the best column.

The fourth column displays the number of nodes visited before BAB termination (proving *'optimality'* of the best column), and within parantheses, the number of nodes visited before identifying the best column

To graphically depict the results, Figures 7.4–7.6 display the BAB trees generated for each of the node picking rules and the problem instances A, B, and C, respectively. In the trees, black nodes have been processed and branched upon, green nodes

[3]Recall from paragraph *Initialization* in Section 6.2 that there will always be at least one column that is—by its heuristic construction—feasible w.r.t. the MBLP RMP (5.11); namely the **1**-column added to the RMP column pool at initialization.

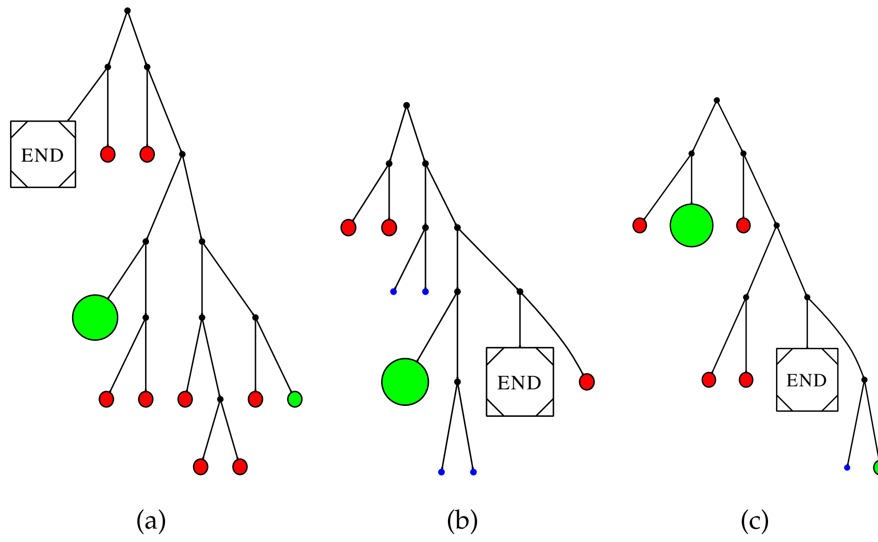


FIGURE 7.4: BAB tree plots for test instance A, using node picking rule (a) P_1, (b) P_2, and (c) P_3.

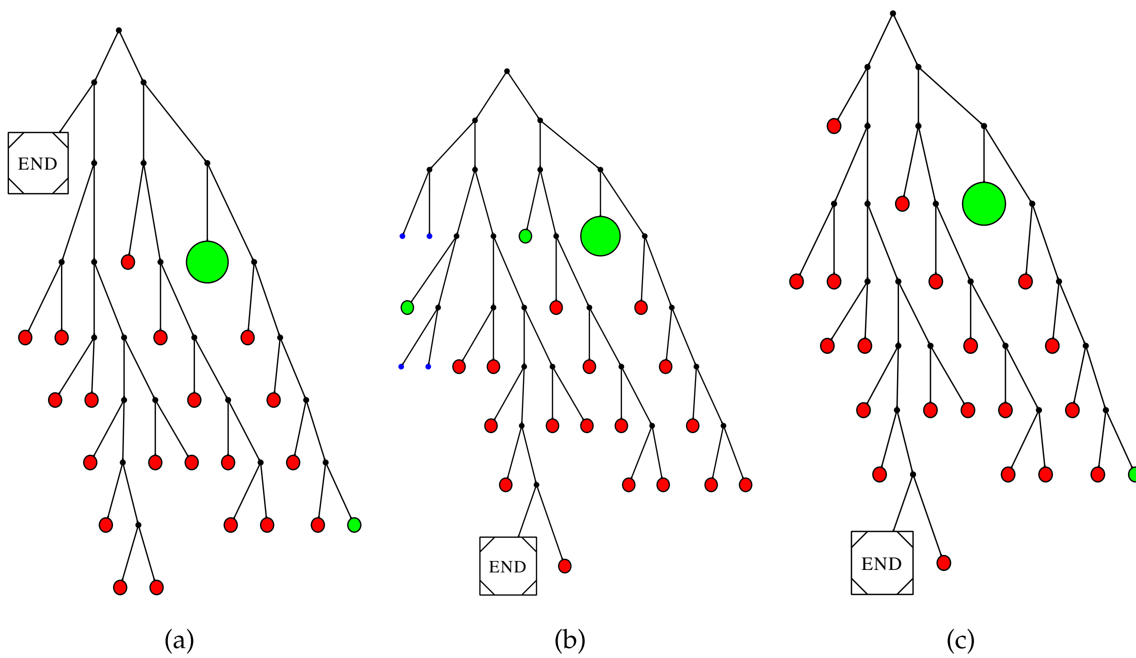


FIGURE 7.5: BAB tree plots for test instance B, using node picking rule (a) P_1, (b) P_2, and (c) P_3.

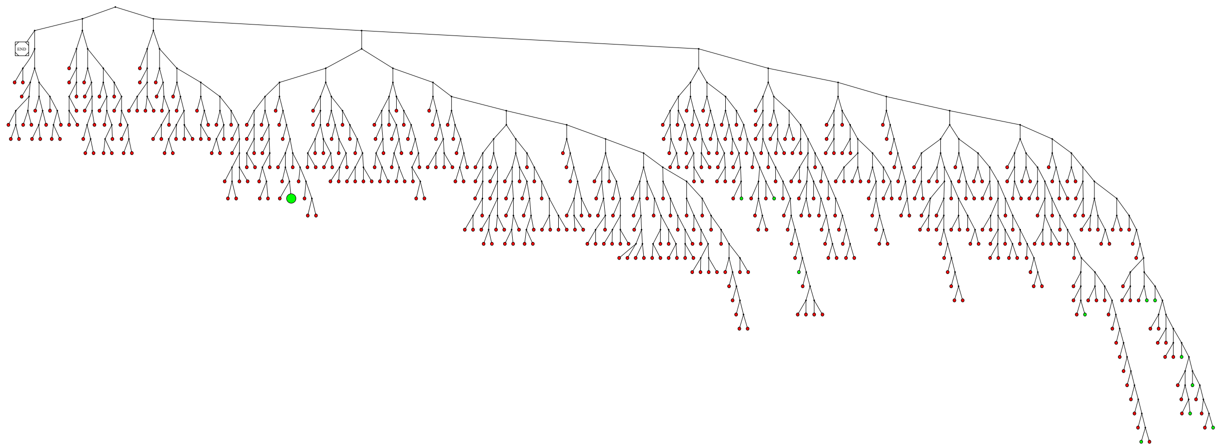
have been pruned due to optimality (large green node denotes optimality), red nodes pruned due to bound, and blue nodes have not been processed, i.e., they are implicitly pruned en masse due to bound at the termination of the BAB process. The end box displays the node after the processing of which the BAB process was terminated.

TABLE 7.2: BAB (after CG initialization of $N_{\text{Cols}}^{\text{CG}}$ columns) for the three different test problem instances A, B, and C, using the node picking rules *the DF strategy* P_1, *the BNF strategy* P_2 and the mixed rule P_3. The fourth column contains two parts of information; the number of nodes visited before BAB termination, and within parantheses, the number of nodes visited before identifying the best column.

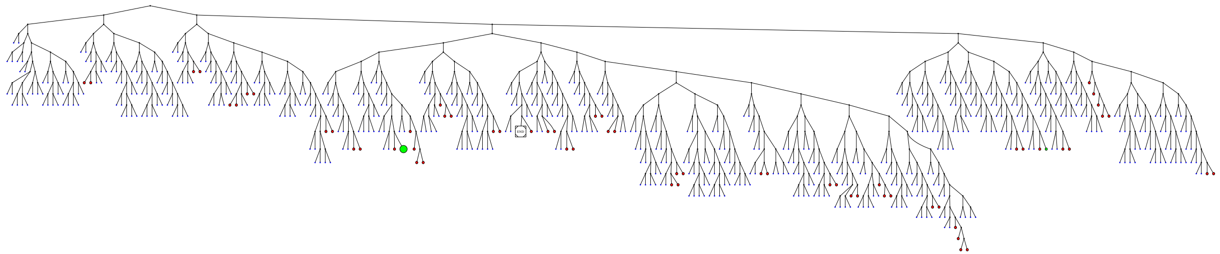
Instance	$N_{\text{Cols}}^{\text{CG}}$	Node picking rule	$N_{\text{Nodes}}^{\text{BAB}}$	$N_{\text{Nodes}}^{(\text{UB/LB}<1.05)}$	$t_{\text{CPU}}^{\text{BAB}}$ $[s]$
A	211	P_1	21 (17)	8	0.16
		P_2	13 (9)	9	0.09
		P_3	14 (8)	6	0.11
B	385	P_1	45 (13)	9	1.60
		P_2	45 (19)	17	1.37
		P_3	45 (28)	9	1.65
C	609	P_1	965 (784)	934	242.50
		P_2	657 (592)	584	193.10
		P_3	749 (718)	251	249.10

Looking at the first rating criterion, *total number of nodes processed/visited*, it becomes quite apparent that the pure DF rule P_1 looses out to the pure BNF rule P_2 and the mixed rule P_3, which is to be expected since P_1 is expected to visit nodes with lower bounds that are higher than the value of the optimal solution. Even without the actual numbers the tree plots show—most notable for the instance C (Figure 7.6)—that the pure DF rule P_1 processes and explicitly prunes (due to bound) many more nodes than does the pure BNF rule P_2; on the other hand, the latter rule yields a seemingly larger tree, in which the mass of nodes are, however, not processed, but instead implicitly pruned at the termination of the BAB algorithm. The rules P_2 and P_3 yield a similar amount of visited nodes, but the latter clearly beats the former with regard to the criterion *number of nodes visited until the gap between the upper and lower bounds is less than* 5%. We consider this criterion quite important with regard the actual BAP algorithm, as we want to close this gap as quickly as possible, before the column pool, that is common for all BAP subproblems (nodes), grows too large (in lack of a good column management).

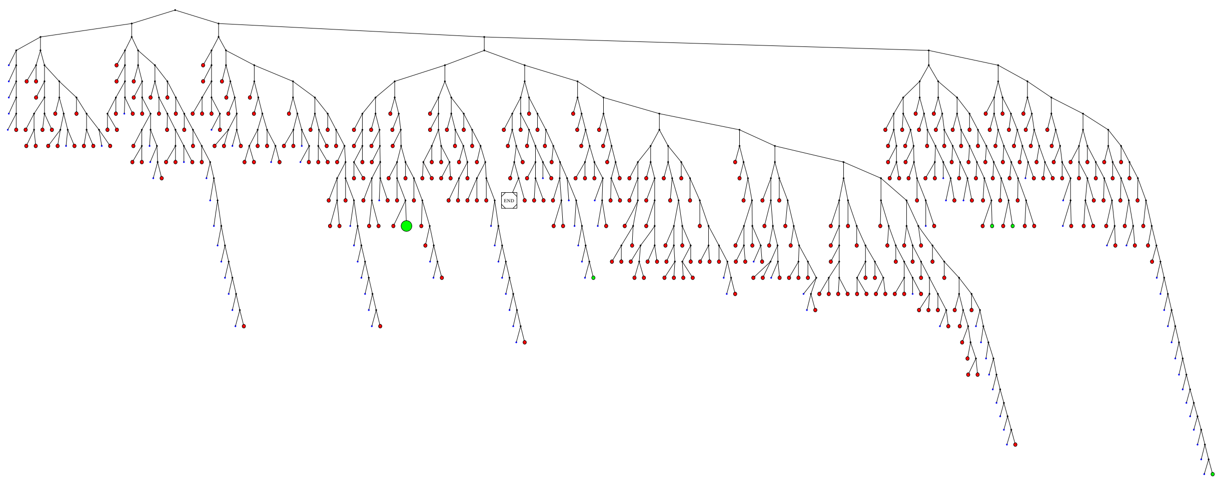
For the final criterion, *execution time for the BAB procedure*, it seems as if the pure



(a)



(b)



(c)

FIGURE 7.6: BAB tree plots for test instance C, using node picking rule (a) P_1, (b) P_2, and (c) P_3.

BNF rule beats its competitors. Looking closer at the results from the larger instance C, reveals interestingly that the total number of visited nodes for P_1 is quite much larger than for P_3, but still their execution times are similar. This is most likely explained by the fact that P_1 will allow for many *'warm-starts'* of the dual simplex method—as dual feasibility is maintained when a constraint is added to the primal—whereas P_3, on the other hand, is likely to move around in the tree inbetween the processing of nodes, and for each such jump, effectively guaranteeing that the LP solution process will be started from scratch for the following node[4].

In light of the discussion above, it seems as if the best choice of node picking rule is P_2 or P_3, and we choose, from this point on, to use the mixed P_3 rule in our BAP implementation. With regard to visited nodes, the mixed rule is seemingly quicker to reach a relatively low gap between the UB and LB; the rule is also a good starting point for future improvements, where it—as suggested—should take into account also the size of the node tree when deciding upon which phase to enter.

Finally notice that all three plots, regardless of the node picking rule, are heavily biased to the right. For the graphical depiction of the BAB tree, a down branch is depicted as a right branch in the tree, whereas up branches are branched left. For all the test instances, we expect quite sparse solutions for the z_m^t variables[5] which explains why the trees seems more heavy on their right sides. Down branches are *'cheaper'* than up branches[6], and for a sparse program, we can expect that most down branches maintain feasibility. If we were to consider a more dense program, with an optimal solution dominated by 1-valued variables—and hence, assuming that 1-valued variables impose costs that 0-valued variables do not, and meaning that sparse solutions would be infeasible—we could expect the tree to be heave to the left instead, as 0-branches would more frequently render infeasibility.

[4]Consider a dual optimal solution for an arbitrary processed node i during the BAB process, with associated LP $\mathcal{P}_{(i)}^{\text{LP}}$, and dual optimal variable values $\boldsymbol{\mu}_{(i)}^{\text{LP}}$. If the following node processed, say $i+1$, describes an LP problem, say $\mathcal{P}_{(i+1)}^{\text{LP}}$, which lacks additional constraints (cuts) that are present in $\mathcal{P}_{(i)}^{\text{LP}}$, then $\boldsymbol{\mu}_{(i)}^{\text{LP}}$ is not trivially transferred to a BFS in $\mathcal{P}_{(i+1)}^{\text{LP}}$ in the same natural fashion as for the case in which $\mathcal{P}_{(i+1)}^{\text{LP}}$ could be described as $\mathcal{P}_{(i)}^{\text{LP}}$ with additional cuts. In the way the CPLEX LP Solver has been used in the BAB/BAP implementaiton of this thesis, generally only the simple latter case will allow for starting the LP solver *'on-the-fly'*.

[5]See Table 7.1; except for $m=1$ and possibly $m=2$ (the variables z_t^1 and z_t^2, respectively), most of the variables z_t^m are expected to take the value 0 in an optimal solution.

[6]Cheaper with regard to the objective function value. For an up branch, we might impose a *demand of inclusion* to use a maintence occasion that could possibly turn out quite costly, whereas for the sparse program we're considering, we can expect the *exclusion* of a maintence occasion not to cause an as large increase of the objective value (due to degeneracy of near-optimal solutions for this sparse program; many maintanence schedules that are similar in setup and differ only slightly in price).

7.4 Third Modification: Choosing the Primal or Dual Simplex Method as the LP Solver

Our current implementation of the BAB/BAP algorithm now deals with the tailing-off effect using weighted Dantzig-Wolfe decomposition, and incoorporates a somewhat sophisticated node picking rule. We continue the tuning of the algorithm, now focusing on which LP solver to use, i.e., using either the primal or the dual simplex method. Our main rating criterias will be how the methods affect *the size of the node tree* and *the execution time for the branching procedure*. We will once again focus on the BAB algorithm, and assume that these two BAB criteria are inherited by the BAP algorithm.

We investigate four different setups, determined by combinations of two setup parameters, *LP solver for the initial CG* and *LP solver for the branching procedure*. The setups are specified in Table 7.3 below.

TABLE 7.3: Construction of the four LP solver setups D–D, D–P, P–D, and P–P.

	Method for the initial CG	
Method for the branching procedure	*Dual simplex*	*Primal simplex*
Dual simplex	D–D	P–D
Primal simplex	D–P	P–P

We perform the initial CG followed by BAB on the three test instances A, B, and C, using the setups defined above. More specifically, CG is used to generate good columns with regard to the LP MP, and thereafter the BAB algorithm is applied to find the best column—among those generated—with regard to the binary MP.

Finally note that we can expect the D–D and D–P setups to yield a different best objective value in the binary program than will the P–D and P–P setups, as the choice

of LP solver in the initial CG phase will affect which columns that are generated to the column pool; even if we can expect the primal and the dual simplex CG to reach the (same) LP optimal value, a non-basic column in their respective LP optimal solutions might turn out to be the best column in the corresponding binary program, and the non-basic columns are likely to differ between the two CG setups. Also, in case of degeneracy, the basic columns in the respective LP optimal solutions for the two CG setups may differ. As our final goal is to achieve a BAP and not a simple CG $\rightarrow$ BAB implementation, however, the best (given the available columns) objective function value of the binary MP is not considered to be relevant for this test, and will not be compared between the different setups.

Results

We denote the execution times for the CG phase and the subsequent BAB phase by $t_{\text{CPU}}^{\text{CG}}$ and $t_{\text{CPU}}^{\text{BAB}}$, respectively. Furthermore, $N_{\text{Cols}}^{\text{CG}}$ denotes the number of columns generated during the initial CG phase, and $N_{\text{Nodes}}^{\text{BAB}}$ the subsequent number of nodes visited until BAB termination.

TABLE 7.4: CG followed by BAB for the three test instances *A*, *B*, and *C*, using the four LP solver setups D–D, D–P, P–D, and P–P.

Instance	LP solver setup	$N_{\text{Cols}}^{\text{CG}}$	$t_{\text{CPU}}^{\text{CG}}$ $[s]$	$N_{\text{Nodes}}^{\text{BAB}}$	$t_{\text{CPU}}^{\text{BAB}}$ $[s]$
A	D–D	211	2.70	14	0.11
	D–P			14	0.27
	P–D	281	3.00	17	0.12
	P–P			19	0.34
B	D–D	385	11.00	45	1.56
	D–P			47	4.34
	P–D	378	10.30	27	0.71
	P–P			25	2.46
C	D–D	609	207.80	811	261.34
	D–P			739	460.43
	P–D	605	224.08	1091	376.83
	P–P			1129	675.09

Table 7.4 shows the results of the tests for the three test instances. With regard to the test of *LP solver for the initial column generation*, it shows only show a small variance—the number of generated colums and the execution time—with respect to solution

method. We focus instead on the criterion *LP solver for the branching procedure*.

For all three instances, irrespective of which solver is used during the initial CG, the dual simplex method is remarkably faster than the primal method during the execution of the BAB algorithm. In the last section of Chapter 2 we discussed how the dual simplex could be preferred over it's primal counterpart in cases where subproblems are generated by adding cuts to the original problem, and in the previous section of this chapter, we discussed how the BAB process is precisely such a case, in which the dual simplex method should allow for more *'on-the-fly'* starts of subsequent BAB subproblems.

Consequently, it is quite an apparent choice to use the dual simplex method for the BAB procedure, and as there seemingly is no major difference between the two solvers for the CG procedure, we choose to use the LP setup D–D, thereby avoiding a frequent change of LP solvers during the process of the BAP algorithm. We proceed to the final improvement test for the algorithm, implementing a sophisticated branching rule.

7.5 Fourth Modification: Introducing a Sophisticated Variable Branching Rule—Pseudocost Branching

So far, we have not put any real effort into choosing branching variables. In our current very simple branching rule, the first variable that from a node-local point of view might be prospective to branch down is chosen. Here, we define a variable to be *prospective* if its fractional value is in the span $(0, 0.5]$. This rule was deduced from a logical perspective—as is explained in Section 6.2—generally favouring exclusions (down cuts) of module maintenance occasions, specifically for maintenance occations in which the corresponding variable has a small fractional value in the optimal solution to the (LP) RMP in the node that is to be branched. In a sparse program most variables in an optimal solution will be 0 and hence branching down fractional variables should generally be more advantageous than branching them up, as the latter enforces inclusion of maintenance occasions, which is likely to be unfavourable in a sparse program. For our test problems, this simple branching rule does indeed outperform the strategies of random variable or *most fractional variable* picking.

As the fourth modification to our BAP implementation, we have, however, implemented an advanced branching rule, *pseudocost branching*, as presented in [21].

Pseudocost branching

Consider, for a BLP problem, an arbitrary processed and non-pruned subproblem during the BAP process, say $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, with optimal value $z^{\text{LP}}_{\text{RMP}}$, and corresponding variable values $\boldsymbol{x}^{\text{LP}}_{\text{RMP}}$, and assume that $\boldsymbol{x}^{\text{LP}}_{\text{RMP}} \notin \{0,1\}^n$. Let $I = \{1, \ldots, n\}$ be the set of indices for the decision variables $x_i \in \boldsymbol{x}^{\text{LP}}_{\text{RMP}}$, and let $K = \{i \in I \mid x_i \notin \{0,1\}\}$ define the set of branching candidates, i.e., fractional values x_i, one of which we want to branch upon. For each $i \in K$, let $\mathcal{P}^{\text{LP},+}_{\text{RMP}(x_i)}$ and $\mathcal{P}^{\text{LP},-}_{\text{RMP}(x_i)}$ denote the two branching subproblems to $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, given that we were to branch on x_i by adding an up and a down cut to $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, respectively. Also, for each $i \in K$, set $f_i^+ = 1 - x_i$ and $f_i^- = x_i$, and note that these measures describe how much x_i is increased and decreased, respectively, in the transition from the optimal solution of $\mathcal{P}^{\text{LP}}_{\text{RMP}}$ to any (possibly) feasible solution for its branching subproblems $\mathcal{P}^{\text{LP},+}_{\text{RMP}(x_i)}$ and $\mathcal{P}^{\text{LP},-}_{\text{RMP}(x_i)}$, respectively. Furthermore, assume that both branching subproblems are feasible, and let $z^{\text{LP},+}_{\text{RMP}(x_i)}$ and $z^{\text{LP},-}_{\text{RMP}(x_i)}$ denote the optimal value of $\mathcal{P}^{\text{LP},+}_{\text{RMP}(x_i)}$ and $\mathcal{P}^{\text{LP},-}_{\text{RMP}(x_i)}$, respectively, and define the changes in objective function value—with regard to $\mathcal{P}^{\text{LP}}_{\text{RMP}}$—for the two branching subproblems as $\Delta_i^+ = z^{\text{LP},+}_{\text{RMP}(x_i)} - z^{\text{LP}}_{\text{RMP}}$ and $\Delta_i^- = z^{\text{LP},-}_{\text{RMP}(x_i)} - z^{\text{LP}}_{\text{RMP}}$, respectively. Finally, let ζ_i^+ and ζ_i^- be measures of the change in the objective function value per unit change of the

variable x_i for the BAP subproblem (or, BAP node) $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, for up and down branching, respectively; i.e., $\zeta_i^+ = \Delta_i^+ / f_i^+$ and $\zeta_i^- = \Delta_i^- / f_i^-$. With these measures, we can quantify the impact of the branching on a specific variable on the optimal objective value of its branching subproblem, as compared to the problem being branched. A branching that yields two subproblems which both notably raises the lower bound of the corresponding branch (w.r.t. the branched node) can be considered a successful branching, as we want the global lower bound (or, tree lower bound) to steadily increase as the nodes are processed, in contrast to an example of a poor branching scenario, in which the branching choices would only expand the tree without any real progress (raise) in the global lower bound. This discussion reveals that ζ_i^+ and ζ_i^- measures the success of the branching of a specific variable in a specific node. The idea of using *pseudocosts* is to keep a history of these measures, to get an idea of the general success of different branching variables with regard to arbitrary positions in the tree. With these pre-requisites, we are ready to define the pseudocosts of a variable.

Let σ_i^+ denote the sum of ζ_i^+ over all node problems $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, in which x_i has been chosen as branching variable and for which a feasible solution to the resulting branching subproblem $\mathcal{P}^{\text{LP},+}_{\text{RMP}(x_i)}$ has been found, and let η_i^+ be the number of such node problems. Let σ_i^- and η_i^- be the analogous measures for down cuts. *The pseudocosts* for the variable x_i are then defined as

$$\Psi_i^+ = \frac{\sigma_i^+}{\eta_i^+}, \tag{7.5a}$$

$$\Psi_i^- = \frac{\sigma_i^-}{\eta_i^-}, \tag{7.5b}$$

for up and down branching, respectively. A score function is used to map the up and down pseudocosts to a single value, according to the definition

$$\text{score}(q^-, q^+) := (1-\mu) \cdot \max\{q^-, q^+\} + \mu \cdot \min\{q^-, q^+\}, \tag{7.6}$$

with $\mu \in [0,1]$. The *pseudocost score* for each variable x_i—the essential component in pseudocost branching—is subsequently defined as

$$s_i := \text{score}(f_i^+ \Psi_i^+, f_i^- \Psi_i^-). \tag{7.7}$$

The idea of *pseudocost branching* is to consistently—for each BAP node that is to be branched—choosing the branching variable x_i among the set of branching candidates $i \in K$ with the highest pseudocost score, i.e., x_i for $i \in \arg\max_{i \in K}\{s_i\}$. We chose the value of $\mu := 1/6$, which means that high pseudocosts are favoured. This choice also yields stability, in the sense that the pseudocosts are good for both up and

down branching. The increase in the global lower bound for an arbitrary branching—assuming that the two branching subproblems are solved instantly—is equal to the smallest increase of the two branching *'children'*, so a branching variable which yields one *'great'* node and one *'poor'* node, is probably a worse branching variable choice than one that yields two *'decent'* nodes. With this setup, we strive to always choose to branch on the variable which is expected to lead to the largest increase in the lower bound of the tree.

Now, in the beginning of the BAP algorithm, there is no data ($\eta_i^+ := \eta_i^- := 0$ for all $i \in I$) to compute any pseudocosts, and all pseudocosts are *uninitialized*, in which case the expressions for the pseudocosts in (7.5) are undefined. Equivalently, we refer to pseudocosts of a variable to which data are available in both cut directions, i.e., the case $\{\eta_i^+ > 0, \eta_i^- > 0\}$ as *initialized*, or in case of data being available only in one cut direction, *initialized in up* or *initialized in down cut direction* for the cases $\{\eta_i^+ > 0, \eta_i^- = 0\}$ and $\{\eta_i^+ = 0, \eta_i^- > 0\}$, respectively. Let $\widetilde{I}^+ = \{i \in I \mid \eta_i^+ > 0\}$ define the set of up cut pseudocosts for which data exists, and define the average of all initialized up cut pseudocosts as

$$\Psi_{\text{Avg}}^+ = \frac{1}{|\widetilde{I}^+|} \sum_{i \in \widetilde{I}^+} \Psi_i^+. \tag{7.8}$$

With this, we can state a more general expression for the upwards pseudocosts as

$$\Psi_i^+ = \begin{cases} 1, & \text{if} \quad \sum_{j \in I} \eta_j^+ = 0, \\ \Psi_{\text{Avg}}^+, & \text{if} \quad \eta_i^+ = 0 \quad \text{and} \quad \sum_{j \in I} \eta_j^+ > 0, \\ \frac{\sigma_i^+}{\eta_i^+}, & \text{if} \quad \eta_i^+ > 0. \end{cases} \tag{7.9}$$

To summarize this expression; if all up cut pseudocosts are uninitialized, then they will all be given the value 1, whereas if some up cut pseudocosts are initialized, then all the uninitialized up cut pseudocosts are given the avarage value of the initialized up cut pseudocosts, until these up cut pseudocosts attain data to become initialized themselves. Down cut pseudocosts, Ψ_i^-, are analogously defined. Finally, if a branching subproblem is infeasible, the associated pseudocost (up or down) ignores the information from that problem, and is not updated.

Using purely this method, however, will mean that there is very little confidence in the pseudocosts in the early stages of the BAP process, and that there will be a satisfactory large set of pseudocosts only after sufficiently many nodes has been processed. We've previously mentioned the importance of good branching in the early stage of the BAP process, so this is quite an inherent flaw, which will be cured in the following paragraph.

Strong branching initialization

To remedy the issue of initially weak pseudocost information, we will combine the pseudocost branching method with *strong branching initialization* [21]. Again consider an arbitrary node in the BAP process, say $\mathcal{P}^{\text{LP}}_{\text{RMP}}$, and assume it is feasible with the optimal value $z^{\text{LP}}_{\text{RMP}}$, and the corresponding variable values $\boldsymbol{x}^{\text{LP}}_{\text{RMP}}$. Furthermore, assume that $\boldsymbol{x}^{\text{LP}}_{\text{RMP}} \notin \{0,1\}^n$, and let $K_0^+ := \{i \in K \mid \eta_i^+ = 0\}$ and $K_0^- := \{i \in K \mid \eta_i^- = 0\}$ be the subsets of branching candidates for which the up and down cut pseudocosts, respectively, are uninitialized. The idea of strong branching initialization is to—prior to choosing branching variables from K using available pseudocost scores—*try* to initialize the up and down cut pseudocosts for all upwards and downwards uninitialized branching candidates, $i \in K_0^+$ and $i \in K_0^-$, respectively. It emphasizes on *try* with regard to the feasibility of the up and down branching subproblems $\mathcal{P}^{\text{LP},+}_{\text{RMP}(x_i)}$ and $\mathcal{P}^{\text{LP},-}_{\text{RMP}(x_i)}$, respectively, of $\mathcal{P}^{\text{LP}}_{\text{RMP}}$.

For the root node in the BAP process, all branching candidates $i \in K$ will be members of the strong branching initialization subsets K_0^+ and K_0^-. Hence, for the root node, the pseudocosts for all candidates $i \in K$ will be initialized in both the up as well as down directions, given that the associated branching subproblems are feasible. As the BAP algorithm progresses, the subsets K_0^+ and K_0^- will generally decrease, and the participation of the strong branching initialization will subside as the number of initialized pseudocosts naturally increases.

Algorithm3 summarizes the pseudocost branching with strong branching initialization, employing the notation used in the foregoing paragraphs of this section. Note that for our specific BLP BAP implementation—aimed for solving instances of the opportunistic maintenance scheduling problem—the branching candidates are always chosen as the fractional variables among the the module maintenance variables z_m^t ($t \in \mathcal{T}$, $m \in \mathcal{M}$) of the original problem (5.3), as is described for the basic branching rule in Section 6.2.

A note on the reliability of the pseudocosts

Before moving on, we note, however, that even with the introduction of the strong branching initialization, the pseudocosts are not necessarily generally representative for the associated branching variables' values in the BAP tree, as some might be based only on a single ζ_i^+ or ζ_i^- calculation. It is possible that after some growth in an arbitrary BAP tree, many pseudocosts could become *unreliable*, in the sense that they *were* measured to be good or bad branching choices in a previously processed part of the BAP tree—say a tree branch that was later fully pruned en masse—whereas, however,

Algorithm 3 Pseudocost branching with strong branching initialization

```
1:  procedure PSEUDOCOST-BRANCHING(K)        ▷ INPUT: Set of branching cand. K ≠ ∅
2:      for each ▽ ∈ {+, −} do                  ▷ Strong branching initialization
3:          K_0^▽ = {i ∈ K | η_i^▽ = 0}
4:          Ĩ^▽ = {i ∈ I | η_i^▽ > 0}
5:          for each i ∈ K_0^▽ do
6:              f_i^▽ ← { 1−x_i,  if ▽=+ ;  x_i,  if ▽=− }
7:              if Δ_i^▽ ← solve(P^{LP,▽}_{RMP(x_i)}) then   ▷ If feasible: initialize pseudocost
8:                  Ψ_i^▽ ← σ_i^▽ ← ζ_i^▽ ← Δ_i^▽ / f_i^▽
9:                  η_i^▽ ← 1
10:             else
11:                 ignore(P^{LP,▽}_{RMP(x_i)})     ▷ If infeasible: keep uninitialized value
12:             end if
13:         end for each
14:         if Ĩ ≠ ∅ then
15:             Ψ_Avg^▽ ← 1/|Ĩ^▽| Σ_{i∈Ĩ^▽} Ψ_i^▽       ▷ Update average pseudocost
16:         else
17:             Ψ_Avg^▽ ← 1
18:         end if
19:     end for each
20:     for each i ∈ K do                       ▷ Calculate pseudocost scores
21:         f_i^+ ← 1 − x_i
22:         f_i^− ← x_i
23:         s_i = score(f_i^+ Ψ_i^+, f_i^− Ψ_i^−)
24:     end for each
25:     ĩ ← arg max_{i∈K} {s_i}
26:     return (ĩ)                ▷ Return branching candidate with highest score
27: end procedure
```

that measure is no longer accurate for some other part of the BAP tree being processed at a later stage. In that stage it might possibly process a branch or part of the tree that, with the appended cuts, describe a part of what might later turn out to be an integer optimal solution, given that good branching variables are chosen when further expanding that branch; i.e., given that we can rely on the pseudocosts. One way to raise the reliability of the pseudocosts would be to perform strong branching for all branching candidates at every node, even if their pseudocosts are already initialized. This, however, is expected to be very time-consuming; it amounts to fully solve a large number of LP:s—all of the same complexity as one can expect from a random node in the BAP tree—however, without actually expanding the tree into possibly fruitful directions. The pseudocosts provide, however, not in any way an exact measure of the general *'branching success value'* of a branching variable. We realize that it is possible to dynamically update the pseudocosts without actually solving all the *'branching children'* to LP optimality. This is the general idea of one of current the top-tier branching rules, the sophisticated *reliability branching rule* [21], an algorithm that is based on the pseudocost branching rule, but which introduces also a reliability measure for the pseudocosts, which relates to η_i^+ and η_i^-. To summarize shortly, for eaching branching choice, the algorithm constructs *'look-ahead'* LP:s for branching candidates that are deemed to be unrealiable, but does not solve these to optimality. Instead it performs a limited number of simplex iterations, in so updating the reliability measure for such branching candidates. The process is repeated until all pseudocosts for all branching candidates at that node are demeed sufficiently reliable, after which the candidate with the highest pseudocost score is chosen and processed for branching. It is however out of the scope of this thesis to further develop the branching rule implementation, but it should be noted as an interesting area of further study. See [21] for details, as well as [40] covering *hybrid branching*, a further developed variant of the reliability branching rule which also combines variable selection rules for fields other than MILP, and hence generalizes its usage.

Now, as the introduction of pseudocost branching marks the final modification of the algorithm, we save the results for the last section of this chapter, where we compare the performance of the basic branching BAP implementation, the pseudocost branching BAP implementatation, and the commercial CPLEX MIP solver [3].

7.6 Results: Comparing Final Setups with CPLEX MIP Solver

We compare our last two BAP implementations with CPLEX MIP Solver, v12.1 [3]. For these BAP implementations, a time limit of one hour is set for the BAP process, excluding the initialization step, which is here defined as generating columns and solving the root node LP, and—in case of pseudocost branching—generating pseudocosts for all branching candidates of this root LP. In the comparison below, naturally also the execution time for the initialization step is listed, however separated from the remaining execution time (as it is interesting to compare this value between the two BAP implementations). The algorithms are tested, as before, applied to the test instances A, B, and C of (5.3), where naturally the BAP implementations make use of the problem in its decomposed form—the RMP (5.11) with associated subproblems 5.12—whereas CPLEX is applied directly upon the original problem (5.3).

Now, denote the two final BAP algorithms implemented in this thesis as follows:

BAP(Basic branching rule): Branch-and-price algorithm including all but the last modifications described in this chapter, i.e., all modifications except for the final branching rule implementation in Section 7.5.

BAP(Pseudocost branching rule): Branch-and-price algorithm including all modifications described in this chapter, including the pseudocost branching with strong branching initialization branching rule.

Table 7.5 shows the results of the comparison. For the two BAP implementations, $\bar{z}_{MBLP}$ denotes the objective value of the best incumbent solution after termination of the algorithm; in line with the theory covered in Chapter 4 and as summarized in Figure 4.4, we expect this value to equal the optimal value, unless the algorithm has been pre-maturely terminated due to the time limit, which is then explicitly noted. Note that the objective values have been normalized with the optimal objective value for each respective instance. N_{Nodes} denotes the number of nodes visited at termination (convergered or pre-mature termination); within parantheses in the same column: the number of nodes visited until reaching the best incumbent solution. Finally, $t_{\text{CPU}}^{\text{Init}}$ and $t_{\text{CPU}}^{\text{BAP}}$ denote the execution time (CPU time) for the initialization phase and BAP phase, respectively, for the two alogrithms.

If we start by looking at the results from the two BAP implementations, it is quite apparent that the introduction of the pseudocost branching rule greatly improves the algorithm. The initialization phase for the pseudocost BAP naturally runs a bit longer than for the basic rule equivalance, but for the remaining BAP phase, the sophisticated rule is clearly superior. It is also noteworthy that pseudocost BAP seemingly

Table 7.5: Performance of the two final BAP implementations, using the basic branching rule and the sophisticated pseudocost branching rule, respectively, and the commercial CPLEX MIP Solver, for the three test instances A, B, and C.

	BAP(Basic branching rule)			
	$\bar{z}_{\text{MBLP}}$	N_{Nodes}	$t_{\text{CPU}}^{\text{Init}}$ $[s]$	$t_{\text{CPU}}^{\text{BAP}}$ $[s]$
A	1	31 (11)	2.70	34.23
B	1.01003[i]	51 (24)	10.90	173.97
C	–[ii]	46 (-)	207.80	3600.00
	BAP(Pseudocost branching rule)			
	$\bar{z}_{\text{MBLP}}$	N_{Nodes}	$t_{\text{CPU}}^{\text{Init}}$ $[s]$	$t_{\text{CPU}}^{\text{BAP}}$ $[s]$
A	1	8 (8)	4.16	4.58
B	1	5 (4)	19.48	14.53
C	1.0046[ii]	132 (89)	460.00	3600.00
	CPLEX MIP Solver			
	$\bar{z}^*_{MBLP}$	N_{Nodes}	t_{CPU} $[sec]$	
A	1	3 (1)	0.51	
B	1	3 (1)	1.42	
C	1	226 (7)	150.35	

[i] Possibly the home-brewed CG cycling solution terminates a CG process pre-maturely, due to cycling at a near-optimal solution.
[ii] Terminated pre-maturely due to $1h$ time limit for the BAP process.

proves optimality quite quick—compare the number of nodes visited until termination with the number of nodes visited until finding the last incumbent solution—with regard to the basic rule; this is to be expected, since one of the main purposes of the sophisticated branching rule is to steadily increase the tree lower bound.

We note also that the basic rule fails in even finding a single best incumbent solution (we do not list the very poor initial heuristically constructed solution) within the time limit for the largest problem instance C, and we can somewhat relate this to the much lower number of visited nodes for the basic BAP vs. the pseudocost BAP, for the instance C. This was, however, not expected, since the number of visited nodes—within an execution time limit—mostly relies on the speed in which the associated LP (RMP) for each node is solved. This in turn, for a BAP framework, is equivalent to the CG run time for solving the RMP associated with each node, which we expected to be similar for the two implementations. A closer look on the detailed results (not presented here) for test instance C, however, shows a huge difference in the mean CG execution time for the basic BAP vs. the pseudocost BAP; the nodes with the longest CG execution times showed similar time values for the two different algorithms, but the basic BAP run lacked the large number of quickly solved nodes that was seen in the pseudocost BAP run. We can speculate that with the pseudocost branching rule, the branching sub-nodes would describe LP:s with somewhat largely different optimal objective function values with regard to the parent node (the goal of pseudocost branching), which means that also the *'CG path'* should differ somewhat substantially from that of the parent node. For the basic rule, however, we could very well find—due to degeneracy—two branching subproblems with optimal objective values identical or very close to their parent node LP value, however excluding a key column in the LP solution of the parent node. This means that small fluctuations in the CG reduced costs would still have a large effect on the dual variables (i.e., an unreliable dual stabilization), which directly relates to which columns are to be generated. Hence, we could end up in scenarios which—for the CG solving of a branching subproblem LP—initially directly *'lands in'* the tailing-off region of the CG generation for that node, but without the valuable information about the dual variables that should've been attained in the early phases of the CG; this is essential for the reliability of the weighted dual variables in the weighted CG method. This could possibly explain why the LP of many nodes in the basic BAP implementation seemingly takes an unproportionately long time to solve.

With regard to the BAP implementation's performance compared to the CPLEX MIP Solver, it is, as expected, obvious that the commercial software is superior our BAP implementation. We can assume that CPLEX not only uses a more sophisticated BAP framework—the branching rule at least comparable to reliability branching [40],

assumingly also a far more advanced node picking rule, clever initial as well as on-the-fly heuristics, and so on—but also includes several pre-processing measures to simplify a MILP prior to entering the process of actually solving it ([3]), and moreover, can be assumed to contain flawless implementations of the different solver algorithms. The proportional (relative) difference in solution times seemingly increases with the size of the test instance, and where the instance C would be considered as a *'hard'* problem for our BAP implementation (not solved within one hour time limit), CPLEX solves the problem in less than three minutes. It is quite interesting, however, to see how CPLEX processes hundreds of BAP nodes within minutes, whereas our own implementation needs hours to do the same. As the most noteworthy time consumer in our current implementation is the solving of the LP RMP:s for each node, we can assume that a lot can be done in the pre-processing as well as solving of this LP, not forgetting possible improvements of the solution of the pricing subproblems. Comparing the number of processed nodes between the two, the performance gap is seemingly not as huge as that of the execution times. For the instances A and B, CPLEX finds the optimal solution after a single node, and spends the remaining node processing to prove optimality; possibly CPLEX finds these instances simple enough to heuristically construct the optimal solution directly, hence using BAP only to raise the lower bound.

To summarize, it seems as if the *'backbone'* of our implementation—node processing and the growth of the BAP node tree—seems to work quite well, whereas the solving of the node LP:s via CG could do with some improvement, foremost good column generation and column management, as will be discussed in the next chapter.

Finally, we graphically present some details of the results of the pseudocost BAP implementation applied to the three test problem instances in Figures 7.7–7.9. Note that for the instance C, the memory usage at termination of the algorithm is limited, and the growth of the active node list seemingly starts to drop off, so one can argue that this BAP implementation can be expected to possibly be able to solve quite large instances of (5.3), given that the solving of the node LP:s is improved (improve CG, heuristics, and *column management*), as the current limitations of the implementation are mainly due to available execution time, and not computer resources.

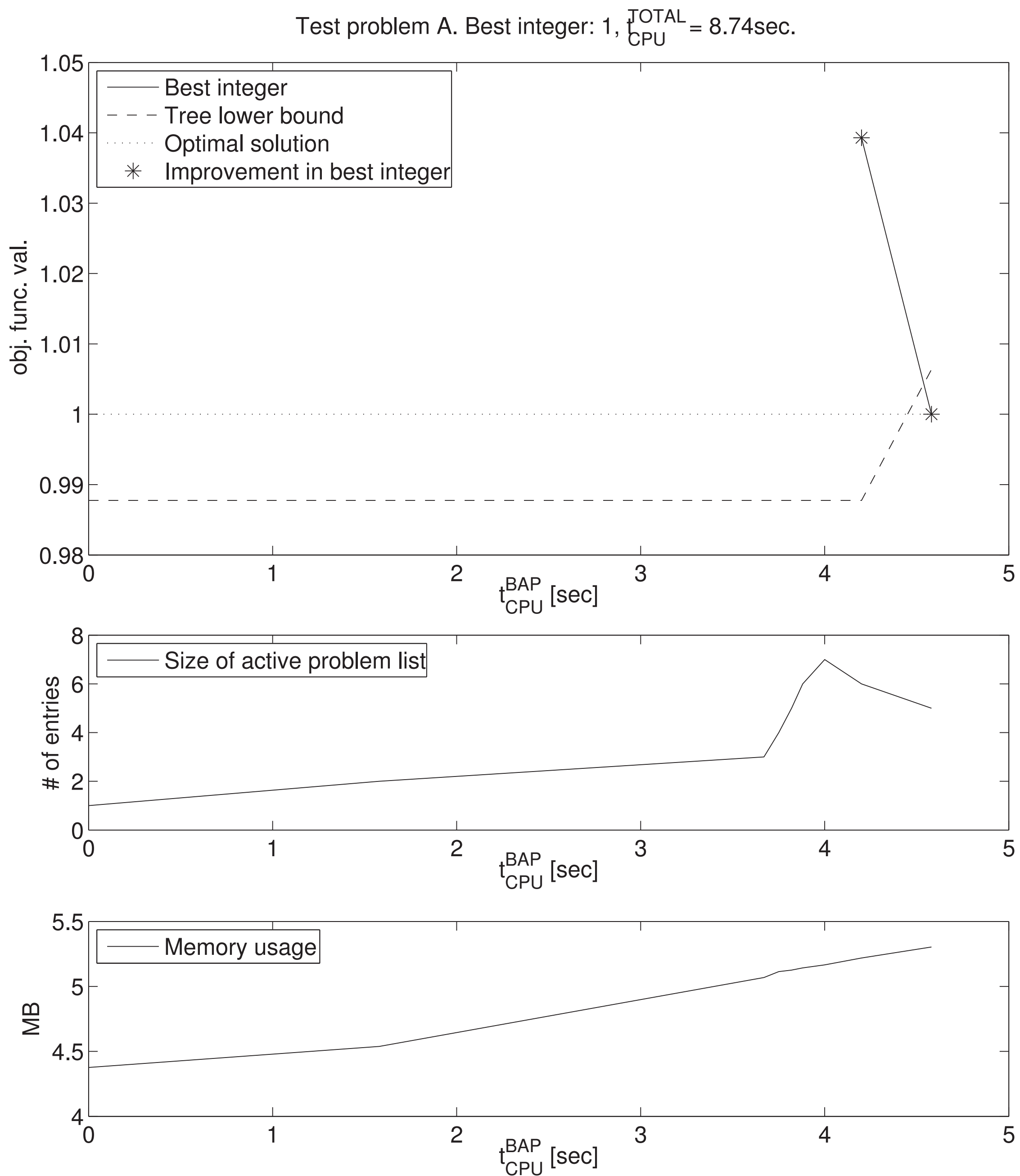


FIGURE 7.7: Final BAP algorithm applied to the instance A. Optimal value $z^*_{MBLP} = 1$ found after $t^{\text{TOTAL}}_{\text{CPU}} = t^{\text{Init}}_{\text{CPU}} + t^{\text{BAP}}_{\text{CPU}} \approx 8.7$ seconds (initialization 4.1 seconds, BAP procedure 4.6 seconds).

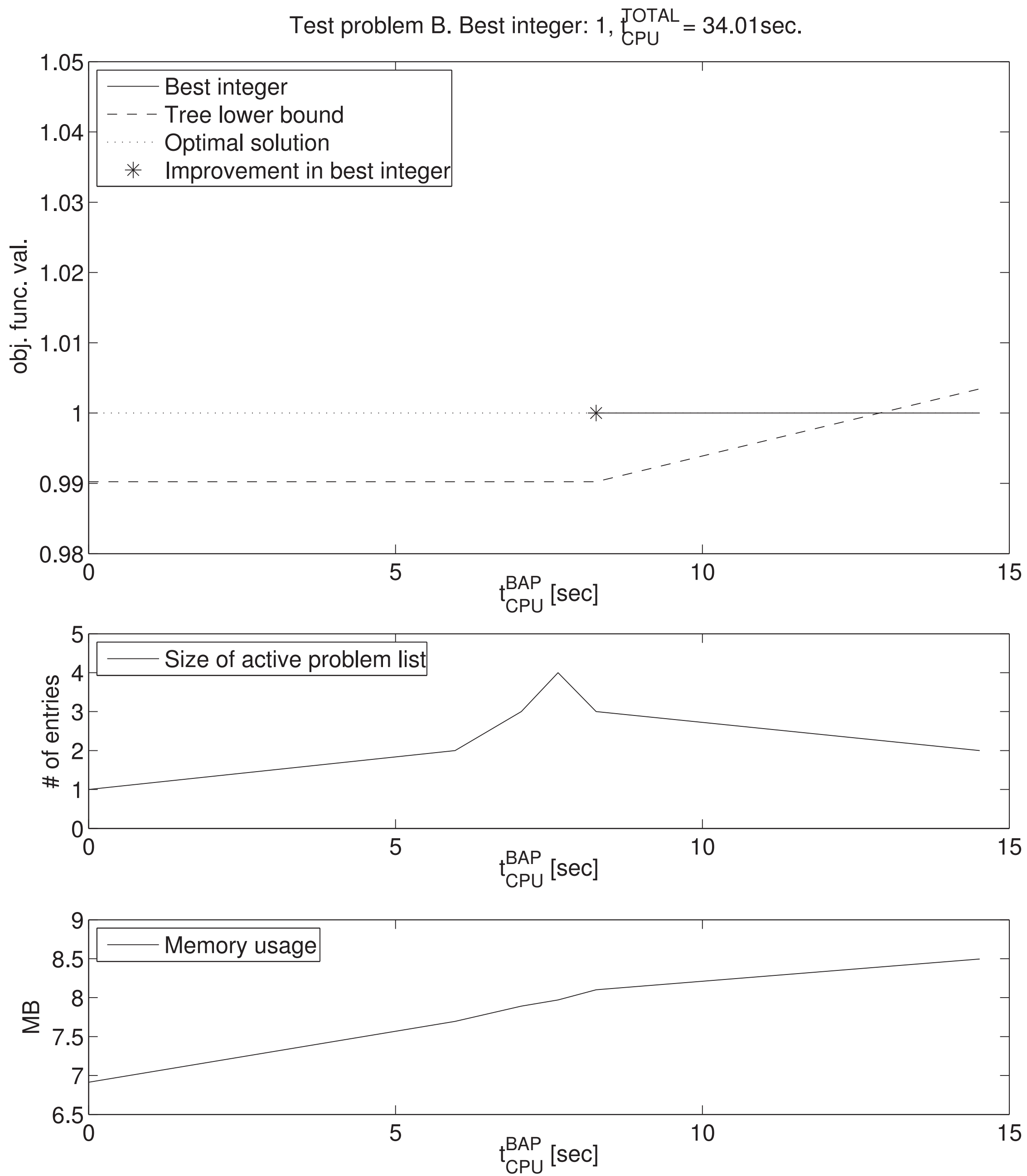


FIGURE 7.8: Final BAP algorithm applied to the instance B. Optimal value $z^*_{MBLP} = 1$ found after $t^{\text{TOTAL}}_{\text{CPU}} = t^{\text{Init}}_{\text{CPU}} + t^{\text{BAP}}_{\text{CPU}} \approx 34.0$ seconds (initialization 19.5 seconds, BAP procedure, 14.5 seconds).

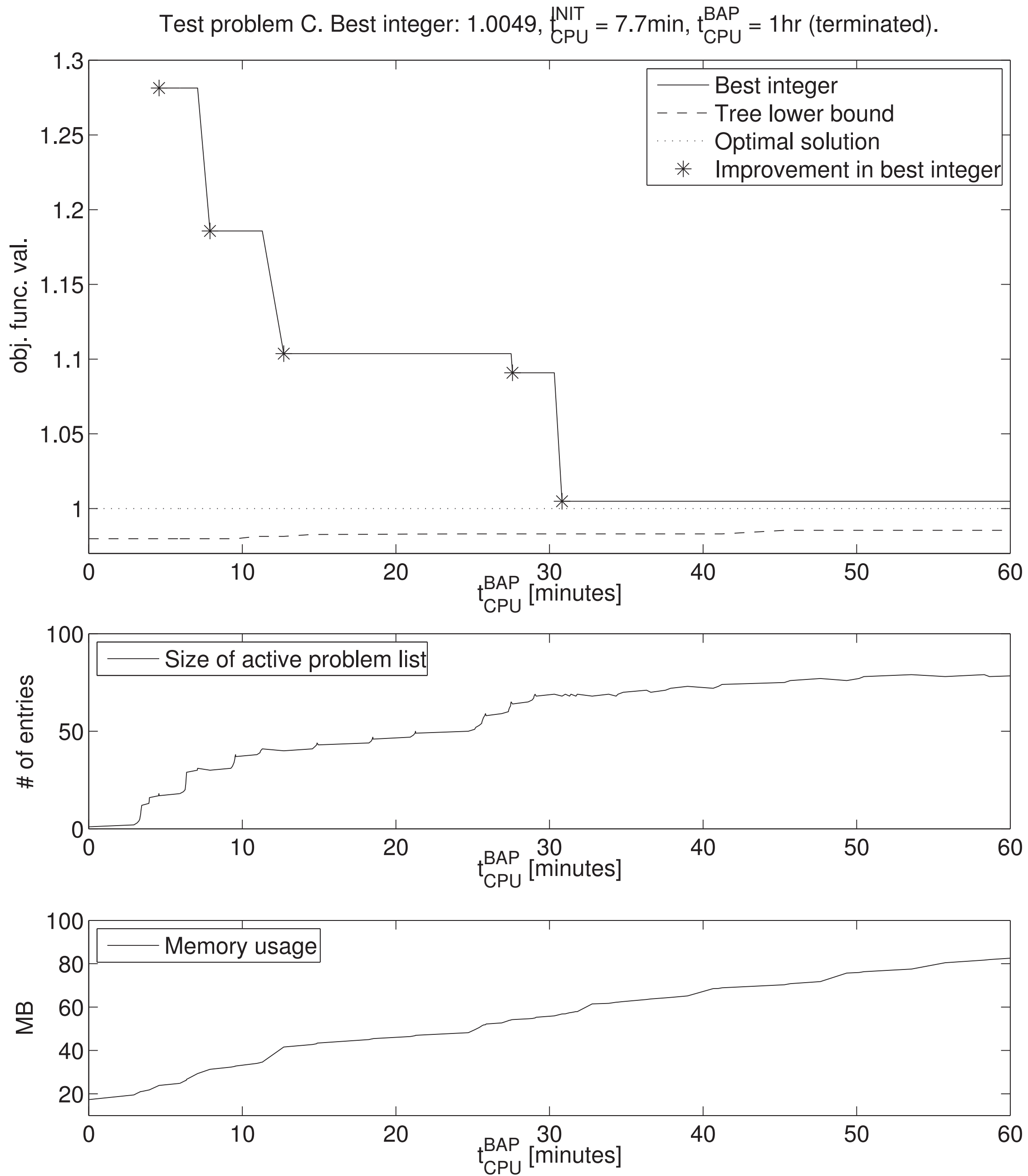


FIGURE 7.9: Final BAP algorithm applied to the instance C. Best incumbent solution $\bar{z}_{MBLP} = 1.0049$ found after 31 minutes of the BAP procedure. Algorithm terminated prematurely after 60 minutes of the BAP procedure, with an upper bound vs. lower bound gap of ∼2% ($\bar{z}_{MBLP}/\underline{z}_{MBLP}^{\text{tree}} = 1.0197$).

Chapter 8

Conclusions and future work

As shown in past chapters, it is possible, limited by the scope of a Master's Thesis, to implement a fully functional problem-specific branch-and-price algorithm, in this case applied to an instance of the ORP. Moreover, as noted in the beginning of the previous chapter, the notation of a *'final'* BAP implementation is really only restricted by the scope and limitations we choose to set for such an implementation. Hence, the performance of this particular algorithm—defined by computational usage and speed—depends heavily on the time limits naturally inhibited by the reserved time for a Master Thesis project, and it is not difficult to find areas that would be interesting subjects for further studies and improvements of the implemented algorithm.

Column generation and column management

We already mentioned the lack of column management; a well-thought out problem specific heuristic column management, making use of our knowledge of the original problem, would arguably improve the performance of the algorithm tremendously, especially for larger problems where the solving of the RMP:s becomes slower as more nodes are processed. E.g. in [19], while describing a general CG scheme, it's proposed that in between each step of the iterative solving of the RMP—within the CG process—non-basic columns with high positive reduced costs (in [19] described for negative reduced costs, for CG of a maximization problem) are removed from the column pool. Now, within a BAP network, a non-basic *'bad'* column present for the RMP in some branch is not necessarily bad in another branch within the BAP tree. Moreover, it is not necessarily non-basic in other branches within the BAP tree. These observations lead towards another interesting area of study: could we possibly make use of separate column pools for different parts of the BAP tree?

In parallel with the implementation of column management, preferrably also the possibe customization of the CG process should be studied; if we're to implement a

column management that repeatedly—fully or partly—empties the column pool, it would be sane-minded to also investigate heuristic techniques that could construct columns with negative reduced costs quicker than by fully solving the pricing subproblems.

Solving the pricing subproblems within the CG framework

Another key factor that could possibly be improved is the solving of the pricing subproblems. CPLEX is a great MIP solver, but with knowledge about the actual problem related to the pricing subproblems, it should be possible to create a custom-built solver outperforming CPLEX, knit specifically for our kind of subproblems, presumably by making use of the in-depth theory of the replacement polytope, as presented in [2]. As the pricing subproblems are solved iteratively for each CG step, which in turn is an iterative process nested within a BAP framework, reducing the solution time of the subproblems could have a great impact on the overall performance of the BAP implementation. Furthermore, as the purpose of the subproblems is—given dual variable values—to generate the columns with low reduced costs (as we solve them to optimality: the lowest), it could possibly be an alternative to combine a custom-built pricing subproblem solver with heuristics that could terminate the solver pre-maturely given that any column with a *'sufficiently low'* negative reduced cost has been found/constructed, where the *'sufficiency'* test would be the key test factor to be tuned.

Further studies on sophisticated branching rules

As shown in Section 7.6, the choice of branching rule can have a huge effect on the overall performance of the algorithm. This is intuitive, as the branching rule can be seen as the single responsible factor for the appearance of the BAP tree; the node picking rule decides where in the tree the further growth (branching) or termination (pruning) of tree branches should be investigated, but the branching rule single-handedly decides upon the *'species'* of each new branch and leaf. A bad branching rule can give rise to long branches—nurtured by the node picking rule—which should not have been available for the node picking rule in the first place. As a suggestion, the pseudocost branching rule could be extended to the *reliability pseudocost branching rule* [21], which should thereafter be compared with the implementation of the related and further extended *hybrid branching rule* [40].

Extending the case study

The first subject for future study—regarding the case study—that comes to mind is naturally the extension of the ORP covered in Chapter 5 to the case where components *within* modules are dependent. Recall from Chapter 5 that this concerns dependence in the sense that the access to components within modules requires the removal[1] of other components in the same module, prior to reaching the component that is to be replaced, and that there is naturally a cost associated with the removal of each component. Another extension would be to adapt the BAP implementation to the case where maintenence costs are not time independent, e.g., increasing or decreasing with time; see [2].

The most notable case study development, however, would be to investigate how the algorithm could be adapted to the bi-objective extension of ORP, the PMSPIC as presented in [26]. It would be interesting to make use of the results and knowledge gained from the current BAP implementation as applied to the ORP—as presented in this thesis—and extend the study to the PMSPIC, investigating to what extent the knowledge of the specific properties of the PMSPIC could be used to customize the key factors in the BAP implementation; most noteably CG (solving of the subproblems; generation of good dual variables) and the variable branching rule. E.g., would it be possible to knit together custom variations of the node picking rule and the pseudocost branching rule, with bias for branching upon variables that *'historically'*—in a pseudocost sense—minimizes risk, while picking nodes in the BAP tree focusing on minimizing costs?

Finally, PMSPIC as presented in [26] is not only interesting due to its application in a bi-objective approach, but also as a single-objective model—described in [26] simply as *The 0-1 ILP model*—which is similar to the one-module case of the ORP (i.e., in the context of the case study covered in this thesis, the MBLP (5.3)); however making use of a different definition of the (binary) component replacement variables (x variables). In the MBLP (5.3)—as is presented in Section 5.2—the (binary) component replacement variables x_{it}^m are used to describe whether component i in module m is replaced at time t ($x_{it}^m = 1$) or not ($x_{it}^m = 0$). In the 0-1 ILP model, the (binary) component replacement variables are defined as x_{ist}, describing whether component i is replaced at times t and s, but *not in-between these times*. This gives rise to an ILP with a network flow structure, which can readily be taken advantage of using LP techniques. The 0-1 ILP model would—in the context of the ORP case study covered in Chapter 5—be interesting to study using a problem-specific BAP implementation, much like has been performed in this thesis for the MBLP (5.3).

[1] Removal, not necessarily replacement.

Improving the use of computer resources and rationalizing the programming code

From a computational point of view, there are quite many apparent areas eligible to improvement. First of all, for the multi-module maintenance problem, in our BAP implementation, the pricing subproblems are solved sequentially. Almost every modern processor contain multiple cores, and for multiple independent subproblems it would be natural to thread up the subproblems solution step, so that subproblems could be solved in parallel. In a more sophisticated BAP network, BAP nodes could also be processed simultaneously with parallel computing, but where the parallelization of the solving of the pricing subproblems is quite intuitive, while a parallelization of the BAP procedure in not as simple. E.g., in the case of solving two BAP nodes simultaneously, the two solution processes depend quite a strongly on each other, and it's important to rigorously define how the two processes should share information that are discovered in the processing of the nodes. The node picking rule for choosing parallel nodes must also be quite sophisticated; in excess of choosing *'good'* nodes it should arguably also take into account the relation between nodes that are processed simultaneusly. It would obviously be a waste of computing resources to process a node that would not even have been considered, provided the knowledge about the solution of another node beeing processed in parallel. As a simple example in such a process, it would seem natural to choose nodes with similar LP bounds for simultaneous processing. This discussion of parallel computing concerns mainly threading of processes using a single multi-cored processor, but for really large problems parallelization could be applied in the context of cloud computing—several different computers connected through a fast network—in which case each parallel process will have a separate local memory, further complicating how to pdistribute the information contained locally to the parallel processes; see [41].

Finally, the current BAP implementation most probably contain programming bottle-necks which are not directly associated with the external solver, and the future work of this as well as any BAP solver would be the fine-tuning of the actual programming implementation, to avoid an unnecessary waste of computational resources.

Summary

As the final paragraph of this thesis, we summarize the discussion above, and list—for the hypothetical situation in which the thesis project would have been extended to another semester—the five top issues that would have been adressed in improving the current BAP implementation:

- *Column management and heuristic customization of the* CG: for test problems that

resulted in somewhat larger BAP trees, it was apparent that a major part of the computational time was condensed to nodes which were processed late in the BAP process—an expected behavior when the column pool is allowed to grow monotonically—resulting in LP RMP:s that size-wise grow for each node processed. Within this concept: possibly using separate column pools for different parts of the BAP tree.

- *Problem specific heuristics*: to create a good initial binary valued solution (best incumbent solution) and make use of RMP solutions to construct columns—i.e., binary valued solutions to the original program—that could possibly lower the upper bounds quicker than when waiting for binary valued solutions to be encountered in the BAP tree. Also, heuristics for constructing columns with negative reduced costs, to be used as an alternative/complement to the solving of the pricing subproblems.

- *Branching rules*: further work with the variable branching rule, examining reliability [21] and hybrid branching [40].

- *Node picking rule*: studies to find a better node picking rule, incoorporating heuristics, as well as a more thorough review of associated studies in the academic literature.

- *Parallel computing*: to solve pricing subproblems in parallel, and possibly examining the larger challenge; to solve BAP nodes in parallel. However limited to threading of processes using one single multi-cored process (with access to the same local memory).

Finally, in addition to addressing these algorithmic and implementational matters, the BAP implementation should, as a suggestion, be adapted to and further studied and tested with regard to the bi-objective PMSPIC problem [26].

Bibliography

[1] T. Almgren, N. Andréasson, M. Palmgren, M. Patriksson, A.-B. Strömberg, A. Wojciechowski, and M. Önnheim, "Optimization models for improving periodic maintenance schedules by utilizing opportunities," in *4th Joint World Conference on Production & Operations Management / 19th International Annual EurOMA Conference*, (Amsterdam, The Netherlands), June 1st–July 5th 2012.

[2] T. Almgren, N. Andréasson, M. Patriksson, A.-B. Strömberg, A. Wojciechowski, and M. Önnheim, "The opportunistic replacement problem: theoretical analyses and numerical tests," *Math. Meth. of Oper. Res.*, vol. 76, no. 3, pp. 289–319, 2012.

[3] IBM ILOG CPLEX, *IBM ILOG CPLEX V12.1 - User's Manual for CPLEX*. IBM Corp., 2009. Web. 20 November, 2015. ftp://public.dhe.ibm.com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex.pdf.

[4] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. New York, NY, USA: John Wiley & Sons, Inc., 2nd ed., 1990.

[5] R. Cottle, E. Johnson, and R. Wets, "George B. Dantzig (1914–2005)," *Notices of the American Mathematical Society*, vol. 54, pp. 344–362, March 2007.

[6] N. Andréasson, A. Evgrafov, and M. Patriksson, *An Introduction to Continuous Optimization*. Studentlitteratur, 2005.

[7] K. G. Murty, *Linear Programming*. John Wiley & Sons, Inc., 1st ed., 1983.

[8] S. Gass, *Linear Programming: Methods and Applications*. Dover Publications, 5th ed., 1985.

[9] L. A. Wolsey, *Integer Programming*. Wiley-Interscience, 1st ed., 1998.

[10] O. Bjerkholt, "Some unresolved problems of mathematical programming," in *Economic Models: Methods, Theory and Applications* (D. Basu, ed.), ch. 1, World Scientific Publishing Company, 2009.

[11] S. G. Garille and S. I. Gass, “Stigler's diet problem revisited,” *Operations Research*, vol. 49, no. 1, pp. 1–13, 2001.

[12] S. I. Gass, “The first linear-programming shoppe,” *Operations Research*, vol. 50, no. 1, pp. 61–68, 2002.

[13] J. Gondzio and R. Kouwenberg, “High-performance computing for asset-liability management,” *Operations Research*, vol. 49, no. 6, pp. 879–891, 2001.

[14] M. E. Lübbecke and J. Desrosiers, “Selected topics in column generation,” *Operations Research*, vol. 53, no. 6, pp. 1007–1023, 2005.

[15] M. E. Lübbecke, “Column generation,” in *Wiley Encyclopedia of Operations Research and Management Science*, John Wiley & Sons, Inc., 2010.

[16] F. Vanderbeck, “On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm,” *Operations Research*, vol. 48, pp. 111–128, January 2000.

[17] K. Aardal, G. Nemhauser, and R. Weismantel, *Discrete Optimization*. Handbooks in operations research and management science, Elsevier, 2005.

[18] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[19] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, “Branch-and-price: Column generation for solving huge integer programs,” *Operations Research*, vol. 46, no. 3, pp. 316–329, 1998.

[20] F. Vanderbeck and L. A. Wolsey, “An exact algorithm for IP column generation,” *Operations Research Letters*, vol. 19, no. 4, pp. 151–159, 1996.

[21] T. Achterberg, T. Koch, and A. Martin, “Branching rules revisited,” *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[22] F. Besnard, M. Patriksson, A.-B. Strömberg, A. Wojciechowski, and L. Bertling, “An optimization framework for opportunistic maintenance of offshore wind power system,” in *IEEE Bucharest PowerTech Conference*, (Bucharest, Romania), pp. 2970–2976, June 28th–July 2nd 2009.

[23] J. Nilsson, M. Patriksson, A.-B. Strömberg, A. Wojciechowski, and L. Bertling, “An opportunistic maintenance optimization model for shaft seals in feed-water pump systems in nuclear power plants,” in *IEEE Bucharest PowerTech Conference*, (Bucharest, Romania), pp. 2962–2969, June 28th–July 2nd 2009.

[24] M. Patriksson, A.-B. Strömberg, and A. Wojciechowski, "The stochastic opportunistic replacement problem, part I: models incorporating individual component lives," *Annals of Operations Research*, vol. 224, no. 1, pp. 25–50, 2015.

[25] M. Patriksson, A.-B. Strömberg, and A. Wojciechowski, "The stochastic opportunistic replacement problem, part II: a two-stage solution approach," *Annals of Operations Research*, vol. 224, no. 1, pp. 51–75, 2015.

[26] E. Gustavsson, M. Patriksson, A.-B. Strömberg, A. Wojciechowski, and M. Önnheim, "Preventive maintenance scheduling of multi-component systems with interval costs," *Computers* & *Industrial Engineering*, vol. 76, pp. 390–400, 2014.

[27] T. Achterberg, "SCIP: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, pp. 1–41, June 2009.

[28] Gurobi Optimization, Inc., *Gurobi Optimizer - Reference Manual - Version 6.5*. Gurobi Optimization, Inc., 2015. Web. 20 November, 2015. www.gurobi.com/documentation/6.5/refman.pdf.

[29] Gurobi Optimization, Inc., "Meet some of the people behind Gurobi," 2015. Web. 20 November, 2015. www.gurobi.com/company/management-team.

[30] COIN OR, "COIN-OR branch-and-cut MIP solver," 2015. Web. 20 Nov., 2015. https://projects.coin-or.org/Cbc.

[31] Gurobi Optimization, Inc., "Dr. Tobias Achterberg joins the Gurobi Optimization R&D team," 3 March 2014. Web. 20 November, 2015. www.gurobi.com/company/news/dr.-achterberg-joins-gurobi.

[32] Gurobi Optimization Inc., "Gurobi 6.5 performance benchmarks," 2015. Web. 20 November, 2015. www.gurobi.com/pdfs/benchmarks.pdf.

[33] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, 2003.

[34] IBM ILOG CPLEX, *IBM ILOG CPLEX Concert Technology version 12.1 - C++ API - Reference Manual*. IBM Corp., 2009. Web. 20 November, 2015. ftp://public.dhe.ibm.com/software/websphere/ilog/docs/optimization/cplex/refcppcplex.pdf.

[35] C. T. Ragsdale and G. W. Shapiro, "Incumbent solutions in branch-and-bound algorithms: Setting the record straight," *Computers & Operations Research*, vol. 23, no. 5, pp. 419–424, 1996.

[36] S. Soltani, “Solution approaches for the opportunistic replacement problem: Benders decomposition and Cvátal-Gomory cut generation,” Master’s thesis, Matematiska institutionen, Stockholms universitet, Stockholm, Sweden, 2011.

[37] O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen, “Stabilized column generation,” *Discrete Mathematics*, vol. 194, pp. 229–237, January 1999.

[38] P. Wentges, “Weighted Dantzig–Wolfe decomposition for linear mixed-integer programming,” *International Transactions in Operational Research*, vol. 4, no. 2, pp. 151–182, 1997.

[39] P. Bonami, M. Kilinc, and J. Linderoth, “Algorithms and software for convex mixed integer nonlinear programs,” in *Mixed Integer Nonlinear Programming* (J. Lee and S. Leyffer, eds.), vol. 154 of *The IMA Volumes in Mathematics and its Applications*, pp. 1–39, Springer New York, 2012.

[40] T. Achterberg and T. Berthold, “Hybrid branching,” in *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR ’09, (Berlin, Heidelberg), pp. 309–311, Springer-Verlag, 2009.

[41] T. Ralphs, L. Ladanyi, and M. Saltzman, “Parallel branch, cut, and price for large-scale discrete optimization,” *Mathematical Programming*, vol. 98, no. 1–3, pp. 253–280, 2003.