Chapter 3

LP-Based Branch-and-Bound

3.1 Introduction

We have seen in Section 2.3 how to design branch-and-bound algorithms for optimisation problems. In this thesis our focus is not on optimisation problems in general, but on specific problems that are the subject of Part II. Each problem discussed in Part II of this thesis can be formulated as a *mixed integer linear programming problem*. A mixed integer linear programming problem can be defined as follows. Given a matrix $A \in \mathbb{Z}^{m \times n}$, vectors $\boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u} \in \mathbb{Z}^n$ and $\boldsymbol{b} \in \mathbb{Z}^m$, and a subset of the column indices $J \subseteq \{1, \ldots, n\}$, find

$$\begin{aligned} \max \quad & z(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} && \text{(3.1a)} \\ \text{subject to} \quad & A\boldsymbol{x} = \boldsymbol{b}, && \text{(3.1b)} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}, && \text{(3.1c)} \\ & \boldsymbol{x}_J \text{ integer}. && \text{(3.1d)} \end{aligned}$$

The special case with $J = \{1, \ldots, n\}$ is called a *(pure) integer linear programming problem*. If in an integer linear program all variables are allowed to take values zero or one only, then it is called a *zero-one* integer linear programming problem. Let

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\},$$

and

$$X = P \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_J \text{ integer}\}.$$

Problem (3.1) is equivalent to

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in X\},$$

which is an instance of an optimisation problem in the sense of Definition 2.1. The *LP relaxation* of problem 3.1 is obtained by removing the constraints (3.1d), which yields the problem

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P\}. \tag{3.2}$$

In this chapter we will study branch-and-bound algorithms that take advantage of the fact that LP relaxations can be solved efficiently. In case the entire formulation can be stored in the main memory of the computer, one can apply the basic LP-based branch-and-bound algorithm. However, it may occur that A is given implicitly, and in this case m and n can even be exponential in the size of a reasonable encoding of the problem data. Sometimes we are still able to handle such problems using some modification of the basic LP-based branch-and-bound algorithm. These modifications lead to so-called branch-and-cut, branch-and-price, and branch-price-and-cut algorithms, corresponding to the cases in which only a subset of the constraints, a subset of the variables, and both a subset of the constraints and of the variables are kept in main memory, respectively.

The area of integer linear programming was pioneered by Gomory [53] who developed his famous cutting plane algorithm for integer linear programming problems in the nineteen fifties. Two of the first branch-and-bound algorithms for integer linear programming were developed by Land and Doig [74] and Dakin [32] in the early nineteen sixties. Since that time, many articles, books and conferences have been devoted to the subject. Among the books on integer programming we mention Papadimitriou and Steiglitz [94], Schrijver [104], Nemhauser and Wolsey [88], and Wolsey [125]. Recent surveys of branch-and-bound algorithms for integer programming are by Jünger, Reinelt and Thienel [67] and Johnson, Nemhauser, and Savelsbergh [66]. Together with the papers by Padberg and Rinaldi [92] and Linderoth and Savelsbergh [79], these references are the main source of the ideas that led to the algorithms in this chapter. Recently, approaches to integer linear programming that are different from LP-based branch-and-bound have been reported on by Aardal, Weismantel and Wolsey [3].

The advances of the theory and the developments in computer hardware and software during the past four decades have resulted in algorithms that are able to solve relevant integer programming problems in practice to optimality. This makes linear integer programming an important subject to study.

Several software packages that allow for the implementation of customised branch-and-cut, branch-and-price, and branch-price-and-cut algorithms exist. Here we mention MINTO [84] and ABACUS [110]. In order to have full freedom of algorithmic design we have implemented our own framework for branch-and-cut, branch-and-price, and branch-price-and-cut, which we use in all computational experiments that are reported on in this thesis. The goal of this chapter is to give the reader the opportunity to find out what we actually implemented, without having to make a guess based upon the literature we refer to.

The remainder of this chapter is organised as follows. We describe a basic LP-based branch-and-bound algorithm in Section 3.2. We describe our version

of the branch-and-cut, branch-and-price, and branch-price-and-cut algorithms in Sections 3.3, 3.4, and 3.5, respectively.

3.2 An LP-Based Branch-and-Bound Algorithm

Here we refine the branch-and-bound algorithm from Section 2.3 for linear integer programming. The relaxations solved in a node of the branch-and-bound tree are given in Section 3.2.1. Sometimes it is possible to improve the linear formulation of the problem in a part of the branch-and-bound tree by tightening the bounds on the variables. This is discussed in Sections 3.2.2 and 3.2.3. We proceed by describing branching schemes that can be employed in Sections 3.2.4 and 3.2.5.

3.2.1 LP Relaxations

Consider iteration i of the branch-and-bound algorithm. The LP relaxation we solve in iteration i of the branch-and-bound algorithm is uniquely determined by its lower and upper bounds on the variables, which we will denote by $\boldsymbol{l}^i$ and $\boldsymbol{u}^i$, respectively. Let

$$P^i = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{l}^i \leq \boldsymbol{x} \leq \boldsymbol{u}^i\},$$

and

$$X^i = \{\boldsymbol{x} \in P^i \mid \boldsymbol{x}_J \text{ integer}\}.$$

The LP relaxation that we solve in iteration i, denoted by LP^i, is given by

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^i\}, \tag{3.3}$$

which is a linear program with bounded variables as discussed in Section 2.4. In the root node v_1 we take $\boldsymbol{l}^1 = \boldsymbol{l}$ and $\boldsymbol{u}^1 = \boldsymbol{u}$ to obtain the LP relaxation (3.2) of the original problem (3.1).

Note that the matrix A is a constant. In an implementation of the LP-based branch-and-bound algorithm this can be exploited by maintaining only one LP formulation of the problem. When formulating LP^i in iteration i of the branch-and-bound algorithm, we do this by imposing the bounds $\boldsymbol{l}^i, \boldsymbol{u}^i$ in this formulation.

Next, we keep track of the basis associated with the optimal solution to the LP relaxation that we solve in each node of the branch-and-bound tree. How we do this is explained in more detail in Section 3.2.2. Recall the construction of the branch-and-bound tree from Section 2.3. Consider iteration $i > 1$ of the branch-and-bound algorithm. The optimal basis $B \subseteq \{1, \ldots, n\}$ associated with the parent of node v_i in the branch-and-bound tree defines a dual solution $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$. Furthermore LP^i is derived from the LP relaxation associated with the parent of node v_i in the branch-and-bound tree by modifying only a small number of variable bounds. Therefore $\boldsymbol{\pi}$ is dual feasible to LP^i, and we can expect $\boldsymbol{\pi}$ to be close to optimal. This is exploited by solving the LP^i starting from $\boldsymbol{\pi}$ using the dual simplex algorithm.

3.2.2 Tightening Variable Bounds and Setting Variables

Suppose we have at our disposal a vector $\boldsymbol{x}^* \in X$ with $z(\boldsymbol{x}^*) = z^*$. Consider iteration i of the branch-and-bound algorithm in which LP^i was feasible, and suppose we have solved it to optimality. Recall that our branch-and-bound algorithm is correct as long as we do not discard any solution that is better than our current best solution from the remaining search-space (or, more precisely, if we maintain condition (2.3) as invariant). We can exploit the information obtained from the optimal solution of LP^i to tighten the bounds $\boldsymbol{l}^i$ and $\boldsymbol{u}^i$. The improved bounds are based on the value z^* and the reduced cost of non-basic variables in an optimal LP solution.

Let $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi})$ be an optimal primal-dual pair to LP^i, where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis $B \subseteq \{1, \ldots, n\}$. Further, let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let $L, U \subseteq \{1, \ldots, n\}$ be the sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \boldsymbol{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \boldsymbol{0}$. The reduced cost $c_j^{\boldsymbol{\pi}}$ can be interpreted as the change of the objective function per unit change of variable x_j. From the reduced cost optimality conditions (see Theorem 2.2) it follows that $x_j = u_j$ if $c_j^{\boldsymbol{\pi}} > 0$ and $x_j = l_j$ if $c_j^{\boldsymbol{\pi}} < 0$. Using these observations and the difference in objective function between the optimal LP solution and $\boldsymbol{x}^*$ we can compute a new lower bound for x_j if $c_j^{\boldsymbol{\pi}} > 0$, and a new upper bound for x_j if $c_j^{\boldsymbol{\pi}} < 0$. These improved bounds are given by $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \mathbb{Q}^n$, where

$$\tilde{l}_j^i = \begin{cases} \max(l_j^i, u_j^i + \lceil (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}} \rceil), & \text{if } j \in U \cap J, \\ \max(l_j^i, u_j^i + (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}}), & \text{if } j \in U \setminus J, \\ l_j^i & \text{otherwise,} \end{cases}$$

and

$$\tilde{u}_j^i = \begin{cases} \min(u_j^i, l_j^i + \lfloor (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}} \rfloor), & \text{if } j \in L \cap J, \\ \min(i_j^i, l_j^i + (z^* - z^{\mathrm{LP}})/c_j^{\boldsymbol{\pi}}), & \text{if } j \in L \setminus J, \\ u_j^i, & \text{otherwise.} \end{cases}$$

The following proposition proves the correctness of the improved bounds.

Proposition 3.1. *All $\boldsymbol{x}^{IP} \in X^i$ with $z(\boldsymbol{x}^{IP}) \geq z^*$ satisfy $\tilde{\boldsymbol{l}}^i \leq \boldsymbol{x}^{IP} \leq \tilde{\boldsymbol{u}}^i$.*

Proof. Let $\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi}, z^{\mathrm{LP}}, L, U$ be as in the construction of $z^*, i, \tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. Assume that there exists a vector $\boldsymbol{x}^{\mathrm{IP}} \in X^i$ with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$. Since $\boldsymbol{x}^{\mathrm{IP}} \in X^i \subseteq P^i$ we have $A\boldsymbol{x}^{\mathrm{IP}} = \boldsymbol{b}$, so by Proposition 2.1 we can write

$$z(\boldsymbol{x}^{\mathrm{IP}}) = \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{x}_U^{\mathrm{IP}},$$

and since $\boldsymbol{x}^{\mathrm{LP}} \in P^i$ we can write

$$\begin{aligned} z(\boldsymbol{x}^{\mathrm{LP}}) &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{x}_L^{\mathrm{LP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{x}_U^{\mathrm{LP}} \\ &= \boldsymbol{\pi}^T \boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T \boldsymbol{l}_L^i + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T \boldsymbol{u}_U^i. \end{aligned}$$

Observe that $\boldsymbol{x}^{\mathrm{IP}} \in P_i$ implies $\boldsymbol{x}^{\mathrm{IP}}_L \geq \boldsymbol{l}^i_L$ and $\boldsymbol{x}^{\mathrm{IP}}_U \leq \boldsymbol{u}^i_U$, so $\boldsymbol{x}^{\mathrm{IP}}_L - \boldsymbol{l}^i_L \geq \mathbf{0}$ and $\boldsymbol{x}^{\mathrm{IP}}_U - \boldsymbol{u}^i_U \leq \mathbf{0}$. Now, choose $j \in U$ arbitrarily. Note that $\boldsymbol{x}^{\mathrm{IP}} \in X^i \subseteq P^i$ directly gives $x^{\mathrm{IP}}_j \geq l^i_j$. Moreover,

$$\begin{aligned} z^* - z^{\mathrm{LP}} &\leq z(\boldsymbol{x}^{\mathrm{IP}}) - z(\boldsymbol{x}^{\mathrm{LP}}) \\ &= (\boldsymbol{c}^{\boldsymbol{\pi}}_L)^T(\boldsymbol{x}^{\mathrm{IP}}_L - \boldsymbol{l}^i_L) + (\boldsymbol{c}^{\boldsymbol{\pi}}_U)^T(\boldsymbol{x}^{\mathrm{IP}}_U - \boldsymbol{u}^i_U) \\ &\leq c^{\boldsymbol{\pi}}_j(x^{\mathrm{IP}}_j - u^i_j). \end{aligned}$$

Hence,

$$x^{\mathrm{IP}}_j \geq u^i_j + (z^* - z^{\mathrm{LP}})/c^{\boldsymbol{\pi}}_j.$$

If $j \in U \setminus J$ this proves that $x^{\mathrm{IP}}_j \geq \tilde{l}^i_j$. Otherwise, $j \in U \cap J$, and $x^{\mathrm{IP}}_j \geq \tilde{l}^i_j$ by integrality of x^{IP}_j. Because j was chosen arbitrarily we have $\boldsymbol{x}^{\mathrm{IP}} \geq \tilde{\boldsymbol{l}}^i$. The proof that $\boldsymbol{x}^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$ is derived similarly starting from an arbitrarily chosen index $j \in L$. □

Denote the sub-tree of the branch-and-bound tree that is rooted at node v_i by T_{v_i}. We can tighten the bounds on the variables after solving the LP in iteration i by replacing the bounds $\boldsymbol{l}^i, \boldsymbol{u}^i$ by $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. By Proposition 3.1 we do not discard any solution satisfying the integrality conditions that is better than the current best solution $\boldsymbol{x}^*$ in doing so, which means that we maintain condition (2.3) as invariant. The improved bounds are used in all iterations of the branch-and-bound algorithm that are associated with a node in the branch-and-bound tree in the sub-tree rooted at node v_i, the node in the branch-and-bound tree associated with iteration i. In our implementation of LP-based branch-and-bound we do not tighten the bounds on continuous variables.

When a variable index $j \in \{1, \ldots, n\}$ satisfies $l^i_j = u^i_j$ we say that x_j is *set* to $l^i_j = u^i_j$ in iteration i (node v_i). When a variable is set in the root node of the branch-and-bound tree, it is called *fixed*. If $l^i_j < u^i_j$, we say that that x_j is *free* in iteration i (node v_i). Variable setting based on reduced cost belongs to the folklore and is used by many authors to improve the formulation of zero-one integer programming problems (for example by Crowder, Johnson, and Padberg [31] and Padberg and Rinaldi [92]). The version in Proposition 3.1 is similar to the one mentioned by Wolsey [125, Exercise 7.8.7].

Note that the new bounds are a function of $\boldsymbol{\pi}, z^{\mathrm{LP}}$, and z^*. As a consequence, each time that we find a new primal solution in the branch-and-bound algorithm we can re-compute the bounds. Suppose we find an improved primal solution in iteration k. An original feature of our implementation of the branch-and-bound algorithm is that we re-compute the bounds in all nodes v_i of the branch-and-bound tree with $i \in \{1, \ldots, k\}$ that are on a path from the root node to a node $v_{k'}$ with $k' > k$. In order to be able to do this, we store a tree T' that mirrors the branch-and-bound tree. Each node w_i in T' corresponds to some iteration i of the branch-and-bound algorithm, and with w_i we store its parent $p(w_i)$ in T', and the values of $\boldsymbol{\pi}$, z^{LP}, and z^* for which we last computed the bounds

in w_i, and the bounds that we can actually improve in node w_i. The values of $\boldsymbol{\pi}$ are stored implicitly by storing only the differences of the optimal LP basis between node w_i and node $p(w_i)$.

The actual re-computation of bounds is done in a lazy fashion as follows. In iteration k of the branch-and-bound algorithm, we compute the path P from w_1 to w_k in T' using the parent pointers. Next, we traverse P from w_1 to w_k, and keep track of the final basis in each node using the differences, and of the best available bounds on each variable using the improved bounds that are stored in the nodes on P. Consider some node w_i in this traversal. If the value of z^* that is stored in w_i is less than the actual value, we re-compute the bounds in w_i. If any of the bounds stored in node w_i contradicts with bounds stored in a node w_j that preceded w_i in the traversal, we have found a proof that $X^k = \varnothing$ and we fathom node w_k. If any of the bounds stored in node w_i is implied by a bound stored in a node w_j that preceded w_i in the traversion, we remove it from node w_i.

Consider an execution of the branch-and-bound algorithm and let η denote the number of times we improve on the primal bound. For $w_i \in T'$ let J' denote the non-basic variables in the final basis of node w_i. Assuming that $n \gg m$, the time spent in the re-computation of bounds of node w_i is dominated by the re-computation of the reduced cost from the final basis of node w_i, which is of the order

$$O(\eta|\text{supp}(A_{J'})|). \tag{3.4}$$

In a typical execution of the branch-and-bound algorithm, we improve on the value of z^* only a few times. Moreover, in our applications we use branch-and-cut and branch-and-price algorithms that call sophisticated and often time-consuming subroutines in each iteration of the branch-and-bound algorithm. These observations imply that the bound (3.4) is dominated by the running time of the other computations performed in iteration i of the branch-and-bound algorithm in our applications. We believe that the benefit of having strengthened formulations is worth the extra terms (3.4) in the running time of the branch-and-bound algorithm, as the improved formulations help in reducing the size of the branch-and-bound tree.

3.2.3 GUB Constraints and Tightening Variable Bounds

Assume that row i of the constraints (3.1b) is of the form

$$\boldsymbol{x}(J_i) = 1,$$

where the variables $\boldsymbol{x}_{J_i}$ are required to be non-negative and integer. In this case row i is called a *generalised upper bound (GUB) constraint*. A GUB constraint models the situation in which we have to choose one option from a set of mutually exclusive options. GUB constraints were introduced by Beale and Tomlin [17]. In any feasible integer solution exactly one $j \in J_i$ has $x_j = 1$.

Whenever we have a formulation in which some of the constraints are GUB constraints, we can exploit this by strengthening the bounds on those variables that are in a GUB constraint using a slightly stronger argument than the one presented in Section 3.2.2 (see also Strijk, Verweij, and Aardal [107]). Let $I' \subseteq \{1, \ldots, m\}$ be the set of row indices corresponding to GUB constraints. The *conflict graph* induced by the GUB constraints is the graph $G = (V, E)$, where the node set V contains all variable indices that are involved in one or more GUB constraints and the edge set E contains all pairs of variable indices that are involved in at least one common GUB constraint, i.e.,

$$V = \bigcup_{i \in I'} J_i, \text{ and}$$
$$E = \{\{j, k\} \subseteq V \mid j, k \in J_i, i \in I'\}.$$

For a variable index $j \in V$, we denote by $N(j)$ the set of variable indices k that are adjacent to j in G, i.e., $N(j) = \{k \mid \{j, k\} \in E\}$.

The stronger arguments can be applied to all variables that are involved in at least one GUB constraint. For $j \in V$, whenever x_j has value one, the GUB constraints imply that $\boldsymbol{x}_{N(j)} = \mathbf{0}$, and whenever x_j has value zero the GUB constraints imply that for at least one $k \in N(j)$ x_k has value one. The strengthened argument for modifying the upper bound on x_j takes into account the reduced cost of the variables $\boldsymbol{x}_{N(j)}$, and the strengthened argument for modifying the lower bound on x_j takes into account the reduced cost of the variables x_k and $\boldsymbol{x}_{N(k)}$ for some properly chosen $k \in N(j)$.

Let z^* again denote the value of the best known solution in X. Consider iteration i in which (3.3) is feasible and let $(\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi})$ be an optimal primal-dual pair to (3.3) in iteration i where $\boldsymbol{\pi} = (\boldsymbol{c}_B^T A_B^{-1})^T$ for some basis $B \subseteq \{1, \ldots, n\}$. Let $z^{\mathrm{LP}} = z(\boldsymbol{x}^{LP})$, and let $L, U \subseteq \{1, \ldots, n\}$ be the maximal sets of variable indices with $\boldsymbol{c}_L^{\boldsymbol{\pi}} < \mathbf{0}$ and $\boldsymbol{c}_U^{\boldsymbol{\pi}} > \mathbf{0}$. The strengthened bounds $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \{0, 1\}^V$ are defined as

$$\tilde{l}_j^i = \begin{cases} \max(0, 1 + \lceil (z^* - z^{\mathrm{LP}}) / \min\{-\tilde{c}_k^{\boldsymbol{\pi}} \mid k \in N(j)\} \rceil), & \text{if } j \in U, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$\tilde{u}_j^i = \begin{cases} \min(1, \lfloor (z^* - z^{\mathrm{LP}}) / \tilde{c}_j^{\boldsymbol{\pi}} \rfloor), & \text{if } j \in L, \\ 1, & \text{otherwise,} \end{cases}$$

where for each $j \in V \setminus U$

$$\tilde{c}_j^{\boldsymbol{\pi}} = c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(N(j) \cap U).$$

Proposition 3.2. *All $\boldsymbol{x}^{IP} \in X^i$ with $z(\boldsymbol{x}^{IP}) \geq z^*$ satisfy $\tilde{\boldsymbol{l}}^i \leq \boldsymbol{x}_V^{IP} \leq \tilde{\boldsymbol{u}}^i$.*

Proof. Let $\boldsymbol{x}^{\mathrm{LP}}, \boldsymbol{\pi}, z^{\mathrm{LP}}, L, U$ be as in the construction of $z^*, i, \tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i$. Assume that there exists a vector $\boldsymbol{x}^{\mathrm{IP}} \in X^i$ with $z(\boldsymbol{x}^{\mathrm{IP}}) \geq z^*$. As in Proposition 3.1 we

can write

$$z(\boldsymbol{x}^{\mathrm{IP}}) = \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T\boldsymbol{x}_L^{\mathrm{IP}} + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T\boldsymbol{x}_U^{\mathrm{IP}}, \text{ and}$$
$$z(\boldsymbol{x}^{\mathrm{LP}}) = \boldsymbol{\pi}^T\boldsymbol{b} + (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T\boldsymbol{l}_L^i + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T\boldsymbol{u}_U^i.$$

Observe that $\boldsymbol{x}^{\mathrm{IP}} \in P_i$ implies $\boldsymbol{x}_L^{\mathrm{IP}} \geq \mathbf{0}$ and $\boldsymbol{x}_U^{\mathrm{IP}} \leq \mathbf{1}$, so $\boldsymbol{x}_U^{\mathrm{IP}} - \mathbf{1} \leq \mathbf{0}$. Now, choose $j \in L \cap V$ arbitrarily. Either $x_j^{\mathrm{IP}} = 0$ or $x_j^{\mathrm{IP}} = 1$. In the case that $x_j^{\mathrm{IP}} = 0$ we directly have $x_j^{\mathrm{IP}} \leq \tilde{u}_j^i$ since $\tilde{u}_j^i \geq 0$. So assume that $x_j^{\mathrm{IP}} = 1$. It follows that $\boldsymbol{x}_{N(j)}^{\mathrm{IP}} = \mathbf{0}$. But then,

$$\begin{aligned} z^* - z^{\mathrm{LP}} &\leq z(\boldsymbol{x}^{\mathrm{IP}}) - z(\boldsymbol{x}^{\mathrm{LP}}) \\ &= (\boldsymbol{c}_L^{\boldsymbol{\pi}})^T(\boldsymbol{x}_L^{\mathrm{IP}} - \boldsymbol{l}_L^i) + (\boldsymbol{c}_U^{\boldsymbol{\pi}})^T(\boldsymbol{x}_U^{\mathrm{IP}} - \boldsymbol{u}_U^i) \\ &\leq (\boldsymbol{c}_{L\cap V}^{\boldsymbol{\pi}})^T\boldsymbol{x}_{L\cap V}^{\mathrm{IP}} + (\boldsymbol{c}_{U\cap V}^{\boldsymbol{\pi}})^T(\boldsymbol{x}_{U\cap V}^{\mathrm{IP}} - \mathbf{1}) \\ &\leq c_j^{\boldsymbol{\pi}} x_j^{\mathrm{IP}} + (\boldsymbol{c}_{U\cap N(j)}^{\boldsymbol{\pi}})^T(\boldsymbol{x}_{U\cap N(j)}^{\mathrm{IP}} - \mathbf{1}) \\ &= c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)). \end{aligned}$$

Since $c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j)) < 0$, we find

$$(z^* - z^{\mathrm{LP}})/(c_j^{\boldsymbol{\pi}} - \boldsymbol{c}^{\boldsymbol{\pi}}(U \cap N(j))) \geq 1.$$

Hence $x_j^{\mathrm{IP}} \leq \tilde{u}_j^i$. Because j was chosen arbitrarily we have $\boldsymbol{x}_V^{\mathrm{IP}} \leq \tilde{\boldsymbol{u}}^i$.

The proof that $\boldsymbol{x}_V^{\mathrm{IP}} \geq \tilde{\boldsymbol{l}}^i$ is derived similarly starting from an arbitrarily chosen index $j \in U \cap V$, assuming that $x_j^{\mathrm{IP}} = 0$, and using the observation that $x_j^{\mathrm{IP}} = 0$ implies $x_k^{\mathrm{IP}} = 1$ for some index $k \in N(j)$, which in turn implies that $\boldsymbol{x}_{N(k)}^{\mathrm{IP}} = \mathbf{0}$. □

The strengthened criteria for setting variables based on reduced cost can be taken into account in an implementation that stores the reduced cost of the variables in an array by replacing $c_j^{\boldsymbol{\pi}}$ by $\min\{-\tilde{c}_k^{\boldsymbol{\pi}} \mid k \in N(j)\}$ for all $j \in U$ and by $\tilde{c}_j^{\boldsymbol{\pi}}$ for all $j \in L$ in this array. Having done this the strengthened bounds can be computed as in Section 3.2.2. The extra time needed for pre-processing the array is $O(|E|)$.

3.2.4 Branching Schemes

Here we describe branching schemes for LP-based branch-and-bound. After we have solved the LP relaxation (3.3) in iteration k, and we have found an optimal solution $\boldsymbol{x}^k \in P^k$ but $\boldsymbol{x}^k \notin X^k$, we have to replace P^k by two sets P_1^k and P_2^k.

Branching on Variables

The most common branching scheme in LP-based branch-and-bound is to select $j \in J$ such that x_j^k is fractional and to define P_1^k and P_2^k as

$$P_1^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \lfloor x_j^k \rfloor\}, \text{ and } P_2^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \lceil x_j^k \rceil\}.$$

This scheme was first proposed by Dakin [32] and is called *variable branching*. Its correctness follows from the observation that every solution $\boldsymbol{x} \in X$ satisfies $x_j \leq \lfloor x_j^k \rfloor$ or $x_j \geq \lceil x_j^k \rceil$.

The choice of the index j can be made in different ways. One way is to make a decision based on $\boldsymbol{x}^k$, possibly in combination with $\boldsymbol{c}$ and $\boldsymbol{x}^*$. An example of such a scheme, which we will refer to as *Padberg-Rinaldi branching* because of its strong resemblance with the branching rule proposed by Padberg and Rinaldi [92], is as follows. The goal here is to find an index j such that the fractional part of x_j^k is close to $\frac{1}{2}$ and $|c_j|$ is large. For each $j \in J$, let $f_j = x_j^k - \lfloor x_j^k \rfloor$ be the fractional part of x_j^k. First, we determine the values $L, U \in [0,1]$ such that

$$L = \max_{j \in J}\{f_j \mid f_j \leq \tfrac{1}{2}\}, \qquad \text{and} \qquad U = \min_{j \in J}\{f_j \mid f_j \geq \tfrac{1}{2}\}.$$

Next, given a parameter $\alpha \in [0,1]$ we determine the set of variable indices

$$J' = \{j \in J \mid (1-\alpha)L \leq f_j \leq U + \alpha(1-U)\}.$$

The set J' contains the variables that are sufficiently close to $\frac{1}{2}$ to be taken into account. From the set J', we choose an index j that maximises $|c_j|$ as the one to define P_1^k and P_2^k as before. In our implementation we have $\alpha = \frac{1}{2}$ as suggested by Padberg and Rinaldi.

Branching on GUB Constraints

Let $\boldsymbol{x}^k \in P^k$ be as above, and let $J^i = \operatorname{supp}(\boldsymbol{a}_i)$ denote the support of row i of the constraint matrix. Suppose that row i is a GUB constraint and that $\boldsymbol{x}^k_{J^i}$ has fractional components. Partition J^i into two non-empty disjoint sets J_1^i, J_2^i. Observe that any $\boldsymbol{x} \in X$ satisfies $\boldsymbol{x}(J_1^i) = 0$ or $\boldsymbol{x}(J_2^i) = 0$. Hence, we can define P_1^k and P_2^k by

$$P_1^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J_1^i} \leq \boldsymbol{0}\}, \quad \text{and} \quad P_2^k = P^k \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J_2^i} \leq \boldsymbol{0}\}.$$

The branching scheme defined this way is called *GUB branching* and is due to Beale and Tomlin [17]. GUB branching is generally believed to yield a more balanced branch-and-bound tree of smaller height, and hence of smaller size.

Before we can apply GUB branching, we need a way to choose the sets J_1^i and J_2^i. The most common way to do this is by using an ordering of the variables in J that is specified as input to the branch-and-bound algorithm. Then J is partitioned in such a way that the elements of J_1^i and J_2^i appear consecutively in the specified order. If we have such an ordering then J is called a *special ordered set* (more details can be found in Nemhauser and Wolsey [88]).

We follow a different approach, which is motivated by the observation that the problems from Part II do not exhibit a special ordering. Our aim is to choose J_1^i and J_2^i such that $\boldsymbol{x}^k(J_1^i)$ and $\boldsymbol{x}^k(J_2^i)$ are close to $\frac{1}{2}$. We do this by considering the problem

$$\max_S\{\boldsymbol{x}^k(S) | S \subseteq J, \boldsymbol{x}^k(S) \leq \tfrac{1}{2}\}, \tag{3.5}$$

which is an instance of the subset-sum problem. A strongly polynomial time approximation scheme for the subset-sum problem was given by Ibarra and Kim [61]. We use their algorithm to find J_1^i such that $(1 - \frac{1}{1000})\boldsymbol{x}^k(S^*) \leq \boldsymbol{x}^k(J_1^i) \leq \frac{1}{2}$, where S^* denotes an optimal solution to (3.5), and take $J_2^i = J \setminus J_1^i$. Note that the sets J_1^i, J_2^i constructed this way will be non-empty if $\boldsymbol{x}_J^k$ has fractional components. The precision of $\frac{1}{1000}$ in the calculation of J_1^i is arbitrary but seems to work well in our implementation.

3.2.5 Branching Decisions using Pseudo-Cost

Now that we have seen different branching schemes, we discuss how to choose between the options, that is, how to compare different branching decisions with each other and how to decide on which one to apply. We do this by using *degradation estimates*, that is, estimates on the degradation of the objective function that are caused by enforcing the branching decision. Degradation estimates have been studied by Driebeek [43], Tomlin [111], Bénichou, Gauthier, Girodet, Hentges, Ribière, and Vincent [18], Gauthier and Ribière [49], and most recently by Linderoth and Savelsbergh [79].

Focus on iteration k of the branch-and-bound algorithm and let $\boldsymbol{x}^k \in P^k$ be as in Section 3.2.4. Observe that all branching decisions of Section 3.2.4 are of the form

$$P_1^k = P^k \cap D_1, \qquad \text{and} \qquad P_2^k = P^k \cap D_2,$$

where $D_b \subseteq \mathbb{R}^n$ enforces the altered variable bounds for each $b \in \{1, 2\}$. In the following, we call D_b a *branch* ($b \in \{1, 2\}$), and a pair (D_1, D_2) a *branching option*. Our goal is to decide which branching option to apply. Once we decided to branch using a pair of branches (D_1, D_2) we will refer to it as a *branching decision*.

So we have a fractional $\boldsymbol{x}^k$, and after applying the ideas of Section 3.2.4 we find ourselves with a set of branching options from which we have to choose the one we will enforce. Denote the set of branching options by $\mathcal{D}$, and let $\mathcal{D} = \{(D_1^1, D_2^1), \ldots, (D_1^N, D_2^N)\}$. Later in this section we will discuss in detail how to obtain the set $\mathcal{D}$. We may assume that $\boldsymbol{x}^k \notin D_b^i$ for all choices of (i, b). For each $D = D_b^i$ a measure of the distance from $\boldsymbol{x}^k$ to D can be defined as

$$f(D) = \begin{cases} x_j^k - \tilde{u}_j, & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \tilde{u}_j\}, \\ \tilde{l}_j - x_j^k, & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \tilde{l}_j\}, \\ \boldsymbol{x}^k(J'), & \text{if } D = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J'} \leq \boldsymbol{0}\}. \end{cases}$$

Note that $0 < f(D_b^i) < 1$ for all $(i, b) \in \{1, \ldots, N\} \times \{1, 2\}$ if we select all branching options as in Section 3.2.4. The per-unit degradation of the objective function caused by enforcing D_b^i is called the *pseudo-cost* of D_b^i, and is given by

$$p_b^i = \frac{\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^k\} - \max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P^k \cap D_b^i\}}{f(D_b^i)}$$

for each $(i, b) \in \{1, \ldots, N\} \times \{1, 2\}$. Actually computing the pseudo-cost for all branches is time consuming, and is not believed to result in improved running times.

It is reported by Gauthier and Ribiére, and more recently by Linderoth and Savelsbergh that, although the pseudo-cost of each variable are different in each iteration of the branch-and-bound algorithm, the order of magnitude of the pseudo-cost of each variable is the same in all but a few iterations. Therefore, instead of calculating the true pseudo-costs, we maintain estimates $\tilde{p}_b^i$ of p_b^i for all D_b^i that have been of interest during the execution of the branch-and-bound algorithm. For each variable x_j with $j \in J$ we store the values of the estimates $\tilde{p}_1^i$ and $\tilde{p}_2^i$ of the degradation of the branches $\{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \leq \tilde{u}_j\}$ and $\{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq \tilde{l}_j\}$. For each $J' \subseteq J$ that was obtained by partitioning the index set of a GUB constraint we store the value of the estimate $\tilde{p}_b^i$ corresponding to the branch $D_b^i = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{J'} \leq \boldsymbol{0}\}$ in a hash table $\mathcal{H} : 2^J \to \mathbb{R}$ using open addressing [29].

We follow the ideas of Linderoth and Savelsbergh in that we do allow our algorithm to spend some time to compute good initial estimates. These initial estimates are calculated using the dual simplex algorithm with an iteration limit $L(d)$ that is a function of the depth d of node v_k in the branch-and-bound tree, i.e.,

$$L(d) = \max\left\{\left\lceil \frac{T\gamma}{2^d N} \right\rceil, 3\right\},$$

where T denotes the maximum amount of time we are willing to spend on level d of the tree (in seconds), and γ is an estimate of the number of simplex iterations performed per second. In our implementation T is four minutes and γ is determined after solving the LP in the root node. We maintain the degradation estimates during the execution of the algorithm by letting $\tilde{p}_b^i$ be the average over all observed degradations caused by enforcing D_b^i that led to feasible relaxations (discarding the initial estimate). The average degradation of a branch D can be maintained in $O(1 + T_D)$ time for each time that we branched on D, where T_D indicates the time needed to access the stored pseudo-cost of D. If D is a variable branch then $T_D = O(1)$, and if D is a GUB branch that affects $\boldsymbol{u}_{J'}$ for some set $J' \subseteq J$ then T_D is a random variable and $E[T_D] = O(|J'|)$.

The actual branching decision is made as follows. For each D_b^i we estimate a degradation

$$d_b^i = \tilde{p}_b^i f(D_b^i).$$

We choose to branch on $(D_1^{i^*}, D_2^{i^*}) \in \mathcal{D}$ determined by

$$i^* = \arg \max_{i \in \{1,\ldots,N\}} \{\alpha_1 \min(d_1^i, d_2^i) + \alpha_2 \max(d_1^i, d_2^i)\},$$

where in our implementation (α_1, α_2) equals $(2, 1)$ as suggested by Linderoth and Savelsbergh. We have only one exception to this rule, namely, if $d_1^{i^*} = d_2^{i^*} = 0$ then we use the Padberg-Rinaldi scheme. In this case all pseudo-costs are zero, and hence provide a poor criterion for comparing branching options.

To complete the description of our branching scheme, we have to specify how we obtain the set of branching options $\mathcal{D}$. We do this by including in $\mathcal{D}$ all branching options that are derived from fractional variables x_j with $j \in J$. In addition, we add to $\mathcal{D}$ a selected number of branching options that are derived from GUB constraints. One might be tempted to include in $\mathcal{D}$ a branching option derived from each GUB constraint that has fractional variables. However, if the number of GUB constraints in the formulation is relatively large then such a scheme would waste too much time in selecting GUB constraints for branching, eliminating the potential benefit obtained from branch selection by pseudo-cost.

Instead we use the following approach, which takes into account that GUB constraints may be added in later stages of the algorithm (this can occur in a branch-and-cut algorithm). Let $I_1 \subseteq \{1, \ldots, m\}$ denote the set of row-indices of GUB constraints, so for each $i \in I_1$ row i has the form

$$\boldsymbol{x}(J^i) = 1,$$

$\boldsymbol{x}_{J^i} \geq \mathbf{0}$, and $J^i \subseteq J$. For each $i \in I_1$ we keep track of the average pseudo-cost p_b^i of branches D_b^i obtained by partitioning J^i, that we use as a heuristic forecast of the pseudo-cost of the GUB branches obtained from row i in this iteration of the branch-and-bound algorithm (although the partitioning of J^i, and hence the resulting branching decision, might be totally different). We pre-select all GUB constraints that have at least four fractional variables in their support. Let $I_2 \subseteq I_1$ be the set of row indices corresponding to GUB constraints such that $\boldsymbol{x}_{J^i}$ has at least four fractional components for all $i \in I_2$. Let $I_3 \subseteq I_2$ be the set of row indices corresponding to GUB constraints from which we did derive a branching option, and let $I_4 = I_2 \setminus I_3$ be the set of row indices corresponding to GUB constraints from which we did not derive a branching option yet. For the rows indexed by I_3 we already have a heuristic forecast of the pseudo-cost of the branching decisions obtained from them, and for the rows indexed by I_4 we have not. From I_3, we select the $K(d)$ constraints with highest average pseudo-cost, and add those to I_4. Here, $K(d)$ is again a function of the depth d of node v^k in the branch-and-bound tree, and is given by

$$K(d) = \max\left\{\left\lceil \frac{m}{2^{d-1}} \right\rceil, 10\right\}.$$

For each $i \in I_4$ we partition J^i as in the normal GUB branching procedure and add the resulting pair of branches to $\mathcal{D}$ if the partition results in two sets J_1^i, J_2^i that each contain at least two fractional variables.

3.2.6 Logical Implications

Whenever we tighten variable bounds either by branching on them or by using reduced cost criteria, we try to tighten more bounds by searching for *logical implications* of the improved bounds. This is done using the structure of the problem at hand, so we do not go into detail here.

Consider iteration k of the branch-and-bound algorithm and let $\boldsymbol{x}^k \in P^k$ be as before. After enforcing the improved bounds obtained by calculating logical

implications we obtain a relaxation

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in \tilde{P}^k\}$$

for some $\tilde{P}^k \subset P^k$. It may occur that $\boldsymbol{x}^k \notin \tilde{P}^k$. If this is the case then we iterate by resolving the strengthened LP, and proceed again with the strengthening of variable bounds.

3.3 Branch-and-Cut

In this section we refine the LP-based branch-and-bound algorithm from Section 3.2 in such a way that it is no longer necessary to include all constraints in the LP formulation that is solved. The resulting refinement is known as a *branch-and-cut* algorithm, and allows us to use formulations with a large or possibly exponential number of constraints. Formulations with an exponential number of constraints are of interest because for some problems it is the only way to formulate them, and for other problems such formulations are significantly stronger than formulations of polynomial size. In both cases we need a *cutting plane* algorithm to solve the linear program, explained in Section 3.3.1, that iteratively adds violated *valid inequalities* to the LP formulation. The cutting plane algorithm relies on problem-specific subroutines that are known as *separation algorithms*. Some implementational issues that arise when dynamically adding constraints are discussed in Section 3.3.2. In Sections 3.3.3–3.3.5 we review some known classes of valid inequalities together with their separation algorithms, which we will use in Part II.

3.3.1 Valid Inequalities and The Cutting Plane Algorithm

Let $P \subseteq \mathbb{R}^n$, and $(\boldsymbol{\pi}, \pi_0) \in \mathbb{R}^n \times \mathbb{R}$. The inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is *valid* for P if

$$P \subseteq \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0\}.$$

Now, let P be a polyhedron in $\mathbb{R}^n$ and let $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ be a valid inequality for P. The set

$$F = P \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{\pi}^T\boldsymbol{x} = \pi_0\}$$

is called a *face* of P. The valid inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is said to *define* F. We say that F is a *proper* face of P if $F \neq \varnothing$ and $F \neq P$. A proper face is called a *facet* if it is not contained in any other proper face of P. Facet-defining valid inequalities are the strongest among all valid inequalities as they cannot be redundant (see e.g. Wolsey [125]). Recent surveys devoted to the subject of valid inequalities are by Aardal and van Hoesel [1] and by Marchand, Martin, Weismantel and Wolsey [82].

Recall the formulation of the mixed integer programming problem from Section 3.1. Let $\Pi \subseteq \mathbb{R}^n \times \mathbb{R}$ be such that each $(\boldsymbol{\pi}, \pi_0) \in \Pi$ defines a valid inequality for $\text{conv}(X)$. Then, the linear program

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P,\ \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0 \text{ for all } (\boldsymbol{\pi}, \pi_0) \in \Pi\} \tag{3.6}$$

```
cuttingPlaneSolver(z, A, b, l, u, separation, α1, α2)
{
    Π̄ := ∅; i := 0;
    do { i := i + 1;
        solve LP max{z(x) | Ax = b, l ≤ x ≤ u, π^T x ≤ π0 ∀(π, π0) ∈ Π̄};
        if LP infeasible return infeasible;
        let x* be an optimal solution of LP; z^i := z(x*);
        improve bounds and calculate logical implications, thus refining LP;
        if x* violates new bounds {
            resolve LP;
            if LP infeasible return infeasible;
            let x* be an optimal solution; z^i := z(x*);
        }
        Π' := separation(x*); Π̄ := Π̄ ∪ Π';
        ξ := max_{(π,π0)∈Π'} π0 − π^T x*;
    } while (i > 1 implies z^{i−1} − z^i ≥ α1) and ξ ≥ α2 and Π' ≠ ∅;
    return x*;
}
```

Algorithm 3.1: The Cutting Plane Algorithm

is a relaxation of the mixed integer programming problem (3.1). Hence, if we can solve (3.6) efficiently, then we can incorporate it in an LP-based branch-and-bound algorithm.

The *cutting plane algorithm* to solve (3.6) works in iterations (starting from 1), and maintains a set $\bar{\Pi} \subseteq \Pi$. Initially, the set $\bar{\Pi}$ is empty. In iteration i, we solve the linear program

$$\max\{z(\boldsymbol{x}) \mid \boldsymbol{x} \in P, \boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0 \text{ for all } (\boldsymbol{\pi}, \pi_0) \in \bar{\Pi}\}. \tag{3.7}$$

If this problem is infeasible then the cutting plane algorithm reports so and terminates. Otherwise, let $\boldsymbol{x}^*$ denote the optimal solution to (3.7), and let z^i denote the value of $z(\boldsymbol{x}^*)$. We first try to improve the bounds using Proposition 3.1. If this succeeds we try to improve more bounds using logical implications as described in Section 3.2.6. If any of these implications are not satisfied by $\boldsymbol{x}^*$, we resolve the LP. If the LP becomes infeasible, then we report so and terminate. Otherwise we update z^i and $\boldsymbol{x}^*$. The cutting plane algorithm then calls the separation algorithm with $\boldsymbol{x}^*$ as input, which returns a set $\Pi' \subseteq \Pi$ such that for each $(\boldsymbol{\pi}, \pi_0) \in \Pi'$ the valid inequality $\boldsymbol{\pi}^T\boldsymbol{x} \leq \pi_0$ is violated by $\boldsymbol{x}^*$. Let

$$\xi = \max_{(\boldsymbol{\pi},\pi_0)\in\Pi'} \pi_0 - \boldsymbol{\pi}^T\boldsymbol{x}^*$$

denote the maximum violation of any valid inequality defined by a pair $(\boldsymbol{\pi}, \pi_0) \in \Pi'$. If Π' is empty, or if $i > 1$, $z^{i-1} - z^i < \alpha_1$ and $\xi < \alpha_2$, then the cutting plane algorithm terminates and reports $\boldsymbol{x}^*$ as solution. Otherwise, it adds Π'

to $\bar{\Pi}$ and proceeds with the next iteration. The parameters (α_1, α_2) are used to avoid a phenomenon that is called *tailing-off* (see e.g. Padberg and Rinaldi [92] and Jünger *et al.* [67]) of the cutting plane algorithm. Tailing-off occurs when the separation routines find cuts that do not add much to the strength of the formulation. If this is the case we are basically wasting precious CPU time. In our implementation we have $(\alpha_1, \alpha_2) = (\frac{1}{100}, \frac{1}{10})$. Pseudo-code for the cutting plane algorithm can be found in Algorithm 3.1.

3.3.2 Initialising LP Relaxations and Constraint Pools

When one uses a cutting plane algorithm to solve the LP relaxation in each iteration, the question arises what constraints to keep in the LP formulation that is maintained by the branch-and-bound algorithm. One possible answer would be to keep all constraints that are generated by the separation routines in the formulation. However, this would result in an LP formulation that keeps growing over the execution of the branch-and-cut algorithm. Therefore the linear programs would take longer and longer to solve, which is undesirable.. Keeping all constraints in the formulation is not what we will do.

Focus on any iteration $i > 1$, and let v_j be the parent of node v_i in the branch-and-bound tree. Denote the optimal solution to LP^j by $\boldsymbol{x}^j$. Padberg and Rinaldi [92] initialise the formulation in each iteration $i > 1$ with the constraints of the matrix A, and the constraints generated by the separation routines that were satisfied with equality by $\boldsymbol{x}^j$, This is the approach we follow. Note that all constraints with a non-basic slack variable are satisfied with equality. Initialising the LP formulation associated with iteration i is then accomplished by eliminating valid inequalities from LP^{i-1} that are not satisfied with equality by $\boldsymbol{x}^j$, and adding those valid inequalities present in LP^j that are satisfied with equality and have a non-basic slack variable, next to imposing the bounds given in Section 3.2.1. The information about the status of the slack variables of valid inequalities that are added to the LP formulation is handled in the same way as the information about optimal the basis in Section 3.2.2.

A second issue that arises when one uses the cutting plane algorithm and one initialises the LP formulations as above is the following. It might be advantageous to cache constraints that are produced by the separation algorithm by storing them in a constraint pool, and then checking the constraint pool for violated valid inequalities before calling the separation algorithm. Checking the constraint pool for violated valid inequalities is called *pool separation* and is implemented by computing an inner product $\boldsymbol{\pi}^T\boldsymbol{x}$ for each valid inequality $(\boldsymbol{\pi}, \pi_0)$ in the constraint pool and comparing the result with π_0. If a constraint in the pool is violated in some iteration of the branch-and-cut algorithm, it is reported as violated and not checked again for the remainder of this iteration. As separation algorithms often solve non-trivial combinatorial optimisation problems, this can indeed be advantageous. However, the size of a constraint pool typically grows over the execution of the algorithm, causing the pool separation to take longer and longer as the branch-and-cut algorithm proceeds.

Consider any iteration of the branch-and-cut algorithm and denote the set

of valid inequalities in the constraint pool by

$$\bar{\Pi} = \{(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i \mid i \in \{1, \ldots, |\bar{\Pi}|\}\}.$$

To avoid the pool separation from stalling our branch-and-cut algorithm, we delete constraints from the constraint pool at certain times. For $i \in \{1, \ldots, |\bar{\Pi}|\}$ we keep track of the number of iterations of the branch-and-bound algorithm k_1^i that the inequality $(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i$ was in the pool since it was last inserted, and the number of iterations k_2^i in which it was actually reported as violated during this period of time. The valid inequality $(\boldsymbol{\pi}^i)^T \boldsymbol{x} \leq \pi_0^i$ is kept in the pool at least as long as $k_1^i \leq \alpha_1$ for some parameter α_1. However, as soon as $k_1^i > \alpha_1$ and $k_2^i / k_1^i < \alpha_2$ we delete it. The advantage of this scheme is that the number of iterations that a constraint stays in the pool is linear in the number of iterations in which it is actually reported as violated. In our implementation we have $(\alpha_1, \alpha_2) = (32, \frac{1}{5})$. In this way a constraint is kept for at least 32 iterations and dropped as soon as it was not violated in at least 20% of the iterations that it was in the pool, which appears to be a reasonable choice of parameters.

3.3.3 Cover Inequalities for Knapsack Sets

Suppose we are given a set of items N, a vector $\boldsymbol{a} \in \mathbb{N}^N$ that gives the size of each item in N, and a knapsack of size $b \in \mathbb{N}$. Let

$$P = \{\boldsymbol{x} \in [0,1]^N \mid \boldsymbol{a}^T \boldsymbol{x} \leq b\}.$$

The set

$$X = P \cap \{0,1\}^N$$

contains all incidence vectors of subsets of N that fit in the knapsack and is called a *knapsack set*. Knapsack sets are of interest to us because they appear as a relaxation of the problem we discuss in Chapter 6. The set $\text{conv}(X)$ is the *knapsack polytope*, which has been studied since the mid-seventies by Padberg [90], Balas [8], Wolsey [122], Hammer, Johnson, and Peled [58], Crowder, Johnson and Padberg [31], and Weismantel [120]. Here we follow the exposition of Wolsey [125].

A *cover* is a set $C \subseteq N$ such that $\boldsymbol{a}(C) > b$. Let C be a cover. A *cover inequality* is an inequality of the form

$$\boldsymbol{x}(C) \leq |C| - 1, \tag{3.8}$$

Cover inequalities are valid inequalities for the knapsack polytope. We say that C is *minimal* if every proper subset $C' \subset C$ has $\boldsymbol{a}(C') \leq b$. A minimal cover inequality is facet-defining if $C = N$.

A Greedy Heuristic for Searching Cover Inequalities

Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$. To find a cover inequality we follow the approach presented by Crowder, Johnson and Padberg [31]. Let

$B \subseteq N$ be the fractional support of $\boldsymbol{x}^*$, and let $U \subseteq N \setminus B$ be the maximal set of items with $\boldsymbol{x}^*_U = \mathbf{1}$. To separate a violated cover inequality, we focus on the set of items B with respect to a knapsack of size $b' = b - \boldsymbol{a}(U)$. Note that the cover inequality (3.8) is equivalent to

$$|C| - \boldsymbol{x}(C) \geq 1.$$

Hence, there exists a cover $C \subseteq B$ with respect to a knapsack of size b' that induces a violated cover inequality in $\boldsymbol{x}^*$ if and only if there exists a set $C \subseteq B$ with $\boldsymbol{a}(C) > b'$ that satisfies

$$|C| - \boldsymbol{x}^*(C) < 1,$$

which is the case if

$$C^* = \arg\min_{C \subseteq B} \left\{ \textstyle\sum_{i \in C}(1 - x^*_i) \mid \boldsymbol{a}(C) > b' \right\}$$

satisfies $|C^*| - \boldsymbol{x}^*(C^*) < 1$.

The cover is computed using a greedy heuristic that first sorts the items in B in non-decreasing order of $(1 - x^*_i)/a_i$. Let $B = \{i_1, \ldots, i_{|B|}\}$ be the resulting ordering. The greedy algorithm proceeds with two stages. In the first stage it finds a cover, and in the second stage it turns it into a minimal cover. Finding a cover is done by determining the smallest index k such that the items $\{i_1, \ldots, i_k\}$ are a cover with respect to a knapsack of size b', i.e., such that $\boldsymbol{a}(\{i_1, \ldots, i_k\}) > b'$. Let k be this smallest index, and let $C = \{i_1, \ldots, i_k\}$. The second stage works in iterations, numbered $j-1, \ldots, 1$. In iteration l, we check whether $C \setminus \{i_l\}$ has $\boldsymbol{a}(C \setminus \{i_l\}) > b'$ and if so, we replace C by $C \setminus \{i_l\}$. Upon termination of the second stage the set C is a minimal cover, and the heuristic returns it. The heuristic can be implemented in $O(|B| \log |B|)$ time.

The cover C that is returned by the heuristic need not be violated. Whether it is violated or not, we will try to strengthen it using the lifting algorithm that is the subject of the next section. If a cover inequality is violated after lifting, we report it.

3.3.4 Lifted Cover Inequalities for Knapsack Sets

Padberg [90], and Wolsey [122, 123] studied *sequential lifting* of valid inequalities. Sequential lifting of valid inequalities for zero-one integer programming problems relies on the following theorem (see also Nemhauser and Wolsey [88, Chapter II.2, Propositions 1.1 and 1.2]):

Theorem 3.3. *Let $S \subseteq \{0,1\}^n$. For $k \in \{0,1\}$, let $S^k = S \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_1 = k\}$.*

(i) Suppose

$$\sum_{j=2}^{n} \pi_j x_j \leq \pi_0 \tag{3.9}$$

is valid for S^0. *If* $S^1 = \varnothing$, *then* $x_1 \leq 0$ *is valid for* S. *If* $S^1 \neq \varnothing$, *then*

$$\sum_{j=1}^{n} \pi_j x_j \leq \pi_0 \tag{3.10}$$

is valid for S *for any* $\pi_1 \leq \pi_0 - \max_{\boldsymbol{x} \in S^1} \{\sum_{j=2}^{n} \pi_j x_j\}$. *If (3.9) defines a face of* $\operatorname{conv}(S^0)$ *of dimension* k, *and* π_1 *is chosen maximal, then (3.10) defines a face of* $\operatorname{conv}(S)$ *of dimension* $k+1$.

(ii) Suppose that (3.9) is valid for S^1. *If* $S^0 = \varnothing$, *then* $x_1 \geq 1$ *is valid for* S. *If* $S^0 \neq \varnothing$, *then*

$$\sum_{j=1}^{n} \pi_j x_j \leq \pi_0 + \pi_1 \tag{3.11}$$

is valid for S *for any* $\pi_1 \geq \max_{\boldsymbol{x} \in S^0} \{\sum_{j=2}^{n} \pi_j x_j\} - \pi_0$. *If (3.9) defines a face of* $\operatorname{conv}(S^1)$ *of dimension* k, *and* π_1 *is chosen minimal, then (3.11) defines a face of* $\operatorname{conv}(S)$ *of dimension* $k+1$.

Theorem 3.3 is a special case of the theorem given by Wolsey [123, Theorem 1] for general integer programming problems.

Consider again the sets P and X defined by a set of items N, a vector $\boldsymbol{a} \in \mathbb{N}^N$ that gives the size of each item, and a knapsack of size b as in Section 3.3.3. Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$ and that we have found a cover inequality

$$\boldsymbol{x}(C) \leq |C| - 1 \tag{3.12}$$

that is valid for the set

$$\big\{\boldsymbol{x} \in \{0,1\}^N \mid \textstyle\sum_{j \in C} a_j x_j \leq b - \boldsymbol{a}(U)\big\},$$

using the separation algorithm described in Section 3.3.3, where U is again the maximal set of items such that $\boldsymbol{x}^*_U = \mathbf{1}$. Van Roy and Wolsey [113] give an algorithm that iteratively applies Theorem 3.3 starting from (3.12) to obtain a lifted cover inequality of the form

$$\boldsymbol{x}(C) + \sum_{j \in D} \alpha_j x_j \leq |C| - 1 + \sum_{j \in U} \alpha_j (1 - x_j),$$

where C, D and U are mutually disjoint sets of items. In each iteration, one coefficient α_j is computed using a dynamic-programming algorithm for the knapsack problem (see e.g. Nemhauser and Wolsey [88, Chapter II.6, Proposition 1.6]). Gu, Nemhauser and Savelsbergh [57] discuss modern implementations of these techniques.

3.3.5 Mod-k Inequalities for Integer Programming

Suppose we are given natural numbers m, n, a matrix $A \in \mathbb{Z}^{m\times n}$, and a vector $\boldsymbol{b} \in \mathbb{Z}^m$. Let

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid A\boldsymbol{x} \leq \boldsymbol{b}\},$$

and let

$$X = P \cap \mathbb{Z}^n.$$

A *Chvátal-Gomory cut* is a valid inequality for $\text{conv}(X)$ of the form

$$\boldsymbol{\lambda}^T A\boldsymbol{x} \leq \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor, \tag{3.13}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m_+$ satisfies $\boldsymbol{\lambda}^T A \in \mathbb{Z}^n$. If $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$ for any $k \geq 2$, then the Chvátal-Gomory cut defined by $\boldsymbol{\lambda}$ is called a *mod-k cut*. Let $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$. Given $\boldsymbol{x}^* \in P$, the violation achieved by $\boldsymbol{x}^*$ of the mod-k cut defined by $\boldsymbol{\lambda}$ is given by

$$\boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor.$$

Since $\boldsymbol{\lambda}^T A\boldsymbol{x} \leq \boldsymbol{\lambda}^T \boldsymbol{b}$ is satisfied by $\boldsymbol{x}^*$, the maximal violation of a mod-k cut that can be obtained by $\boldsymbol{x}^*$ is $(k-1)/k$. Mod-k cuts for which $\boldsymbol{x}^*$ achieves this violation are called *maximally violated* by $\boldsymbol{x}^*$. The separation of mod-2 cuts was studied by Caprara and Fischetti [22]. The separation of mod-k cuts as treated in this section is by Caprara, Fischetti, and Letchford [23, 24]. The study of mod-k cuts is motivated by the fact that several well-known classes of valid inequalities for which no efficient separation algorithms were known, can be interpreted as mod-k cuts. This includes the class of comb inequalities of the travelling salesman problem.

Suppose we are given a fractional solution $\boldsymbol{x}^* \in P$. There exists a mod-k cut that is violated by $\boldsymbol{x}^*$ if and only if there exists $\boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m$ with

$$\boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor > 0 \text{ and } \boldsymbol{\lambda}^T A \in \mathbb{Z}^n,$$

which is the case if and only if

$$\zeta = \max_{\boldsymbol{\lambda}} \left\{ \boldsymbol{\lambda}^T A\boldsymbol{x}^* - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor \mid \boldsymbol{\lambda}^T A \in \mathbb{Z}^n, \boldsymbol{\lambda} \in \{0, 1/k, \ldots, (k-1)/k\}^m \right\} \tag{3.14}$$

satisfies $\zeta > 0$. Let $\boldsymbol{s}^* = \boldsymbol{b} - A\boldsymbol{x}^*$ denote the slack of $\boldsymbol{x}^*$. By substituting $A\boldsymbol{x}^* = \boldsymbol{b} - \boldsymbol{s}^*$, $\boldsymbol{\lambda}^T \boldsymbol{b} - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor = \theta/k$, and $\boldsymbol{\lambda} = \frac{1}{k}\boldsymbol{\mu}$ in (3.14) and observing that $\boldsymbol{\lambda}^T \boldsymbol{b} - \lfloor \boldsymbol{\lambda}^T \boldsymbol{b} \rfloor = \theta/k$ if and only if $\boldsymbol{b}^T \boldsymbol{\mu} \equiv \theta \pmod{k}$ and that $\boldsymbol{\lambda}^T A \in \mathbb{Z}^n$ if and only if $A^T \boldsymbol{\mu} \equiv \boldsymbol{0} \pmod{k}$, we obtain the equivalent formulation

$$\zeta = \frac{1}{k} \max_{(\boldsymbol{\mu}, \theta)} \quad \theta - (\boldsymbol{s}^*)^T \boldsymbol{\mu} \tag{3.15a}$$

$$\text{subject to} \quad A^T \boldsymbol{\mu} \equiv \boldsymbol{0} \pmod{k}, \tag{3.15b}$$

$$\boldsymbol{b}^T \boldsymbol{\mu} \equiv \theta \pmod{k}, \tag{3.15c}$$

$$(\boldsymbol{\mu}, \theta) \in \{0, \ldots, k-1\}^{m+1}. \tag{3.15d}$$

Caprara *et al.* show that (3.15) is $\mathcal{NP}$-hard using a reduction from the maximum cut problem. Furthermore, they show that the problem is polynomially solvable if we restrict ourselves to maximally violated mod-k inequalities. If $(\boldsymbol{\mu}^*, \theta^*)$ is an optimal solution to (3.15), then $\boldsymbol{\lambda} = \frac{1}{k}\boldsymbol{\mu}^*$ defines a mod-k cut that is violated by $\boldsymbol{x}^*$ if $\zeta > 0$, and ζ is the violation obtained by $\boldsymbol{x}^*$. Let $I = \{i \in \{1, \ldots, m\} \mid s_i^* = 0\}$ denote the set of row indices that have a slack equal to zero with respect to $\boldsymbol{x}^*$. From the objective function of (3.15) it is immediately clear that $(\boldsymbol{\mu}^*, \theta^*)$ defines a maximally violated mod-k cut if and only if $\theta^* = k - 1$ and $\mu_i^* = 0$ for all i with $s_i^* > 0$. From this it follows that a maximally violated mod-k exists if and only if the following system of mod-k congruences has a solution:

$$A_I^T \boldsymbol{\mu} \equiv \mathbf{0} \pmod{k}, \tag{3.16a}$$

$$b_I^T \boldsymbol{\mu} \equiv k - 1 \pmod{k}, \tag{3.16b}$$

$$\boldsymbol{\mu} \in \{0, \ldots, k-1\}^I. \tag{3.16c}$$

Determining feasibility of (3.16) and finding a solution to it if it is feasible can be done in $O(mn \min(m, n))$ time if k is prime using standard Gaussian elimination in $GF(k)$. Moreover, they show that the existence of a maximally violated mod-k cut for any non-prime value of k implies the existence of a maximally violated mod-l cut for every prime factor l of k. Therefore, we can restrict our attention to the separation of maximally violated mod-k cuts for which k is prime.

In our implementation the system of mod-k congruences (3.16) is solved by computing an LU-factorisation (see e.g. Chvátal [28]) of a sub-matrix of (3.16a)–(3.16b) that has full rank using arithmetic in $GF(k)$ (see e.g. Cormen *et al.* [29]), and checking the solution against the remaining constraints. To preserve the sparsity of the matrix we use a greedy heuristic that selects the next pivot from the sparsest remaining non-zero row. Our implementation is preliminary in that there are more sophisticated heuristics to preserve sparsity, and in that we compute the factorisation from scratch over and over again.

3.4 Branch-and-Price

Now that we have seen how to adapt the LP-based branch-and-bound algorithm for integer linear programming formulations with a large number of constraints, we consider how to adapt the LP-based branch-and-bound algorithm in such a way that it is no longer necessary to include all variables in the formulation that is passed to the LP solver. This allows us to use formulations that have a large, or even exponential number of variables. Problems that give rise to such formulations have been studied since the nineteen eighties. These studies include the ones by Desrosiers, Soumis, and Desrochers [40], Desrochers, Desrosiers, and Solomon [38], Barnhart, Johnson, Nemhauser, Savelsbergh, and Vance [14], Vanderbeck and Wolsey [115], and Vanderbeck [114].

3.4.1 LP Column Generation

Suppose we are given a matrix

$$A = \begin{pmatrix} a_{I_0} \\ a_{I \setminus I_0} \end{pmatrix} \in \mathbb{Z}^{m \times n},$$

and a vector

$$\boldsymbol{b} = \begin{pmatrix} \boldsymbol{b}_{I_0} \\ \boldsymbol{b}_{I \setminus I_0} \end{pmatrix} \in \mathbb{Z}^{m},$$

where m, n are natural numbers, and $I_0 \subseteq I = \{1, \dots, m\}$ are sets of row indices. Consider the linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \mid a_{I_0} \boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in \operatorname{conv}(X)\}, \tag{3.17}$$

where $\boldsymbol{c} \in \mathbb{Z}^n$ is the cost vector, and X is defined by the sub-matrix $a_{I \setminus I_0}$ of A and bound vectors $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{Z}^n$ as follows:

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I \setminus I_0} \boldsymbol{x} = \boldsymbol{b}_{I \setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

We will reformulate (3.17) using *Dantzig-Wolfe decomposition* [36] to obtain a formulation with a possibly exponential number of variables. Suppose that $a_{I \setminus I_0}$ has a block-diagonal structure, and let $I_1, \dots, I_k \subseteq I \setminus I_0$ and $J_1, \dots, J_k \subseteq \{1, \dots, n\}$ be a partitioning of the row and variable indices of $a_{I \setminus I_0}$, respectively, such that

$$A = \begin{pmatrix} A^1 & A^2 & \dots & A^k \\ A_{I_1 J_1} & & & \\ & A_{I_2 J_2} & & \\ & & \dots & \\ & & & A_{I_k J_k} \end{pmatrix}, \tag{3.18}$$

where $A^i = A_{I_0 J_i}$. For each $i \in \{1, \dots, k\}$, let

$$P_i = \{\boldsymbol{x} \in \mathbb{R}^{J_i} \mid A_{I_i J_i} \boldsymbol{x} = \boldsymbol{b}_{I_i}, \boldsymbol{l}_{J_i} \leq \boldsymbol{x} \leq \boldsymbol{u}_{J_i}\},$$

and

$$X_i = P_i \cap \mathbb{Z}^{J_i}.$$

To simplify the following discussion, assume that X_i is finite for each $i \in \{1, \dots, k\}$.

Note that any fractional solution $\boldsymbol{x} \in \operatorname{conv}(X)$ satisfies $\boldsymbol{x}_{J_i} \in \operatorname{conv}(X_i)$, so $\boldsymbol{x}_{J_i}$ can be expressed as a convex combination of the elements of X_i:

$$\boldsymbol{x}_{J_i} = \sum_{\boldsymbol{x}' \in X_i} \boldsymbol{x}' \lambda_{\boldsymbol{x}'}, \text{ with } \boldsymbol{\lambda}(X_i) = 1, \text{ and } \boldsymbol{\lambda} \geq \boldsymbol{0}.$$

Now substituting for $\boldsymbol{x}_{J_i}$ in (3.17) yields an alternative linear programming formulation to (3.17) with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$z^*_{(\boldsymbol{l},\boldsymbol{u})} = \max \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in X_i} (\boldsymbol{c}^T_{J_i}\boldsymbol{x})\lambda_{\boldsymbol{x}} \tag{3.19a}$$

$$\text{subject to} \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in X_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, \tag{3.19b}$$

$$\boldsymbol{\lambda}(X_i) = 1, \qquad \forall i \in \{1,\dots,k\}, \tag{3.19c}$$

$$\boldsymbol{\lambda} \geq \boldsymbol{0}. \tag{3.19d}$$

The linear programming problem (3.19) is referred to as the *master problem* that we want to solve. The subscript $(\boldsymbol{l},\boldsymbol{u})$ is added to the optimal value z^* to stress the implicit dependence on the lower and upper bounds $\boldsymbol{l}$ and $\boldsymbol{u}$.

Due to the large number of $\boldsymbol{\lambda}$-variables it is difficult in general to solve (3.19) directly. Instead, we assume that we have at our disposal a relatively small subset $\bar{X}_i \subseteq X_i$ for each $i \in \{1,\dots,k\}$. In that case we can solve the following *restricted* master problem:

$$\max \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in \bar{X}_i} (\boldsymbol{c}^T_{J_i}\boldsymbol{x})\lambda_{\boldsymbol{x}} \tag{3.20a}$$

$$\text{subject to} \quad \sum_{i=1}^{k} \sum_{\boldsymbol{x}\in \bar{X}_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, \tag{3.20b}$$

$$\boldsymbol{\lambda}(\bar{X}_i) = 1, \qquad \forall i \in \{1,\dots,k\}, \tag{3.20c}$$

$$\boldsymbol{\lambda}_{\bar{X}_i} \geq \boldsymbol{0}, \qquad \forall i \in \{1,\dots,k\}. \tag{3.20d}$$

Suppose that the restricted master problem is feasible and that we have an optimal primal-dual pair $(\boldsymbol{\lambda}, (\boldsymbol{\pi}, \boldsymbol{\mu}))$ to it, where $\boldsymbol{\pi} \in \mathbb{R}^{I_0}$ is associated with the constraints (3.20b), and $\boldsymbol{\mu} \in \mathbb{R}^k$ is associated with the constraints (3.20c). Obviously $\boldsymbol{\lambda}$ is a feasible solution to the master problem, and $(\boldsymbol{\lambda}, (\boldsymbol{\pi}, \boldsymbol{\mu}))$ is an optimal primal-dual pair to the master problem if it satisfies the reduced cost optimality conditions of Theorem 2.2. This is the case if for all $i \in \{1,\dots,k\}$, for all $\boldsymbol{x} \in X_i$, we have

$$\boldsymbol{c}^T_{J_i}\boldsymbol{x} - \boldsymbol{\pi}^T A^i\boldsymbol{x} - \mu_i \leq 0,$$

which is the case if

$$\zeta_i = \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T\boldsymbol{x} \mid \boldsymbol{x} \in X_i\} \tag{3.21}$$

satisfies $\zeta_i \leq \mu_i$, where $\boldsymbol{c}^{\boldsymbol{\pi}} \in \mathbb{R}^n$ is defined by $\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i} = \boldsymbol{c}_{J_i} - (\boldsymbol{\pi}^T A^i)^T$ for all $i \in \{1,\dots,k\}$. Problem (3.21) is called *pricing problem* i. If $\zeta_i > \mu_i$ for some $i \in \{1,\dots,k\}$, then the corresponding optimal solution $\boldsymbol{x}^*$ to pricing problem i is added to $\bar{X}_i$, and the master problem is resolved. A *column generation*

algorithm iteratively solves a restricted master problem, and pricing problems for all $i \in \{1, \ldots, k\}$, adding $\boldsymbol{\lambda}$-variables that violate the reduced cost optimality conditions of the master to the restricted master, until an optimal primal-dual pair to the master problem is found.

3.4.2 IP Column Generation

Now that we have seen how to apply Dantzig-Wolfe decomposition and column generation to linear programming problems, we will show how to adapt the LP-based branch-and-bound algorithm to integer linear programming problems that have a similar structure. For simplicity we concentrate on linear integer programming formulations that do not contain continuous variables. Consider the integer linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \mid a_{I_0}\boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in X\}, \tag{3.22}$$

where $\boldsymbol{b}, \boldsymbol{c}, A$ and I_0 are as in the previous section, and X is again defined in terms of $a_{I\setminus I_0}, \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l}$ and $\boldsymbol{u}$ by

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I\setminus I_0}\boldsymbol{x} = \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

Note that (3.17) is an LP relaxation of (3.22).

Assuming that A is of the form (3.18) and after applying Dantzig-Wolfe decomposition to (3.22) we obtain the following equivalent integer linear programming formulation with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$\begin{aligned}
\max \quad & \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} (\boldsymbol{c}_{J_i}^T \boldsymbol{x})\lambda_{\boldsymbol{x}} && && \text{(3.23a)}\\
\text{subject to} \quad & \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} (A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, && && \text{(3.23b)}\\
& \boldsymbol{\lambda}(X_i) = 1, && \forall i \in \{1, \ldots, k\}, && \text{(3.23c)}\\
& \boldsymbol{\lambda}_{X_i} \in \{0,1\}^{X_i}, && \forall i \in \{1, \ldots, k\}, && \text{(3.23d)}
\end{aligned}$$

where $X_i \subseteq \mathbb{Z}^{J_i}$, and A^i are defined as before. Problem (3.23) is called the *IP master problem*, and in the context of IP column generation its LP relaxation (3.19) is called the *LP master problem*.

A *branch-and-price* algorithm for solving (3.22) is derived from the LP-based branch-and-bound algorithm as follows. The algorithm maintains a restricted LP master problem, that we assume to be feasible. The situation that the restricted LP master problem is infeasible is discussed in Section 3.4.4. As before, with each iteration j of the branch-and-price algorithm we associate a

unique set of bounds $\boldsymbol{l}^j, \boldsymbol{u}^j \in \mathbb{Z}^n$. In iteration j, the branch-and-price algorithm solves the LP master problem obtained from (3.17) by imposing the bounds $\boldsymbol{l}^j, \boldsymbol{u}^j$ instead of $\boldsymbol{l}, \boldsymbol{u}$. This is done by column generation. The formulation is obtained by imposing an upper bound of zero on all variables $\lambda_{\boldsymbol{x}}$ for which $\boldsymbol{l}^j_{J_i} \not\leq \boldsymbol{x}_{J_i}$ or $\boldsymbol{x}_{J_i} \not\leq \boldsymbol{u}^j_{J_i}$, where $\boldsymbol{x}_{J_i} \in \bar{X}_i, i \in \{1, \ldots, k\}$. The resulting pricing problems are of the form

$$\zeta_i = \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T \boldsymbol{x} \mid \boldsymbol{x} \in X_i\}, \tag{3.24}$$

where for all $i \in \{1, \ldots, k\}$

$$X_i = P_i \cap \mathbb{Z}^{J_i},$$

and

$$P_i = \{\boldsymbol{x} \in \mathbb{R}^{J_i} \mid a_{I_i}\boldsymbol{x} = \boldsymbol{b}_{I_i}, \boldsymbol{l}^j_{J_i} \leq \boldsymbol{x} \leq \boldsymbol{u}^j_{J_i}\}.$$

If the pricing problems (3.24) are easy to solve, then we can use the column generation algorithm from Section 3.4.1 to solve the LP master problem in each iteration of the branch-and-bound algorithm. However, it may very well be that the pricing problems (3.24) are $\mathcal{NP}$-hard combinatorial optimisation problems. This is the case, for example, in Chapter 7 of this thesis, where we apply Dantzig-Wolfe decomposition to a vehicle routing problem, and the pricing problems are the problems of how to route and load the individual vehicles. Under such circumstances it would be rather optimistic to believe that we can solve the LP master problems to optimality by column generation even for medium size problems, as this would require us to solve numerous $\mathcal{NP}$-hard pricing problems in each iteration of the branch-and-price algorithm.

If we have an efficient heuristic for finding good feasible solutions to the pricing problems, we can use this heuristic to generate new columns instead of spending a lot of time in optimising the pricing problems. However, this approach gives us a new problem, namely, that a restricted LP master problem itself is not a relaxation of the IP master problem (3.23). The value of the restricted LP master problem is only an upper bound for the value of the IP master problem if the optimal LP solution to the restricted LP master problem is also optimal to the LP master problem. This is not necessarily the case when we generate columns by solving the pricing problems using primal heuristics as suggested above.

A solution to this problem was observed by Vanderbeck and Wolsey [115], and was successfully applied by Vanderbeck [114]. Focus on iteration $j \geq 1$ of the branch-and-price algorithm. Suppose we have solved a restricted master LP associated with iteration j that was feasible, and let $(\boldsymbol{\pi}, \boldsymbol{\mu}) \in \mathbb{R}^{I_1} \times \mathbb{R}^k$ be a dual solution corresponding to an optimal basis B. Vanderbeck and Wolsey observe that if we have

$$\bar{\zeta}_i \geq \max\{(\boldsymbol{c}^{\boldsymbol{\pi}}_{J_i})^T \boldsymbol{x} \mid \boldsymbol{x} \in X_i\}, \tag{3.25}$$

```
columnGenerationSolver(c', A', b', l, u, pricing)      // c', A' given implicitly,
{                                                      // returns λ*, z̄ ≥ z*_(l,u)
    let X be a subset of the column indices of A'; // e.g. X := ∅;
    do { solve LP z := max{(c'_X)^T λ_X | A'_X λ_X = b', λ_X ≥ 0}
        if LP infeasible {
            (X, feasible) := makeFeasible(A', b', l, u, X, pricing)
            if feasible {
                solve LP z := max{(c'_X)^T λ_X | A'_X λ_X = b', λ_X ≥ 0}; }
            else { return infeasible; }
        }
        let π be an optimal LP dual;
        (X', ζ, optimal) := pricing(π, l, u);
        z̄ := min(z̄, (b')^T (π + (0, ζ)^T));
        X := X ∪ X';
    } while X' ≠ ∅ and ζ ≠ 0;
    if optimal or ζ = 0 { z̄ := z; }
    let λ*_X be an optimal LP solution;
    return (λ*_X, z̄);
}
```

Algorithm 3.2: Solving LP by Column Generation in Branch-and-Price

where X_i is defined as before for all $i \in \{1, \ldots, k\}$, then the vector $(\boldsymbol{\pi}, \boldsymbol{\mu}+\bar{\boldsymbol{\zeta}})$ is a feasible solution to the dual linear program of the LP master problem associated with iteration j. From this, by Theorem 2.3 it follows that

$$z^*_{(\boldsymbol{l}^j, \boldsymbol{u}^j)} \leq \boldsymbol{b}^T \boldsymbol{\pi} + \mathbf{1}^T (\boldsymbol{\mu} + \bar{\boldsymbol{\zeta}}),$$

and that $\boldsymbol{\lambda}^*$ and $(\boldsymbol{\pi}, \boldsymbol{\mu})$ are optimal solutions to the primal and dual LP master problem, respectively, if $\bar{\boldsymbol{\zeta}} = \mathbf{0}$. Note that a value of $\bar{\boldsymbol{\zeta}}$ can be obtained by solving relaxations of the pricing problems, instead of the pricing problems themselves.

The IP master problem (3.23) can be restated as

$$\begin{aligned} \max \quad & (\boldsymbol{c}')^T \boldsymbol{\lambda} && \text{(3.26a)} \\ \text{subject to} \quad & A' \boldsymbol{\lambda} = \boldsymbol{b}, && \text{(3.26b)} \\ & \boldsymbol{\lambda} \geq \mathbf{0}, \text{ integer.} && \text{(3.26c)} \end{aligned}$$

Pseudo-code of the LP column generation algorithm that we employ in our branch-and-price code is given in Algorithm 3.2. Note that A' and $\boldsymbol{c}'$ are given implicitly, i.e., in such a way that we can compute $A'_{\boldsymbol{x}}$ and $c'_{\boldsymbol{x}}$ for each $\boldsymbol{x} \in \bigcup_{i=1}^k X_i$ once we have $\boldsymbol{x}$. The pricing problems depend on the original problem and are problem specific. Therefore, the column generation algorithm assumes as input a function pricing that it uses to solve the pricing problems. This function is of the form

$$\text{pricing} : \mathbb{R}^{m+k} \times \mathbb{R}^n \times \mathbb{R}^n \to 2^{\bigcup_{i=1}^k X_i} \times \mathbb{R}^k \times \{0, 1\},$$

and maps each triple $(\boldsymbol{\pi}, \boldsymbol{l}, \boldsymbol{u})$ to a triple $(X', \boldsymbol{\zeta}, f)$, where $\boldsymbol{\pi}$ is an optimal dual to the restricted LP master, $\boldsymbol{l}, \boldsymbol{u}$ are bounds, X' is a set of column indices that correspond to $\boldsymbol{\lambda}$ variables that violated the reduced cost optimality criteria of the LP master problem, $\boldsymbol{\zeta}$ is an upper bound on the value of the pricing problems, and $f = 1$ if the pricing problems were solved to optimality, and $f = 0$ otherwise. Furthermore, the algorithm makes use of a function makeFeasible. Given a subset of the column indices X', bound vectors $\boldsymbol{l}, \boldsymbol{u}$, and the pricing function, makeFeasible determines whether there exists a set of column indices $X'' \supseteq X'$ such that $A'_{X''} = \boldsymbol{b}'$ is feasible. The function makeFeasible returns a pair (X'', f), where f is 1 if $A'_{X''} = \boldsymbol{b}'$ and 0 otherwise.

3.4.3 The Branching Scheme

After we have solved the LP relaxation in some iteration i of the branch-and-price algorithm, we have at our disposal a vector $\boldsymbol{\lambda}^*$ that is a feasible solution to the LP master problem, and what we want to have is a feasible solution to our original problem (3.22). It is undesirable to branch on the $\boldsymbol{\lambda}$-variables, as this changes the structure of the pricing problems in all nodes that are not the root node of the branch-and-bound tree. Indeed, we would have to prevent $\boldsymbol{x}$ for which $\lambda_{\boldsymbol{x}}$ has been set to either zero or one to be regenerated by a pricing problem. For the same reason we do not set $\boldsymbol{\lambda}$-variables based on reduced cost criteria.

Several alternative branching schemes are reported in the literature. Here, we discuss a scheme that is known as branching *on original variables* (see e.g. Barnart *et al.* [14]). In this scheme, we compute $\boldsymbol{x}^* \in \mathbb{R}^n$ given by

$$\boldsymbol{x}^* = \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in X_i} \boldsymbol{x} \lambda^*_{\boldsymbol{x}},$$

which is the same solution as $\boldsymbol{\lambda}^*$ but restated in terms of the variables of the original integer programming problem (3.22). If $\boldsymbol{x}^* \in \mathbb{Z}^n$, we are done. Otherwise we can select an index $j^* \in \{1, \ldots, n\}$ such that $x^*_{j^*}$ is fractional and partition the feasible region by enforcing lower and upper bounds $(\boldsymbol{l}^i, \tilde{\boldsymbol{u}}^i)$ on one branch and $(\tilde{\boldsymbol{l}}^i, \boldsymbol{u}^i)$ on the other branch, where $\tilde{\boldsymbol{l}}^i, \tilde{\boldsymbol{u}}^i \in \mathbb{Z}^n$ are given by

$$\tilde{l}^i_j = \begin{cases} \lceil x^*_j \rceil, & \text{if } j = j^*, \\ l^i_j & \text{otherwise,} \end{cases} \qquad \text{and} \qquad \tilde{u}^i_j = \begin{cases} \lfloor x^*_j \rfloor, & \text{if } j = j^*, \\ u^i_j & \text{otherwise.} \end{cases}$$

In our implementation the index j^* is selected using the Padberg-Rinaldi rule (see Section 3.2.4).

We end this section by remarking that for the correctness of the branch-and-price algorithm it is necessary to branch if $\boldsymbol{x}^*$ is integer but one does not have a proof that $\boldsymbol{x}^*$ is optimal. This can be done in a similar way as above.

```
makeFeasible(A', b', l, u, X, pricing)        // returns (X̃, f): f = 1 iff
{                                             // A'_X̃ λ_X̃ = b', λ_X̃ ≥ 0 is feasible
    c := 0;                                   // implicitly used by pricing
    do { solve z := max{−s | A'_X λ_X + b' s = b', λ ≥ 0, s ≥ 0};
         if z = 0 { return (X, 1); }
         let π be an optimal dual solution;
         (X', ζ) := pricing(π, l, u);         // solve to optimality
         X := X ∪ X';
    } while X ≠ ∅ and ζ ≠ 0;
    return (X, 0);
}
```

Algorithm 3.3: Searching a Feasible Set of Columns

3.4.4 Infeasible Nodes

Consider iteration $i > 1$ of the branch-and-price algorithm, and observe that there is no reason to assume that the restricted LP master problem is feasible after we impose the bounds $(\boldsymbol{l}^i, \boldsymbol{u}^i)$. Moreover, the infeasibility of the restricted LP master problem does not imply the infeasibility of the LP master problem associated with iteration i as it contains only a subset of the $\boldsymbol{\lambda}$-variables. It follows that our branch-and-price algorithm is not correct unless we have a way of either proving that the LP master problem associated with iteration i is infeasible, or finding a feasible solution to it.

A possible solution here is to use the so-called "big M" method that is used for finding an initial feasible basis in linear programming. In this method one adds slack variables that have a symbolic cost coefficient that is interpreted as a large negative value (and usually denoted by M). Unfortunately, this would require us to write subroutines for solving the pricing problems that can handle symbolic values, which is undesirable. A solution suggested by Barnhart *et al.* [14] is to use slack variables with real-valued negative cost coefficients instead. For such a scheme to be correct one needs a lower bound on the cost of the slack variables that suffices to prove infeasibility. Moreover, this lower bound should be of such a small magnitude in absolute value that the pricing algorithms can manipulate the resulting dual solutions without numerical problems.

If the objective function is zero, then any negative cost coefficient suffices to prove infeasibility. This way, one obtains a well defined two-phase approach for LP column generation that is similar to the two-phase approach that is used for linear programming (see e.g. Chvátal [28, Chapter 8]). Suppose we have an infeasible restricted LP master problem of the form

$$\begin{aligned} \max \quad & (\boldsymbol{c}'_X)^T \boldsymbol{\lambda}_X && \text{(3.27a)} \\ \text{subject to} \quad & A'_X \boldsymbol{\lambda}_X = \boldsymbol{b}', && \text{(3.27b)} \\ & \boldsymbol{\lambda}_X \geq \boldsymbol{0}. && \text{(3.27c)} \end{aligned}$$

To find a feasible solution to (3.27), we construct an auxiliary master linear program by replacing the cost vector $\boldsymbol{c}$ by the zero vector and adding a single artificial variable $s \in \mathbb{R}$ with cost -1, and column $\boldsymbol{b}'$:

$$\begin{aligned} \max \quad & z(\boldsymbol{\lambda}, s) = -s && \text{(3.28a)} \\ \text{subject to} \quad & A'\boldsymbol{\lambda} + \boldsymbol{b}'s = \boldsymbol{b}', && \text{(3.28b)} \\ & \boldsymbol{\lambda} \geq \boldsymbol{0}, s \geq 0. && \text{(3.28c)} \end{aligned}$$

Note that $(\boldsymbol{\lambda}, s) := (\boldsymbol{0}, 1)$ is a feasible solution to (3.28) with value -1. Now, if $\boldsymbol{\lambda}^*$ is a feasible solution to (3.27) then $(\boldsymbol{\lambda}^*, 0)$ is a feasible solution to (3.28) with value 0. Conversely, if $(\boldsymbol{\lambda}^*, s^*)$ is a feasible solution to (3.28) with value 0 then $s^* = 0$, so $\boldsymbol{\lambda}^*$ is a feasible solution to (3.27). Pseudo-code for a procedure that solves (3.28) by column generation to either prove infeasibility of the LP master problem, or to find a set of column indices X that defines a feasible LP master problem can be found in Algorithm 3.3. Note that in order to prove that the LP master problem associated with iteration i is infeasible, it is necessary to solve the pricing problems to optimality.

3.5 Branch-Price-and-Cut

Consider again the integer linear programming problem

$$\max\{z(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \mid a_{I_0}\boldsymbol{x} = \boldsymbol{b}_{I_0}, \boldsymbol{x} \in X\}, \tag{3.29}$$

where $\boldsymbol{b}, \boldsymbol{c}, A$ and I_0 are as in the previous section, and X is again defined in terms of $a_{I\setminus I_0}, \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l}$ and $\boldsymbol{u}$ by

$$P = \{\boldsymbol{x} \in \mathbb{R}^n \mid a_{I\setminus I_0}\boldsymbol{x} = \boldsymbol{b}_{I\setminus I_0}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

and

$$X = P \cap \mathbb{Z}^n.$$

Assume that A is of the form (3.18). We again apply Dantzig-Wolfe decomposition, but this time we leave $\boldsymbol{x}_{J_1}$ aside and only reformulate $\boldsymbol{x}_{J_i}$ for $i \in \{2, \ldots, k\}$. In this way we obtain the following equivalent integer linear programming problem with a large but finite number of $\boldsymbol{\lambda}$-variables:

$$\begin{aligned} \max \quad & \boldsymbol{c}_{J_1}^T\boldsymbol{x}_{J_1} + \sum_{i=2}^{k}\sum_{\boldsymbol{x}\in X_i}(\boldsymbol{c}_{J_i}^T\boldsymbol{x})\lambda_{\boldsymbol{x}} && && \text{(3.30a)} \\ \text{subject to} \quad & A^1\boldsymbol{x}_{J_1} + \sum_{i=2}^{k}\sum_{\boldsymbol{x}\in X_i}(A^i\boldsymbol{x})\lambda_{\boldsymbol{x}} = \boldsymbol{b}_{I_0}, && && \text{(3.30b)} \\ & \boldsymbol{l}_{J_1} \leq \boldsymbol{x}_{J_1} \leq \boldsymbol{u}_{J_1},\ \boldsymbol{x}_{J_1} \text{ integer}, && && \text{(3.30c)} \\ & \boldsymbol{\lambda}(X_i) = 1, && \forall i \in \{2, \ldots, k\}, && \text{(3.30d)} \\ & \boldsymbol{\lambda}_{X_i} \in \{0, 1\}^{X_i}, && \forall i \in \{2, \ldots, k\}, && \text{(3.30e)} \end{aligned}$$

where $X_i \subseteq \mathbb{Z}^{J_i}$, and A_i are defined as before. The resulting model can be solved using the branch-and-price algorithm of the previous section, allowing for branching on the $\boldsymbol{x}_{J_1}$-variables as described in Section 3.2.4, and branching on the original $\boldsymbol{x}_{J_i}$-variables for $i \in \{2, \ldots, k\}$ as described in Section 3.4.3.

If we have one or more classes of valid inequalities for $\text{conv}(X_1)$ at our disposal, then we can add these to strengthen the LP formulation in each iteration of the branch-and-price algorithm. Furthermore, one may apply variable setting based on reduced cost criteria to the $\boldsymbol{x}_{J_1}$-variables whenever we have a proof that the bound obtained from the restricted LP master problem is valid for the LP master problem. We will call the resulting algorithm a *branch-price-and-cut* algorithm. A branch-price-and-cut algorithm applies a combined cutting plane and column generation algorithm to solve the LP relaxations in each iteration. In our implementation we alternate between rounds of column generation and separation until the termination criteria of both the column generation algorithm and the cutting plane algorithm have been satisfied in two consecutive rounds.