Math. Prog. Comp. (2018) 10:33–68
https://doi.org/10.1007/s12532-017-0122-5

FULL LENGTH PAPER

Branch-and-cut for linear programs with overlapping SOS1 constraints

Tobias Fischer[1] · **Marc E. Pfetsch**[2]

Received: 1 April 2015 / Accepted: 23 May 2017 / Published online: 13 June 2017
© Springer-Verlag GmbH Germany and The Mathematical Programming Society 2017

Abstract SOS1 constraints require that at most one of a given set of variables is nonzero. In this article, we investigate a branch-and-cut algorithm to solve linear programs with SOS1 constraints. We focus on the case in which the SOS1 constraints overlap. The corresponding conflict graph can algorithmically be exploited, for instance, for improved branching rules, preprocessing, primal heuristics, and cutting planes. In an extensive computational study, we evaluate the components of our implementation on instances for three different applications. We also demonstrate the effectiveness of this approach by comparing it to the solution of a mixed-integer programming formulation, if the variables appearing in SOS1 constraints ar bounded.

Keywords Complementarity constraints · Special ordered sets · Mixed-integer programming · Branch-and-cut · SOS1 branching · Bipartite branching

Mathematics Subject Classification 90C11 · 90C33

✉ Tobias Fischer
fischer@gsc.tu-darmstadt.de

Marc E. Pfetsch
pfetsch@mathematik.tu-darmstadt.de

1 Graduate School of Computational Engineering, TU Darmstadt, Dolivostraße 15, 64293 Darmstadt, Germany

2 Department of Mathematics, Research Group Optimization, TU Darmstadt, Dolivostraße 15, 64293 Darmstadt, Germany

1 Introduction

This article deals with optimization problems of the following form:

$$\text{(LPCC)} \qquad \begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^\top x \\ \text{s.t.} \quad & Ax = b, \\ & 0 \le x \le u, \\ & x_i \cdot x_j = 0 \quad \forall \{i, j\} \in E, \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $u \in (\mathbb{R} \cup \{\infty\})^n$. Moreover, $E \subseteq \binom{V}{2}$ with $V := \{1, \ldots, n\}$. This yields the so-called *conflict graph* $G = (V, E)$. We assume G to be simple and we identify variables by their nodes in G.

The conditions $x_i \cdot x_j = 0$, $\{i, j\} \in E$, are called *complementarity constraints* and the above problem consequently *Linear Program with Complementarity Constraints* (LPCC). Complementarity constraints form a special case of *Special Ordered Set of type 1* (SOS1) constraints, which are defined by cliques in G. Thus, for a set of variables $S \subseteq V$ forming a clique, i.e., $\{i, j\} \in E$ for every $i, j \in S$ with $i \neq j$, at most one variable is allowed to be nonzero. We note that the original name "special ordered set" may be misleading in our context, since there is no particular ordering in which variables are treated.

LPCCs are a common class of NP-hard (see, e.g., Ibaraki et al. [35]) mathematical optimization problems. The classical paper of Beale and Tomlin [10] of the early 1970s showed how to treat SOS1 constraints via branching. Most mixed-integer programming (MIP) solvers allow to handle SOS1 constraints in this way. LPCCs have many applications. These include communication systems (see, e.g., Gupta and Kumar [27]), logistics (see, e.g., Cao [17]), finance (see, e.g., Lin [40]), and scheduling (see, e.g., Baker and Coffman [8], Dowsland [18]).

In some of the applications of (LPCC), the conflict graph G consists of disjoint cliques, i.e., every variable $i \in V$ is contained in at most one SOS1 constraint. One example are KKT conditions of linear optimization problems with a quadratic objective, see, e.g., Hu et al. [31,32]. In other applications, however, the conflict graph G contains intersecting cliques, i.e., some of the SOS1 constraints overlap. In this article, we concentrate on this latter case. Our goal is to exploit the structure of the conflict graph G in order to derive and implement an efficient branch-and-cut solution approach.

As one main component we discuss new branching rules (Sect. 2). One way is to branch on the neighborhood of a single variable i, i.e., in one branch x_i is fixed to 0 and in the other its neighbors. This imitates standard 0/1-branching on a binary variable and can result in an unbalanced branch-and-bound tree. In contrast, the classical *SOS1 branching* strategy proposed by Beale and Tomlin ensures a certain balance between the branching nodes. We further develop this approach for the case of overlapping SOS1 constraints by considering complete bipartite subgraphs. We also investigate ways to strengthen the resulting subproblems by adding complementarity constraints. Finally, we consider different selection rules of different branching possibilities.

In order to obtain an efficient solution method for (LPCC), we then present several additional solver components. This includes presolving techniques (Sect. 3), primal heuristics (Sect. 4), and cutting planes (Sect. 5). In each case, we take ideas from the literature and adapt and extend them to the LPCC case.

We will demonstrate computationally in Sect. 6.2 that this yields an efficient branch-and-cut solver, which we implemented using SCIP [2,47]. We first introduce three applications of "overlapping" LPCCs and then examine the effectiveness of our solver for these applications.

It is important to note that if the variable bounds u are finite, (LPCC) can be reformulated as an MIP with the help of auxiliary binary variables, see Sect. 1.2. SOS1 constraints are then turned into set packing constraints and the wealth of solution techniques for MIPs can be used. Thus, ideas for MIPs with set packing constraints can be carried over to (LPCC) and conversely. Note, however, that the resulting MIP contains up to twice as many variables as (LPCC) and its effectiveness will depend on whether the bounds of the variables are tight. We demonstrate by computational experiments that for the three applications considered in this article, our branch-and-cut solver outperforms CPLEX 12.6.

1.1 Literature overview

Beale and Tomlin [10] were the first to explicitly introduce SOS1 constraints and SOS1 branching, which allows to handle SOS1 constraints in a branch-and-bound search; this was first implemented in the programming system UMPIRE (see Forrest et al. [24]). Much later, de Farias et al. [19–21] analyzed the polyhedral properties of "nonoverlapping" LPCCs and extended the approach of Beale and Tomlin to a branch-and-cut scheme with promising computational results. Their cutting planes are derived by a sequential lifting procedure of cover inequalities.

A further class of cutting planes are *disjunctive cuts*, which can be directly generated from the simplex tableau. Disjunctive cuts for SOS1 constraints have been investigated in the literature since the 1970s (see Owen [43]). In recent years, their effectiveness was confirmed by computational results: Audet et al. [7] incorporated disjunctive cuts in a branch-and-cut algorithm for bilevel problems. In addition, Júdice et al. [39] tested them on several problem classes such as bilevel or bilinear problems, which were reformulated as LPCCs using KKT conditions.

Moreover, complementarity conditions arising from KKT conditions for linear problems with quadratic objective are the subject of Hu et al. [31,32], who proposed a Benders type approach for solving the resulting LPCC. In their algorithm, they iteratively generate cutting planes, so-called point and ray cuts, which express feasibility and objective bound requirements in a combinatorial way.

LPCCs also form a special case of *mathematical programs with equilibrium constraints* (MPECs), which are often investigated in the context of Nash equilibria. There are many articles investigating solution approaches for finding local optima of MPECs and corresponding constraint qualifications; in the context of (LPCC) these methods provide primal heuristics. As one example, we mention the article of Hoheisel et al. [30] and the references therein.

The feasibility problems corresponding to LPCCs are also strongly related to the *linear complementarity problem* (LCP), see Murty [41] for an extensive study. Here, for a given square matrix $M \in \mathbb{R}^{n\times n}$ and a vector $q \in \mathbb{R}^n$, the goal is to find some vector $x \in \mathbb{R}^n_+$ satisfying the orthogonality constraint $x \perp (q+M\,x)$ and $q+M\,x \geq 0$. Introducing an auxiliary vector $w = q + Mx$, these orthogonality constraints can be enforced via complementarity constraints $x_i \cdot w_i = 0$, $i \in V$.

All of the references mentioned so far focus on nonoverlapping SOS1 constraints. For the first time, probably Benichou et al. [11] explicitly mentioned the appearance of overlapping SOS1 constraints. Later, Hummeltenberg [33] investigated a reformulation of the "overlapping" problem description with auxiliary binary variables, such that all the resulting SOS1 constraints are disjoint under certain conditions. Moreover, conflict graphs with overlapping cliques were considered for MIPs, see Atamtürk et al. [5,6] and Agra et al. [4].

To the best of our knowledge, the direct treatment of overlapping SOS1 constraints and the corresponding conflict graph has been widely unexplored so far.

1.2 Mixed-integer programming reformulation

In this section, we discuss the relation of (LPCC) to its reformulation as a mixed-integer program.

Before we start, we observe that the requirement of nonnegativity of the variables in (LPCC) is not restrictive: free variables x_i can be split into two nonnegative variables x_i^+ and x_i^- with $x_i = x_i^+ - x_i^-$. Complementarity constraints involving x_i are replaced by two constraints with x_i being replaced by x_i^+ and x_i^-. Furthermore, we may assume that the upper bounds u_i, $i \in V$, of the nonnegative variables are nonzero, since otherwise variables can be removed from the model.

If all bounds u_i, $i \in V$, are finite, (LPCC) can be reformulated as a *Mixed-Integer Program with Packing Constraints* (MIPPC):

$$
\begin{aligned}
\text{(MIPPC)} \qquad \min_{x\in\mathbb{R}^n} \quad & c^\top x \\
\text{s.t.} \quad & Ax = b, \\
& 0 \leq x_i \leq u_i\, y_i \qquad \forall\, i \in V, \\
& y_i + y_j \leq 1 \qquad \forall\, \{i, j\} \in E, \\
& y \in \{0, 1\}^n.
\end{aligned}
$$

Note that if the bounds are infinite, complementarity constraints are not MIP-representable in sense of Jeroslow [38].

All results of this article can be directly adapted to (MIPPC). Using this reformulation has several advantages and disadvantages: On the positive side, standard MIP-solving techniques can be used to solve (MIPPC). Moreover, the LP-relaxation of (MIPPC) contains a representation of the complementarity constraints via the packing constraints $y_i + y_j \leq 1$; this is different for the LP-relaxation of (LPCC), which is obtained by neglecting the complementarity constraints. On the other hand, (MIPPC) contains up to twice the number of variables. Moreover, there exist feasible points

(x, y) of the LP-relaxation of (MIPPC) such that x is feasible for (LPCC), but y is not integral. Furthermore, if the bounds u are large, the resulting LP-relaxation turns out to be weak (the typical "big-M" behavior). Thus, this might lead to an increased number of branch-and-bound nodes. We will return to the comparison of (MIPPC) and (LPCC) in the computational results in Sect. 6.2.

In order to strengthen the weak LP-relaxation of (LPCC), consider an SOS1 constraint on variables $S \subseteq V$, denoted by SOS1(S). Thus, we have $x_i \cdot x_j = 0$ for every $i, j \in S, i \neq j$. If $u_j < \infty$ for all $j \in S$, we can add a *bound inequality* of the form

$$\sum_{j \in S} \frac{x_j}{u_j} \leq 1.$$

This inequality arises from the projection of the *clique inequality*

$$\sum_{j \in S} y_j \leq 1$$

for (MIPPC) to the x-variables.

The benefit of clique inequalities in comparison to bound inequalities is that they have $0/1$ coefficients. However, as a result of the before mentioned aspects, handling (LPCC) can compensate this circumstance in many cases. Moreover, unlike clique inequalities, a bound inequality may be further strengthened if every variable x_j, for $j \in S$, is restricted by a variable bound constraint $x_j \leq u_j z_i$, where z_i is some variable of (LPCC) with $0 \leq z_i \leq 1$. The resulting *strengthened bound inequality* is

$$\sum_{j \in S} \frac{x_j}{u_j} \leq z_i. \tag{1.1}$$

Such variable bound constraints often occur if a certain event is time-limited. One example—routing in multi-hop wireless networks—will be explained in detail in Sect. 6.1.1, compare Inequality (6.3).

We will use the following notation: for $i \in V$, define $\Gamma(i)$ as the neighbors of i in G. Note that $i \notin \Gamma(i)$. Furthermore, let $\Gamma(M) := \bigcup_{i \in M} \Gamma(i)$ for a subset $M \subseteq V$. The *support* of a vector $x \in \mathbb{R}^n$ is $\text{supp}(x) = \{i \in V : x_i \neq 0\}$. By x_M we denote the subvector of x restricted to entries in a set $M \subseteq V$.

2 Branching approaches

In this section, we discuss branching rules in a branch-and-bound approach for solving (LPCC). That is, we define ways how to subdivide a given subproblem into two further subproblems Π_1 and Π_2. We say a branching rule is *correct* if the feasible solution set of the corresponding subproblem is covered by the feasible solution sets of Π_1 and Π_2. Ideally Π_1 and Π_2 should be disjoint or, at least, tend to disjointness.

All presented branching rules are based on selecting two sets $C_1, C_2 \subseteq V$ and to add domain fixings $x_j = 0$ for every $j \in C_1$ on the *left* branch and $x_j = 0$ for every

$j \in C_2$ on the *right* one. Using the sets C_1 and C_2 defines a branching rule that splits some of the original SOS1 constraints into two smaller ones. Note that this branching rule is correct if $\{i, j\} \in E$ for every $(i, j) \in C_1 \times C_2$. Moreover, the feasible solution sets of the left and the right branching node are not necessarily disjoint, since 0 is feasible for both if it is feasible for $Ax = b$. In fact, there might exist points with $x_{C_1} = 0$ and $x_{C_2} = 0$, which are thus feasible for both nodes.

In the following, we present three branching rules based on this idea and two further branching variations dealing especially with the disjointness aspect. Throughout this section, we let x^* be the current LP solution and we consider G as the local conflict graph which is induced by the variables not fixed to 0. Furthermore, we require that propagation has already been performed, i.e., if it is known that some variable x_i has to be nonzero for the current branching node (e.g., by other propagation methods like bound strengthening), then we assume that all the variables x_j, $j \in \Gamma(i)$, are locally fixed to zero (and removed from G).

2.1 SOS1 branching

We start with reviewing the SOS1 branching approach of Beale and Tomlin [10], which is predominantly used for solving optimization problems with SOS1 constraints.

Let SOS1($\{1, \ldots, s\}$) be an SOS1 constraint that is violated by the current LP solution x^*. For SOS1 branching, a problem specific ordering of the variables is used. This ordering arises from predefined weights $w \in \mathbb{R}^s$. W.l.o.g. we assume that the variables $x_1, \ldots, x_s$ are sorted such that $w_1 < w_2 < \cdots < w_s$. For the specific problems considered in [10,51], the weights represent a natural ordering. However, if meaningful weights cannot be specified beforehand, one usually takes $w_j = j$, $j = 1, \ldots, s$.

The task is now to split the set $\{1, \ldots, s\}$ into two disjoint parts $C_1 = \{1, \ldots, r\}$ and $C_2 = \{r + 1, \ldots, s\}$ for some index r with $1 \le r \le s - 1$. To achieve a suitable balance, this is done by choosing r such that $w_r \le \bar{w} < w_{r+1}$ for the average weight

$$\bar{w} := \frac{\sum_{j=1}^{s} w_j \cdot x_j^*}{\sum_{j=1}^{s} x_j^*}.$$

Note that the denominator is nonzero by assumption.

2.2 Neighborhood branching

A further branching rule arises from the neighborhood of some variable $i \in V$ with $x_i^* > 0$ such that at least one complementarity constraint $\{i, j\} \in E$ is violated. In this case, we simply use $C_1 = \{i\}$ and $C_2 = \Gamma(i)$. Note that $x_{C_2}^* \neq 0$.

This branching rule may result in an unbalanced branching tree, since on the left branch we fix only one variable to zero and may fix multiple variables in the right branch. Nevertheless, our computational results in Sect. 6.2 show that this rule is effective for some problem instances.

Observe that neighborhood branching corresponds to standard 0/1-branching on the binary variables of (MIPPC) projected on the x-space: using the big-M and packing constraints, branching on variable y_i, $i \in V$, results in $x_i = 0$ in one branch and $x_j = 0$, $j \in \Gamma(i)$, in the other; i.e., the same outcome as with neighborhood branching.

2.3 Bipartite branching

The branching rule we describe in the following is especially designed to obtain a more balanced search tree for LPCCs with overlapping SOS1 constraints. We branch using complete bipartite subgraphs of G associated to some node partition $C_1 \dot\cup C_2$, i.e., we have $\{i, j\} \in E$ for every $(i, j) \in C_1 \times C_2$. It follows that a valid decomposition of the solution set is derived from the disjunction $x_{C_1} = 0 \vee x_{C_2} = 0$ and the branching rule established by C_1 and C_2 is correct. Note that SOS1 and neighborhood branching represent special forms of bipartite branching, since the sets C_1 and C_2 define the parts of a complete bipartite subgraph of G in these cases.

The larger C_1 and C_2 are, the larger is the effect of the branching disjunction, and the tree tends to be more balanced if C_1 and C_2 are of about equal size. Note that finding a maximum balanced complete bipartite subgraph is NP-hard (see Garey and Johnson [25]). In order to determine the sets C_1 and C_2 in practice, we proceed in the following way: given some node $i \in V$, we select a nonempty subset $M \subseteq \Gamma(i)$ and define the sets

$$C_1 := \bigcap_{j \in M} \Gamma(j), \qquad C_2 := \bigcap_{j \in C_1} \Gamma(j).$$

An example is shown in Fig. 1. Here, the choice $i = 1$ and $M = \{2, 3\}$ leads to $C_1 = \{1, 5, 6\}$ and $C_2 = \{2, 3, 4\}$. Note that the presence of the gray edge $\{3, 4\}$ has no relevance, since we are only interested in complete bipartite subgraphs, which are not necessarily induced.

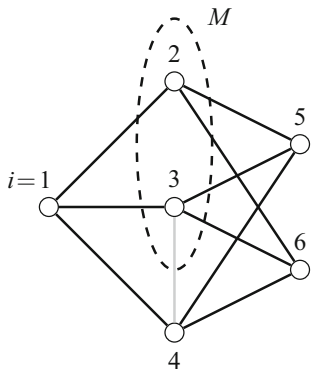

Fig. 1 Illustration of bipartite branching

The sets C_1 and C_2 have the following properties:

Lemma 1 *For $i \in V$, let $M \subseteq \Gamma(i)$, $M \neq \emptyset$. Then C_1 and C_2 satisfy*

(i) *$C_1 \neq \emptyset$ and $C_2 \neq \emptyset$,*
(ii) *$C_1 = \bigcap_{j \in C_2} \Gamma(j)$, and*
(iii) *the bipartite graph B with node partition $C_1 \dot\cup C_2$ is a maximal complete bipartite subgraph of G.*

Proof Statement (i) follows from $i \in C_1$ and $\emptyset \neq M \subseteq C_2$. For the proof of (ii), we observe that $C_1 \subseteq \Gamma(j)$ for every $j \in C_2$ by definition of C_2. This implies $C_1 \subseteq \bigcap_{j \in C_2} \Gamma(j)$. Because of $M \subseteq C_2$, we get the other inclusion $C_1 \supseteq \bigcap_{j \in C_2} \Gamma(j)$.

Finally, we show (iii): as seen above $C_1 \subseteq \Gamma(j)$ for every $j \in C_2$. Analogously one can see that $C_2 \subseteq \Gamma(j)$ for every $j \in C_1$. Since, by assumption, there are no self-loops in G, we also know that $C_1 \cap C_2 = \emptyset$. Therefore, the graph B with node partition $C_1 \dot\cup C_2$ is a complete bipartite subgraph in G. By the presentation of C_1 in (ii) and the definition of C_2, it is easy to see that this subgraph is maximal. □

We observe that the larger C_1 is, the smaller C_2 becomes, and vice versa. This shows that the set M should be carefully selected. In practice, it will be hard to establish which choice of M results in a good balance between C_1 and C_2. Nevertheless, in preliminary tests we have made good experience using

$$M = \{j \in \Gamma(i) \ : \ x_j^* \neq 0\}$$

for some given $i \in V$ if the resulting sets C_1 and C_2 satisfy

$$|\{j \in C_1 \ : \ x_j^* \neq 0\}| > 1, \qquad |\{j \in C_2 \ : \ x_j^* \neq 0\}| > 1.$$

If the above condition is not fulfilled, we use $M = \Gamma(i)$. Note that this often yields neighborhood branching ($C_1 = \{i\}$ and $C_2 = \Gamma(i)$). However, $C_1 \supsetneq \{i\}$ is possible.

2.4 Further branching variations

As mentioned before, the branching rules above have the undesired property that their two branching nodes are not necessarily disjoint, since both of them contain feasible points with $x_{C_1} = 0 \wedge x_{C_2} = 0$. In the following, we present further branching variations dealing with this issue.

2.4.1 Nonzero fixing

If neighborhood branching is used, i.e., $C_1 = \{i\}$ and $C_2 = \Gamma(i)$ for some $i \in V$, we can additionally fix the lower bound of x_i to "nonzero" for the right branching node. In practice, this can be done by adding $x_i \geq 1.5 \cdot \varepsilon$, where ε is the feasibility tolerance (e.g., $\varepsilon = 10^{-6}$ in SCIP). This excludes points x with $x_{C_1} = 0 \wedge x_{C_2} = 0$ from the right node. Of course, nonzero fixing has only a small effect on the solution and objective function. However, it sometimes may lead to an infeasible solution set such that nodes can be pruned from the branch-and-bound tree.

2.4.2 Adding complementarity constraints

In addition to variable domain fixings, it is sometimes also possible to add new complementarity constraints to the branching nodes. This results in a nonstatic conflict graph, which may change dynamically with every branching node.

Applying one of the above mentioned branching rules to the subgraph induced by the variables not fixed to 0, we get nonempty sets C_1 and C_2, leading to variable domain fixings $x_{C_1} = 0$ for the left and $x_{C_2} = 0$ for the right branching node, respectively. The following lemma shows how the conflict graph of the left branching node can be modified locally.

Lemma 2 *Let $B_1, B_2 \subseteq V$ be sets with*

(i) $(B_1 \cup B_2) \cap C_1 = \emptyset$,
(ii) $(B_1 \cup B_2) \cap C_2 = \emptyset$,
(iii) $C_2 \subseteq \Gamma(i) \cup \Gamma(j)$ *for every* $(i, j) \in B_1 \times B_2$.

Then for every $(i, j) \in B_1 \times B_2$ (with $\{i, j\} \notin E$), we can add the complementarity constraint $x_i \cdot x_j = 0$ to the left branching node, i.e., the node in which $x_{C_1} = 0$.

Proof The statement is obviously correct if $B_1 = \emptyset$ or $B_2 = \emptyset$. Thus, we assume that $B_1 \neq \emptyset$ and $B_2 \neq \emptyset$. Let $(i, j) \in B_1 \times B_2$, and let $\bar{x}$ be a feasible point for the left branch with $\bar{x}_i \neq 0$ and $\bar{x}_j \neq 0$; this is possible by (i). Due to (ii) and (iii), we get $\bar{x}_{C_2} = 0$. Therefore, $\bar{x}$ is feasible for the right branching node as well. Thus, the constraint $x_i \cdot x_j = 0$ can be added to the left branching node. □

Remark 1 An analogous statement to Lemma 2 holds for the right branching node. However, note that adding constraints to both nodes needs special care, see [23].

Remark 2 A natural question is whether Lemma 2 gives a complete characterization of all the complementarity constraints that may be added to the branching nodes if the sets C_1 and C_2 form the partition of a maximal complete bipartite subgraph of the current conflict graph. This is studied further in [23].

The sets B_1, B_2 can be computed as indicated in Algorithm 1.

Theorem 1 *Algorithm 1 computes sets $B_1, B_2 \subseteq V$ satisfying (i)–(iii) of Lemma 2.*

Proof If $R = \Gamma(C_2) \setminus (C_1 \cup C_2)$ in Step 1 is not empty, then ℓ in Step 5 exists and has at least one neighbor in C_2. Thus, the set $N_1 = C_2 \cap \Gamma(\ell)$ in Step 6 is nonempty. If $N_2 = C_2 \setminus N_1 = \emptyset$, then $B_2 = \emptyset$ would be the output of the algorithm. In this case, we know that $C_1 \dot\cup C_2$ does not define a maximal partition of a complete bipartite subgraph, since an even larger one exits: $(C_1 \cup \{\ell\}) \dot\cup C_2$. We note that the case $B_2 = \emptyset$ is possible, even if $N_2 = \emptyset$. In this case, no local conflicts are added.

Otherwise, if $B_2 \neq \emptyset$, then the reverse conclusion from above shows that $M_2 \supseteq N_2 \neq \emptyset$ in Step 9. By Lemma 1, the sets B_2 and M_2 define the partition of a maximal complete bipartite subgraph of the induced subgraph $G[C_2 \dot\cup R]$. Similarly, if $M_1 = C_2 \setminus M_2 \neq \emptyset$ in Step 10, we see that B_1 in Step 11 is nonempty due to $\ell \in B_1$ and that $M_1 \dot\cup B_1$ defines the partition of a complete bipartite subgraph (only maximal w.r.t. B_1) of the induced subgraph $G[C_2 \dot\cup R]$. The assertion now follows by construction of B_1 and B_2, since $C_2 = M_1 \cup M_2$. □

Algorithm 1: Computing sets B_1, B_2 for adding complementarity constraints

Input: sets $C_1, C_2 \subseteq V$ with $\{i, j\} \in E$ for every $(i, j) \in C_1 \times C_2$
Output: sets $B_1, B_2 \subseteq V$ satisfying (i)–(iii) of Lemma 2

1 $R \leftarrow \Gamma(C_2) \setminus (C_1 \cup C_2)$;
2 **if** $R = \emptyset$ **then**
3 $\quad B_1 \leftarrow \emptyset, B_2 \leftarrow \emptyset$;
4 **else**
5 $\quad$ select $\ell \in R$;
6 $\quad N_1 \leftarrow C_2 \cap \Gamma(\ell)$;
7 $\quad N_2 \leftarrow C_2 \setminus N_1$;
8 $\quad B_2 \leftarrow \bigcap_{j \in N_2} \Gamma(j) \setminus (C_1 \cup C_2)$;
9 $\quad M_2 \leftarrow C_2 \cap \bigcap_{j \in B_2} \Gamma(j)$;
10 $\quad M_1 \leftarrow C_2 \setminus M_2$;
11 $\quad B_1 \leftarrow \bigcap_{j \in M_1} \Gamma(j) \setminus (C_1 \cup C_2)$;

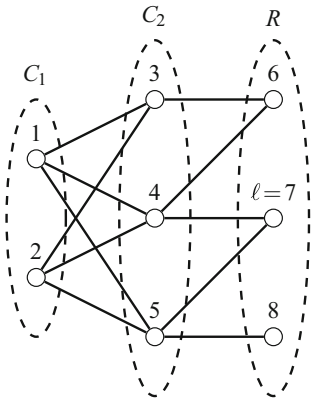

Fig. 2 Illustration of Algorithm 1

Remark 3 Clearly, Algorithm 1 may be executed several times with different choices of the node ℓ in Step 5. If one executes it for every $\ell \in R$, one may set $B_1 = \{\ell\}$ and stop already after Step 8.

For an illustration, we consider the graph in Fig. 2. Let the input of Algorithm 1 be $C_1 = \{1, 2\}$ and $C_2 = \{3, 4, 5\}$. We obtain $R = \{6, 7, 8\}$ in Step 1. Suppose we select $\ell = 7$ in Step 5, which has the two neighbors 4 and 5 in C_2. This yields the set $N_1 = \{4, 5\}$ in Step 6. Then $N_2 = \{3\}$. After the computation of $B_2 = \{6\}$ in Step 8, the sets M_1 and M_2 are $\{5\}$ and $\{3, 4\}$, respectively. Finally, the algorithm returns $B_1 = \{7, 8\}$ and $B_2 = \{6\}$. Subsequently, the complementarity constraints $x_6 \cdot x_7 = 0$ and $x_6 \cdot x_8 = 0$ may be added to the left branching node.

We use Algorithm 1 as a subroutine of our branching procedure. The computed sets B_1 and B_2 allow to add the complementarity constraint $x_i \cdot x_j = 0$ for each (i, j) in $B_1 \times B_2$ to the left branching node. Clearly, a bound inequality derived from a clique S with $\{i, j\} \subseteq S$ can be added as well. Observe that $B_1 \cap B_2 \neq \emptyset$ is possible if C_1 is not maximal. In this case, it follows that $x_i = 0$ for all $i \in B_1 \cap B_2$.

Switching from one branching node to another may be very time consuming when a large number of local constraints is present. Therefore, we only add local constraints with sufficiently large violation of the current LP solution: $x_i \cdot x_j = 0$ is added if $x_i^*/u_i + x_j^*/u_j > 0.4$ and a bound inequality if it is violated by more than 0.5. If no complementarity constraints were added to the left node, one may check whether it is possible to add complementarity constraints to the right one by repeating Algorithm 1 in an analogue manner. We do not add complementarity constraints to both nodes, since this only would be possible with limitations, as mentioned in Remark 1. Of course, the above operations should always be performed on the locally valid conflict graph.

2.5 Selection rules

Branching is performed if the LP solution violates at least one complementarity constraint. Since after each branching step at least one variable is additionally fixed to 0, the branch-and-bound tree has a maximal depth of n and therefore at most 2^n nodes. This proves finite termination of the algorithm.

A selection rule defines on which system of variables or constraints to branch next. The general goal is to find a fast strategy that tends to result in a small number of total branching nodes. In the following, we present two different selection rules which we used in our implementation.

2.5.1 Most infeasible branching

Depending on the used branching rule, the *most infeasible* rule simply chooses the sets C_1 and C_2 with largest value

$$\rho(C_1, C_2) = \sum_{i \in C_1} \sum_{j \in C_2} x_i^* \cdot x_j^*. \tag{2.1}$$

This value tries to capture the amount by which the current LP-relaxation solution would be changed by branching on C_1 and C_2; the hope is that this leads to a significant change of the current dual bound.

2.5.2 Strong branching

We also generalize the idea of *strong branching* (see, e.g., Achterberg et al. [3]). Here, the LP-relaxation is partially solved for each branching decision among a candidate list. Then the decision with best progress is chosen.

For our purposes, we decided to use the following strong branching settings. Let d be the depth of the current branching node. The maximal size of the candidate list is

$$\kappa = \begin{cases} \max\left(10, \lfloor \log_{10}(n) \rfloor\right), & \text{if } d \leq 10, \\ \max\left(5, \left\lfloor \left(\log_{10}(n)\right)^{0.7} \right\rfloor\right), & \text{if } 10 < d \leq 20, \\ 1, & \text{else.} \end{cases}$$

Clearly, if $\kappa = 1$, no LP needs to be solved.

The members of the candidate list are determined with the help of the measure in (2.1) preferring largest values. For each candidate, we tentatively branch by creating the two subproblems and perform a limited number of dual simplex iterations on each; in our implementation, we use a limit of 10,000 iterations. In this way, we can estimate the change Δ_1 and Δ_2 in the objective function w.r.t. the left and the right branching node, respectively. We also tested an adaptive iteration rule like implemented in SCIP. However, this did not improve the CPU time. We follow Achterberg [2] and select a branching decision with largest score value

$$\text{score}(\Delta_1, \Delta_2) = \max(\Delta_1, \varepsilon) \cdot \max(\Delta_2, \varepsilon),$$

where ε is some small positive number (we use the feasibility tolerance in our implementation).

3 Presolving

The purpose of presolving techniques is to simplify the formulation of a given optimization problem with the goal to speed up the solution process. This can lead both to reduced memory requirements and a tightened LP-relaxation. In the context of SOS1 constraints, the handling of cliques is of particular importance.

Let $\mathcal{S}$ be an initial set of cliques in G, such that each edge of G is contained in at least one of these cliques ($\mathcal{S} = E$ is possible). Since $\mathcal{S}$ may include non-maximal or duplicate cliques, we perform a clique extension procedure as follows: in descending order of their size, we check for each clique $S \in \mathcal{S}$ whether it is a subset of any of the already examined ones. If this is the case, the corresponding SOS1 constraint can be deleted from (LPCC). Otherwise, we perform k iterations of the Bron–Kerbosch Algorithm (see [16]), which enumerates with each iteration one different maximal clique that contains S. The resulting SOS1 constraints can then be added to the problem description. Besides the memory aspect, a positive effect of clique extension is that it may improve the efficiency of SOS1 branching, and also a special class of disjunctive cuts that originate from cliques (see Sect. 5.1).

Preliminary tests indicated that $k = 1$ is an adequate choice for the instances we considered. For larger k, the large total number of SOS1 constraints compensates the positive effects of additional cliques. Moreover, the cliques the Bron–Kerbosch Algorithm computes mostly differ in only a few components.

Furthermore, standard presolving techniques are applied, see, e.g., [1,46]. In particular, strengthening variable bounds improves bound inequalities.

4 Primal heuristics

During the branch-and-bound process we make use of several primal heuristics that attempt to derive a good feasible solution from the current LP solution x^*. Primal heuristics can considerably speed up the entire solution process by allowing for pruning of nodes in the enumeration tree. In the following, we present three diving heuristics,

one simple variable fixing heuristic based on independent sets, and one improvement heuristic.

4.1 Diving heuristics

Starting from the current LP solution, diving heuristics iteratively choose a branching decision and resolve the LP (see, e.g., Achterberg [1], Bonami and Gonçalves [14]). In other words, one examines a single root-to-leaf path of the branch-and-bound tree; sometimes limited backtracking is allowed as well.

In the context of (LPCC), we restrict attention to fixing variables to 0, without backtracking. We present three rules to select variables to be fixed, resulting in three different diving heuristics, which we call *violation diving*, *objective shift diving*, and *fractional diving*.

For a given $i \in V$ with $x_i^* \neq 0$, let $\nu(i) = |\mathrm{supp}(x^*_{\Gamma(i)})|$ denote the number of variables that violate a complementarity constraint together with i for the current LP solution. In the following, we require that the upper bounds u are finite. In practice, this always can be ensured by using $\min\{u_i, K\}$ as upper bounds, where $K \in \mathbb{R}$ is some very large value.

Violation diving Select a variable i with $x_i^* \neq 0$ and $\nu(i) \geq 1$ that minimizes

$$\omega_i := \frac{1}{\nu(i)} \cdot \frac{x_i^*}{\sum_{j \in \Gamma(i)} x_j^*}. \tag{4.1}$$

Note that $\sum_{j \in \Gamma(i)} x_j^* \neq 0$ by assumption. The second term of (4.1) describes the violation ratio, i.e., the ratio of the change of $\sum_{j \in V} x_j^*$ due to fixing x_i to zero and fixing all the neighboring variables of x_i to zero. Additionally, we scale by the inverse of $\nu(i)$ in order to prefer variables that violate many complementarity constraints together with their neighbors.

Objective shift diving Let $\varepsilon > 0$. Select a variable i with $x_i^* \neq 0$ and $\nu(i) \geq 1$ that minimizes

$$\omega_i := \frac{1}{\nu(i)} \cdot \frac{\left|c_i\, x_i^*\right| + \varepsilon}{\left|\sum_{j \in \Gamma(i)} c_j\, x_j^*\right| + \varepsilon}. \tag{4.2}$$

Here we again scale by the inverse of $\nu(i)$.

Fractional diving For this selection rule, we use the ratio x_i^*/u_i for each variable i; this amounts to the values of the y-variables of (MIPPC) in an LP solution. We also allow that x_i has a variable upper bound $u_i\, z_i$ for some variable z_i with $z_i \geq 0$. Thus, the resulting weights are either

$$\omega_i = \frac{x_i^*}{u_i} \qquad \text{or} \qquad \omega_i = \frac{x_i^*}{u_i\, z_i^*}. \tag{4.3}$$

We then choose some variable i with $\nu(i) \geq 1$ that has maximum weight (4.3), and fix all neighboring variables x_j, $j \in \Gamma(i)$, to zero. The idea is that a large weight (tending to be one) often indicates that the neighboring variables will be zero in a feasible solution. In contrast, our experience has shown that a small weight (4.3) usually does not provide an appropriate indication for fixing variable x_i itself to zero and not its neighbors. This can be explained by the fact that the upper bounds u_i, $i \in V$, are often quite large.

After the selection of the variable(s), we fix it (them) to zero, resolve the LP, and iterate further until the LP gets infeasible or we obtain a solution that is feasible for the original problem.

4.2 Maximum weighted independent set fixing

The next heuristic is based on independent sets in G, i.e., subsets of the nodes V for which no two nodes are adjacent. For an independent set $I \subseteq V$, consider the following linear program

$$\begin{aligned} \xi(I) \;:=\; \min_{x \in \mathbb{R}^n} \quad & c^\top x \\ \text{s.t.} \quad & Ax = b, \\ & x_i = 0 \qquad\qquad \forall\, i \notin I, \\ & 0 \leq x \leq u. \end{aligned} \tag{4.4}$$

In fact, every feasible solution of (4.4) is a feasible solution of (LPCC). Thus, using the LP solution x^* of (LPCC), the goal is to find some independent set I for which $\xi(I)$ gives an improvement of the current primal bound. To find I, we solve the *maximum weighted independent set* (MWIS) problem

$$\max_{y \in \{0,1\}^V} \{\omega^\top y \;:\; y_i + y_j \leq 1,\ \{i, j\} \in E\}. \tag{4.5}$$

The weights $\omega_i \in \mathbb{R}$, $i \in V$, can be chosen by all three options (4.1)–(4.3). We use the weights in (4.3), since this produced the best results.

The computation of the optimal solution of (4.5) is NP-hard in general. It is thus advisable to use a heuristic. In our implementation we sequentially apply two different heuristics: First, we use the greedy heuristic GGWMIN [45] and then the branch-and-bound algorithm *tclique*, which is available as a subroutine in SCIP (see also Borndörfer and Kormos [15]); limiting the number of branching nodes turns *tclique* into a heuristic. Then (4.4) is solved for both of the produced independent sets and the best result is taken. This possibly leads to a good feasible solution of (LPCC).

4.3 Local search

In order to improve feasible solutions, we use the following local search procedure. Let I_0 be some initial independent set we get, for instance, by solving (4.5) or a

diving-based heuristic. In case I_0 is not maximal, we greedily extend it to a maximal independent set. Starting from I_0, we then move iteratively from one maximal independent set I_k to another I_{k+1}. For the selection of I_{k+1}, we scan a set of promising candidates $\mathcal{C}(I_k)$. If possible, we choose one that leads to the best improvement w.r.t. the problem $\xi(I_{k+1})$ defined in (4.4). This process is continued until no improvement occurs anymore. Note that in each iteration (4.4) has to be solved several times.

It remains to specify an adequate candidate set $\mathcal{C}(I_k)$. Let ω be the weights defined in (4.1). For an independent set $I \subset V$ and $i \in V \setminus I$, we consider $I^* := C(I, i)$ to be a possible candidate containing i if it is computed according to the following rule:

1. Initialize $I^* \leftarrow (I \setminus \Gamma(i)) \cup \{i\}$.
2. Compute $L := \{\ell \in V \ : \ I^* \cap (\{\ell\} \cup \Gamma(\ell)) = \emptyset\}$, the set of possible extensions for I^*.
3. If $L = \emptyset$, then I^* is a maximal independent set and we stop. Otherwise, select the smallest (for uniqueness) index $\ell \in L$ with $\omega_\ell = \max_{j \in L} \omega_j$. Set $I^* \leftarrow I^* \cup \{\ell\}$ and go to Step 2.

As candidate set, we choose

$$\mathcal{C}(I_k) := \{C(I_k, i) \ : \ i \in V \setminus I_k, \ \omega_i \geq \alpha\}$$

for some $\alpha \geq 0$. In our implementation, α is chosen such that the size of $\mathcal{C}(I_k)$ is restricted to 40.

5 Cutting planes

To tighten the LP-relaxation of (LPCC), we adapt several classes of cutting planes known from the literature to (LPCC) with overlapping SOS1 constraints. All of them might as well be used for (MIPPC), for which some can even be equivalently stated with 0/1-coefficients. This is one advantage of (MIPPC) in comparison to (LPCC), for which the coefficients often are real valued. However, as already mentioned, we will substantiate with our computational results in Sect. 6.2 that using (LPCC) can be beneficial in many cases.

5.1 Disjunctive cuts

We first recall different variants of *disjunctive cuts*, taking special structures of the problem description into account. In their simplest form, i.e., for edge disjunctions $x_i \leq 0 \vee x_j \leq 0$ with $\{i, j\} \in E$, they already appeared in the early 1970s in Owen [43]. Later, de Farias et al. [19] mentioned in passing the idea to generate disjunctive cuts for more general disjunctions derived from cliques in G. As far as we know, disjunctive cuts until now have been tested only for edge disjunctions of conflict graphs consisting of a matching (see, e.g., Audet et al. [7] and Júdice et al. [39]).

Let x^* a feasible basic solution of the LP-relaxation of (LPCC)

$$\min \left\{ c^\top x \ : \ Ax = b, \ 0 \leq x \leq u \right\}.$$

We denote by B and N the indices of its basic and nonbasic variables, respectively. To simplify the presentation, we assume that all nonbasic variables are at their lower bound 0; the general case can be handled by variable complementation, i.e., replacing x_i by $u_i - x_i$ (see Júdice et al. [39] for further information). In terms of the simplex tableau, each feasible point x satisfies

$$x_{B(k)} = \overline{a}_{k0} - \sum_{\nu \in N} \overline{a}_{k\nu} x_\nu \quad \forall k \in \{1, \ldots, m\}, \tag{5.1}$$

where $B(k)$ is the kth entry of B, the coefficients $\overline{a}_{k\nu}$ are entries of the matrix $A_B^{-1} A$, and $\overline{a}_{k0} = (A_B^{-1} b)_k$.

5.1.1 Disjunctive cuts on two-term disjunctions

If x^* satisfies all complementarity constraints, then it is feasible for (LPCC). Otherwise, there exist $i, k \in \{1, \ldots, m\}$, $i \neq k$, with $\{B(i), B(k)\} \in E$ and $x^*_{B(i)}, x^*_{B(k)}$ both positive. We consider the two-term disjunction

$$x_{B(i)} = \overline{a}_{i0} - \sum_{\nu \in N} \overline{a}_{i\nu} x_\nu \leq 0 \quad \vee \quad x_{B(k)} = \overline{a}_{k0} - \sum_{\nu \in N} \overline{a}_{k\nu} x_\nu \leq 0,$$

which is a reformulation of $x_{B(i)} \cdot x_{B(k)} = 0$ and thus satisfied by each feasible point. Since $\overline{a}_{r0} = x^*_{B(r)} > 0$ for $r = i, k$, one can observe that the disjunctive cut

$$\sum_{\nu \in N} \max \left\{ \frac{\overline{a}_{i\nu}}{\overline{a}_{i0}}, \frac{\overline{a}_{k\nu}}{\overline{a}_{k0}} \right\} x_\nu \geq 1 \tag{5.2}$$

is valid for (LPCC). Moreover, (5.2) is violated by x^* because $x^*_N = 0$.

This method can easily be generalized to two-term disjunctions of the form

$$\sum_{i \in C_1} x_i \leq 0 \quad \vee \quad \sum_{i \in C_2} x_i \leq 0, \tag{5.3}$$

for disjoint $C_1, C_2 \subseteq V$ such that $\{i, j\} \in E$ for all $(i, j) \in C_1 \times C_2$ and $\sum_{i \in C_1} x^*_i > 0$ and $\sum_{i \in C_2} x^*_i > 0$. We choose C_1 and C_2 by one of the following criteria:

1. *Edge* For an edge $\{i, j\} \in E$, choose $C_1 = \{i\}$ and $C_2 = \{j\}$.
2. *Neighborhood* For a node $i \in V$, choose $C_1 = \{i\}$ and $C_2 \subseteq \Gamma(i)$.
3. *Clique* For a clique S in G, choose disjoint subsets $C_1, C_2 \subseteq S$.
4. *Bipartite* Choose C_1, C_2 as the partition of a complete bipartite subgraph of G.

We can compute C_1 and C_2 by analogous methods as for branching rules in Sect. 2. Here, it is sensible to concentrate on the positive components of $x^*_{C_1}$ and $x^*_{C_2}$.

5.1.2 Disjunctive cuts on box disjunctions

Assume that x_i and x_j with $i, j \in V$ and $\{i, j\} \notin E$ are restricted by upper bounds u_i and u_j, respectively. Further assume that we know that there exists an optimal solution $\bar{x}$ of (LPCC) which satisfies

$$x_i \leq 0 \quad \vee \quad x_i \geq u_i \quad \vee \quad x_j \leq 0 \quad \vee \quad x_j \geq u_j. \tag{5.4}$$

Similarly, if there exist variable upper bounds $x_i \leq u_i z_i$ and $x_j \leq u_j z_j$ for nonnegative variables z_i and z_j, we assume that the optimal solution $(\bar{x}, \bar{z})$ satisfies

$$x_i \leq 0 \quad \vee \quad x_i \geq u_i z_i \quad \vee \quad x_j \leq 0 \quad \vee \quad x_j \geq u_j z_j. \tag{5.5}$$

We call (5.4) or (5.5) four-term disjunction in box-shaped form. These arise, for example, in flow problems with parallel arcs, see Sect. 6.1 for applications. In this case, one may assume that at most one flow value on parallel arcs does not attain the bounds. Moreover, in the complementarity constrained continuous knapsack problem this can be seen by the well-known fact that there always must exist an optimal solution $\bar{x}$ with at most one component $\bar{x}_i$ not being equal to its bounds.

Remark 4 Note that any optimal solution to an LPCC coincides with a solution of an LP in which appropriate variables are fixed to 0, compare (4.4). Thus, the property of the LP-relaxation having at most one variable value that is not at its bound carries over to the corresponding LPCC.

If x^* violates (5.4) or (5.5), one can then generate a disjunctive cut by inserting the four corresponding rows (5.1) of the simplex tableau and proceeding analogously as in Sect. 5.1.1.

Example 1 To illustrate the approach, we consider the following example:

$$\begin{aligned} \max \quad & \frac{6}{5} x_1 + x_2 + x_3 \\ \text{s.t.} \quad & x_1 + x_2 + x_3 \leq \frac{3}{2}, \\ & 0 \leq x_1, x_2, x_3 \leq 1, \\ & x_1 \cdot x_2 = 0, \quad x_1 \cdot x_3 = 0. \end{aligned}$$

We add the bound inequalities $x_1 + x_2 \leq 1$ and $x_1 + x_3 \leq 1$. Then the solution of the LP-relaxation is $x^* = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)^\top$ with objective value 1.6, which violates the disjunction $x_2 \leq 0 \vee x_2 \geq 1 \vee x_3 \leq 0 \vee x_3 \geq 1$; note that $\{2, 3\} \notin E$. If we proceed as described above, we obtain (after some calculations) the disjunctive cut $6x_1 + 4x_2 + 4x_3 \leq 6$, which is added to the LP-relaxation. Resolving yields the (feasible) optimal solution $\bar{x} = \left(0, \frac{1}{2}, 1\right)^\top$ with objective value 1.5.

5.2 Bound cuts

The separation of bound inequalities that we defined in Sect. 1.2 is NP-hard, as already well-known for clique inequalities (see Nemhauser and Wolsey [42]). Nevertheless, there exist fast separation heuristics. The one we present in the following is an adaptation of the clique inequality separator implemented in SCIP.

We consider the induced subgraph $G[U]$ of G on $U := \{i \in V \ : \ u_i < \infty\}$. Let $\bar{E} = \{\{i, j\} \ : \ i, j \in U, \ i \neq j, \text{ and } \{i, j\} \notin E\}$ be the edges of its complement graph. The separation procedure is based on the maximum weighted clique problem

$$\max_{y \in \{0,1\}^U} \left\{ \omega^\top y \ : \ y_i + y_j \leq 1, \ \{i, j\} \in \bar{E} \right\}. \tag{5.6}$$

Using the weights ω defined in (4.3), every feasible solution of (5.6) with objective value greater than 1 defines a (strengthened) bound inequality that cuts off x^*. In our cutting plane separator, we again use the *tclique* algorithm of SCIP using a node limit of 10,000 in order to solve (5.6) heuristically.

5.3 Cover cuts

In this section, we introduce a family of *cover inequalities* of the form $\sum_{i \in I} \frac{x_i}{u_i} \leq |I| - 1$, for $I \subseteq V$, which represent a linear relaxation of the nonlinear cardinality constraints $|\text{supp}(x_I)| \leq |I| - 1$.

Let $S \subseteq V$ and $\{S_k \ : \ k \in K\}$ be a clique partition of $G[S]$, i.e., $S = \bigcup_{k \in K} S_k$ and each node of S belongs to exactly one clique S_k, $k \in K$. Note that the trivial case $S_k = \{k\}$ for $K = S$ is possible, but leads to trivial cuts. We then consider the system

$$\begin{aligned} &\sum_{i \in S} x_i \geq \beta, \\ &0 \leq x_i \leq u_i \quad \forall i \in S, \\ &\text{SOS1}(S_k) \quad \forall k \in K, \end{aligned} \tag{5.7}$$

with $\beta > 0$ and $u_i > 0$ for every $i \in S$.

Let $I \subseteq V$ be an independent set. Note that if $I \cap S_k \neq \emptyset$, then $|S_k \setminus \Gamma(I)| = 1$. Using the convention $\max(\emptyset) = 0$, we define

$$\phi_k^I := \max\{u_i \ : \ i \in S_k \setminus \Gamma(I)\},$$

for $k \in K$. Assuming that

$$\sum_{k \in K} \phi_k^I < \beta, \tag{5.8}$$

it is impossible for all variables x_i with $i \in I$ to be nonzero while the feasibility of (5.7) is maintained, since every solution that satisfies the SOS1 constraints would violate

$\sum_{i \in S} x_i \geq \beta$. In this case, we say that I is a *cover*. Every cover induces a cardinality constraint $|\text{supp}(x_I)| \leq |I| - 1$; thus, if $u_i < \infty$, $i \in I$, the cover inequality

$$\sum_{i \in I} \frac{x_i}{u_i} \leq |I| - 1 \tag{5.9}$$

is valid. In order to get strong cover inequalities, we require that I is a *minimal cover*, i.e.,

$$\sum_{k \in K} \phi_k^{I \setminus \{i\}} \geq \beta,$$

for every $i \in I$.

We use the following separation heuristic: first, we compute a clique partition of $G[S]$. For this, we recursively search for some maximal clique and remove its induced graph from $G[S]$ until no node remains. To compute a minimal cover, we initialize $I = \emptyset$. Then, in decreasing order of weights $\omega_i = x_i^*/u_i$, we iteratively add an element $i \in \Gamma(S) \setminus (\Gamma(I) \cup I)$ to I until (5.8) is satisfied or otherwise $\Gamma(S) \setminus (\Gamma(I) \cup I) = \emptyset$. In the first case, we iteratively delete an element from I, in reverse order, if its deletion still leaves a cover. The resulting I is a minimal cover, and inequality (5.9) may be added to the cutting plane pool if it is violated by x^*.

Of course, a lifting can help to obtain strengthened or even facet defining inequalities. We leave this as an open topic for further research.

6 Computational experiments

We implemented all ideas presented in this paper in C using the framework SCIP [2,47] version 3.1.1 with CPLEX 12.6 as LP-solver. We compare its results with the MIP-solver CPLEX 12.6, which we apply to (LPCC) as well as (MIPPC), and the MIP-solver SCIP 3.1.1, which we apply to (MIPPC). We used a time limit of two hours. Our tests were run on a linux cluster with 64 bit Intel dual core i3 CPUs with 3.2 GHz, 4 MB cache, and 8 GB main memory.

6.1 Instance sets

As stated before, optimization problems with complementarity constraints arise in many areas. In the following, we describe three applications for which we will generate instances, used for our computational experiments.

6.1.1 Maximum flow problem with conflicts

Gupta and Kumar [27] present the so-called *protocol model* for modeling interference in a *multi-hop wireless network*. Based on this, Jain [36] and Shi et al. [48] investigated different variants with different assumptions for computing the optimal throughput on such a network. In the following, we present a combination of both.

A multi-hop wireless network consists of a set N of transmission nodes that may receive or transmit data over wireless links L. We assume that multiple frequencies are not available for data transmission. Instead, we suppose that there exists only one frequency, but each time period is split into different time slots T of variable length; an assumption that is made in [36] as well. Given a source node s and a destination node d, the objective is to find a maximum data flow from s to d subject to the condition that interference is avoided. Interference is caused due to the reuse of the same time slot on nearby links.

In the protocol model, interference is modeled in the following way: For a given $u \in N$, let $\mathcal{T}_u$ its *transmission area* and $\mathcal{I}_u$ its *interference area*. For simplicity of presentation, it is assumed that the nodes are placed in the Euclidean plane and $\mathcal{T}_u$, $\mathcal{I}_u$ are determined by disks with center $u \in N$ and radius R_T and R_I, respectively. In general, R_T should be smaller than R_I. We define

$$L = \{(u, v) \in N \times N \ : \ v \in \mathcal{T}_u, \ v \neq u\}$$

as the set of available links and

$$E = \{\{(u, v), (w, q)\} \in L \times L \ : \ q \in \mathcal{I}_u, \ (u, v) \neq (w, q)\}$$

as the set of interfering links. Thus, if there is a transmission from u to v over time slot $t \in T$, then every node $q \in \mathcal{I}_u$ may not receive data from some node w over the same time slot t. This includes that every node may neither transmit to more than one other node nor receive from more than one other node at the same time.

Let x_ℓ^t be the flow on link $\ell \in L$ at time slot $t \in T$. The sets $\delta^-(u)$ and $\delta^+(u)$ denote the incoming and the outgoing links of each node $u \in N$, respectively. The variable z_t with $t \in T$ describes the relative share of time slot t of each time period. We consider the *Maximum Flow Problem with Conflicts* (MFPC) which computes the optimal periodic throughput (in a steady state). The MFPC is defined formally as

$$\begin{aligned}
\text{(MFPC)} \quad \max \quad & r \\
\text{s.t.} \quad & \sum_{t \in T} \left(\sum_{\ell \in \delta^+(u)} x_\ell^t - \sum_{\ell \in \delta^-(u)} x_\ell^t \right) = \beta_u && \forall u \in N, && (6.1)\\
& \sum_{t \in T} z_t \leq 1, && && (6.2)\\
& 0 \leq x_\ell^t \leq C_\ell \, z_t && \forall \ell \in L, \ t \in T, && (6.3)\\
& z_t \geq 0 && \forall t \in T, && (6.4)\\
& x_\ell^t \cdot x_{\bar{\ell}}^t = 0 && \forall \{\ell, \bar{\ell}\} \in E, \ t \in T, && (6.5)
\end{aligned}$$

where β_u is r if u is the source, $-r$ if u is the destination, and 0 otherwise. The first set of constraints (6.1) are the flow conservation constraints, which enforce the balance between the incoming and the outgoing flow over the time period. Constraint (6.2) describes the partitioning of each time period into time slots. In (6.3), the value $C_\ell \in \mathbb{R}$,

$\ell \in L$, denotes the capacity of $\sum_{t \in T} x_\ell^t$, which depends on the distance between the endnodes of ℓ and the maximum transmission power (see the formula in [48]). The subsequent complementarity constraints (6.5) model mutual interference between links. Optionally, one can add symmetry-breaking constraints $z_{t-1} \leq z_t$, $t = 2, \ldots, |T|$, to reduce the negative effect of symmetry with respect to arbitrary permutations of the time slots.

For the MFPC, we generated instances of three different types; type one with 25, type two with 35, and type three with 50 transmission nodes, from which we selected three to be the sources (all connected to a virtual super-source) and one to be the destination, respectively. The nodes were located on a 1000×1000 grid. Their position in the grid is chosen uniformly at random for the instances of type two. Therefore, it might happen that there is no connection from all the sources to the destination. For type one and three, we first positioned the source in the center of the grid. For each source-destination pair we then generated a connecting path of length three, each arc having a length uniformly distributed in the interval [90, 180] and a direction change from the former arc direction of at most 90°. The position of the remaining nodes was chosen at random. Further parameter settings are $R_T = 200$, $R_I = 300$, and $|T| = 8$.

6.1.2 Transportation problem with exclusionary side constraints

A common scenario of logistic problems is the cost-effective transportation of goods from several sources $i \in S$ to destinations $j \in D$. For the *Transportation Problem with Exclusionary (or Nonlinear) Side Constraints* (TPESC)—originally proposed by Cao [17] and also studied by Sun [49], Syarif and Gen [50]—it is not allowed to carry certain goods to the same destination. This type of problems has many applications. For instance, hazardous materials (toxic, corrosive, flammable, etc.) may not be stored in the same warehouse or vehicle.

The transportation network has nodes $S \cup D$ and source-destination pairs $L = S \times D$ as arcs. The flow on $\ell \in L$ is given by x_ℓ. TPESC can be modeled as a classical transportation problem with additional complementarity constraints:

$$
\begin{aligned}
\text{(TPESC)} \qquad \min \quad & \sum_{\ell \in L} c_\ell \, x_\ell \\
\text{s.t.} \quad & \sum_{\ell \in \delta^+(i)} x_\ell = \alpha_i \qquad \forall\, i \in S, \\
& \sum_{\ell \in \delta^-(j)} x_\ell = \beta_j \qquad \forall\, j \in D, \\
& x \geq 0, \\
& x_\ell \cdot x_{\bar{\ell}} = 0 \qquad\qquad \forall\, \{\ell, \bar{\ell}\} \in E,
\end{aligned}
$$

where the supplies and demands are given by $\alpha \in \mathbb{R}_+^S$, $\beta \in \mathbb{R}_+^D$, respectively. We assume that the balancing condition $\sum_{i \in S} \alpha_i = \sum_{j \in D} \beta_j$ is satisfied. Moreover, c_ℓ denotes the transportation costs for arc ℓ. Every source i is assigned to a given category $C_i \in \{1, \ldots, t\}$, where $t \in \mathbb{N}$. We assume transportation of sources from

conflicting categories to the same destination is not allowed. This defines a conflict graph $G = (L, E)$ with node set L and edge set

$$E = \{\{(i, j), (k, j)\} \in L \times L \ : \ C_i \text{ and } C_k \text{ are in conflict}\}.$$

We use $u_{ij} := \min\{\alpha_i, \beta_j\}$ as a big-M value for the MIP-reformulation of (TPESC).

We generated three instance sets of three different types; type one and two with $|S| = 150$ and $|D| = 15$, type three with $|S| = 200$ and $|D| = 15$. The other inputs were randomly generated; c_ℓ is chosen uniformly at random from $[3, 8]$, α_i and β_j on $[100, 500] \cap \mathbb{N}$ and $[1000, 5000] \cap \mathbb{N}$, respectively. In case of a surplus, i.e., $\sum_{i \in S} \alpha_i > \sum_{j \in D} \beta_j$, the α_i values were evenly reduced until the sums are balanced. We analogously treated the β_j values in case of an undersupply. The categories were also assigned randomly to each source. Conflicts between different categories were determined by the entries of a conflict matrix $Z \in \{0, 1\}^{t \times t}$. For type one, we used for all instances the same 9×9 realistic conflict matrix from Hamdi et al. [28]. For each instance of type two and three, we used a different randomly generated conflict matrix of size 15×15 and expected density 0.35 (w.r.t. the nondiagonal entries).

6.1.3 Continuous knapsack problem with conflicts

Our last instance sets arise from the *Continuous Knapsack Problem with Conflicts* (CKPC). Let $V = \{1, \ldots, n\}$ and $G = (V, E)$ be an undirected conflict graph. The mathematical formulation of CKPC is

$$\begin{aligned} \text{(CKPC)} \qquad \max \quad & \sum_{i \in V} c_i x_i \\ \text{s.t.} \quad & \sum_{i \in V} a_i x_i \le b, \\ & 0 \le x \le \mathbb{1}, \\ & x_i \cdot x_j = 0 \qquad \{i, j\} \in E, \end{aligned}$$

where $b \ge 0$ and $c, a \in \mathbb{R}^V_+$. We assume w.l.o.g. that the upper bounds are scaled to be $\mathbb{1}$. The CKPC, specifically with nonoverlapping SOS1 constraints, was first investigated by Ibaraki et al. [34,35]. Later, de Farias et al. [20] studied inequalities that define facets of the convex hull of its feasible set. Applications arise, e.g., in capital budgeting (see Lin [40]). Other articles (see, e.g., Hifi and Michrafy [29], Pferschy and Schauer [44]) deal with the knapsack problem with overlapping SOS1 constraints, but binary variables.

As an application, we consider the following scenario: given is a set of jobs V of different sizes a_i and benefit c_i, as well as a processor with a certain capacity b. The goal is to optimally assign a subset $I \subseteq V$ of jobs to the processor, where some jobs may be incompatible with one another, i.e., they cannot be executed both on the processor; reasons are given in Baker, Coffmann [8] and Jansen [37]. In the context of the CKPC, we assume it is permissible that jobs are only partly executed.

The input data for three different types of our problem generation are $n = 200$ for type one, $n = 250$ for type two, and $n = 260$ for type three. The objective coefficients c_i and row coefficients a_i were generated uniformly at random in the interval [10, 25]. In order to show the effect of disjunctive cuts on box disjunctions (see Sect. 5.1.2), we ensured, especially for the type three instances, that several objective coefficients are of equal value. This sets an incentive for optimal LP solutions with at least two fractional variables, see Example 1. Equal objective values were achieved by choosing c_i among 20 different random values. The right hand side b was determined according to $b = 0.3 \cdot \mathbb{1}^\top a$, which comes from de Farias and Nemhauser [22]. Mutual conflicts between variables were generated uniformly at random with expected density 35% (the density is the percentage number of edges in comparison to the complete graph).

6.2 Experimental results

Table 1 gives statistics on the test sets that we used for our experiments. The data relate to the transformed instances after the presolving step. Column "#" denotes the number of instances of the given problem type. Moreover, in arithmetic mean over all instances of each problem type, we list the number of variables in column "#vars", the number of linear constraints in "#linear", the number of SOS1 constraints in "#SOS1", the arithmetic mean over the size of the SOS1 constraints in "SOS1 size", and the conflict graph density in "dens". All instances are available for download from the web pages of the authors.

The computations are organized into six experiments. The goal of the first four experiments is to compare the different branching rules and cutting planes introduced in Sects. 2 and 5, respectively. For this purpose, we initialized the algorithm with a precomputed optimal solution in order to eliminate the impact of heuristics on the performance. In Experiment 5, we evaluate the use of heuristics. Finally, in Experiment 6, we compare our implementation with other branch-and-cut solvers and with the (MIPPC) formulation.

Table 1 Statistics of the problem instances after presolving (in arithmetic mean)

Problem	Type	#	#Vars	#Linear	#SOS1	SOS1 size	Dens/%
MFPC	1	20	746.6	766.1	1031.5	27.6	8.4
	2	20	977.2	1007.9	1445.7	18.0	4.8
	3	20	2085.8	2131.3	3554.6	39.4	5.4
TPESC	1	20	2250.0	165.0	56496.0	3.4	2.3
	2	40	2250.0	165.0	55604.2	3.1	2.4
	3	30	3000.0	215.0	92598.0	3.0	2.2
CKPC	1	30	200.0	1.0	1849.3	4.9	35.0
	2	30	250.0	1.0	2739.7	5.2	35.0
	3	20	260.0	1.0	2929.7	5.2	35.0

For SOS1, neighborhood, and bipartite branching, we use the following common settings:

Branching Rules Use most infeasible branching as a selection rule.
Cutting Planes Add (strengthened) bound inequalities during the separation phase at the root node; all the other cutting planes are turned off.

We turned bound inequalities on, since they are necessary to solve a significant number of instances within the time and memory limit. In each experiment, we additionally turn different components of our solver on or off in order to decide which components should be considered in our favorite settings.

The data of Tables 3, 4 and 5 report aggregated results of our first four experiments. In each table, we list the number of instances that reached the time or memory limit (column "limit"), the shifted geometric mean of the number of nodes (column "nodes"), and the CPU time (column "time"). The shifted geometric mean of values $t_1, \ldots, t_n$ with shift value δ is defined as $\left(\prod_{i=1}^{n}(t_i + \delta)\right)^{1/n} - \delta$. We used a shift of 100 for the number of nodes and 10 for the CPU time, see [1]. If the memory limit is reached, we credit the full time limit of two hours. Detailed results for each instance can be found in the online supplement.

We start with an evaluation of the three basic branching variants SOS1, neighborhood, and bipartite branching, denoted by "B-SOS1", "B-neigh" and "B-bip", respectively. SOS1 branching turns out to have a poor performance on all instance sets. If the SOS1 constraints overlap, this leads to the fact that neighborhood and bipartite branching usually fix more variables to zero; consider a star graph, for example.

For the TPESC instances, a very good performance of bipartite branching in comparison to neighborhood branching is clearly noticeable. For these instances, the conflict graph consists of families of vertices with an identical neighborhood structure; a characteristic that usually results in large bipartite subgraphs. This statement is confirmed by Table 2, where we present in column "bip. size" the average size of the smaller part of each complete bipartite subgraph that the algorithm selects for branching.

In contrast, the "unbalanced" neighborhood branching performed better for the the MFPC instances. Here, few large "balanced" bipartite subgraphs exist in the problem specific conflict graph structure. Recall that a graph might have large neighborhoods,

Table 2 Average size of the smaller part of each complete bipartite subgraph

Problem	Type	bip. size
MFPC	1	2.1
	2	1.9
	3	2.6
TPESC	1	28.5
	2	13.8
	3	17.7
CKPC	1	1.0
	2	1.0
	3	1.0

Table 3 Experiments on the MFPC instances (nodes and time in shifted geometric mean)

Setting	MFPC type 1 (20)			MFPC type 2 (20)			MFPC type 3 (20)		
	Limit	Nodes	Time	Limit	Nodes	Time	Limit	Nodes	Time
B-SOS1	11	593535	2426.0	17	861814	3680.5	18	452631	6364.2
B-neigh	0	19106	79.7	3	131716	619.7	3	86885	1204.5
B-bip	0	21991	95.7	3	146992	714.5	4	98423	1499.3
B-nonzero	0	19199	84.3	3	132680	645.3	3	86950	1267.6
B-neigh-comp	0	19099	79.9	3	131808	620.5	3	86819	1206.1
B-bip-comp	0	21970	96.1	3	145745	710.1	4	98594	1507.6
B-neigh-strong	0	8651	91.0	2	61713	397.1	3	55451	1153.8
B-bip-strong	0	8702	104.2	2	68055	460.7	3	72541	1686.7
B-neigh-noC-bds	20	5803068	7200.0	20	6154051	7200.0	20	1773165	7200.0
B-neigh-noC-strBds	20	5551050	7200.0	20	5919731	7200.0	20	961099	7200.0
B-neigh-C-bds10	0	18470	79.3	3	120043	596.8	2	68325	1014.9
B-neigh-C-edge	0	17270	79.5	0	54093	344.2	0	16880	362.1
B-neigh-C-neigh	0	18440	81.7	2	99247	601.5	1	19541	400.1
B-neigh-C-clique	0	19143	88.2	0	75565	470.7	2	23342	479.7
B-neigh-C-bip	0	17661	78.1	2	90875	538.9	1	19416	393.0
B-neigh-C-box	0	18078	78.1	1	76323	413.6	1	32354	538.1
B-neigh-C-cover	–	–	–	–	–	–	–	–	–
B-neigh-favor	0	17068	80.3	0	52167	336.6	0	15862	348.5

but only small balanced bipartite subgraphs. This is true even for the CKPC instances, whose conflict graph is generated randomly without any clear structure. Here, the algorithm automatically nearly always prefers to branch on a neighborhood, see the remark at the end of Sect. 2.3.

Experiment 2: Further Branching Variations In the second experiment, we investigate the use of nonzero fixing ("B-nonzero") and adding complementarity constraints during the branching process in combination with either bipartite or neighborhood branching ("B-bip-comp" and "B-neigh-comp").

Recall that "B-nonzero" is a variant of neighborhood branching with additional fixings. In fact, nonzero fixing does not have a positive effect w.r.t. the standard neighborhood branching rule. It seems to rarely allow for pruning nodes in the branch-and-bound tree and sometimes increases the number of nodes.

Adding complementarity constraints during the branching process significantly speeds up neighborhood branching for the TPESC instances. However, neighborhood branching with the addition of complementarity constraints is still slower than bipartite branching on these instances. For bipartite branching, adding complementarity constraints reduces the number of nodes, but not the CPU time (a Wilcoxon signed rank test, see Berthold [13], confirmed a statistically significant reduction in the number of nodes with a p value of less than 0.0005, but the time reduction is not significant with a p value larger than 0.1). The reason is that adding comple-

Table 4 Experiments on the TPESC instances (nodes and time in shifted geometric mean)

Setting	TPESC type 1 (20)			TPESC type 2 (40)			TPESC type 3 (30)		
	Limit	Nodes	Time	Limit	Nodes	Time	Limit	Nodes	Time
B-SOS1	20	508073	7200.0	40	457255	7200.0	30	255846	7200.0
B-neigh	0	7103	49.8	0	108461	680.2	3	171970	1852.9
B-bip	0	971	9.3	0	15871	105.6	0	18587	202.4
B-nonzero	0	7103	57.0	0	108460	817.3	3	167342	2186.7
B-neigh-comp	0	1230	10.4	0	17762	136.3	0	19488	264.0
B-bip-comp	0	971	9.3	0	15550	105.5	0	18099	202.0
B-neigh-strong	0	373	93.4	0	22174	762.8	2	42756	1834.3
B-bip-strong	0	247	39.6	0	3725	341.6	0	4869	661.6
B-bip-noC-bds	0	229503	701.8	40	1796328	7200.0	30	1127433	7200.0
B-bip-noC-strBds	–	–	–	–	–	–	–	–	–
B-bip-C-bds10	0	628	8.1	0	3020	41.0	0	3665	82.8
B-bip-C-edge	0	553	20.1	0	6001	84.8	0	7143	161.6
B-bip-C-neigh	0	559	29.7	0	6702	98.0	0	7694	189.2
B-bip-C-clique	0	586	20.2	0	6588	89.0	0	7723	171.5
B-bip-C-bip	0	565	30.8	0	6486	101.3	0	8359	195.8
B-bip-C-box	–	–	–	–	–	–	–	–	–
B-bip-C-cover	0	1022	13.2	0	15573	105.2	0	18317	208.2
B-bip-favor	0	419	19.5	0	1819	53.6	0	1990	99.8

mentarity constraints increases the time needed for node-switching. Moreover, for the MFPC and CKPC instances, adding complementarity constraints essentially does not make a difference. As before, one can explain this with the specific structure of the conflict graph of these instances. We conclude that adding complementarity constraints only is an option for hard instances with a structure similar to the TPESC instances.

Experiment 3: selection rules In this last experiment regarding branching rules, we analyze the behavior of strong branching ("B-bip-strong" and "B-neigh-strong"). The outcome reflects the expected behavior: On the one hand, strong branching significantly reduces the number of branching nodes. On the other hand, it is very time consuming; on average, it uses around 20% of the total CPU time. Nevertheless, for the MFPC type two and three instance sets, strong branching reduces the CPU time or solves more instances within the limits. For the TPESC type three instances, this is only true w.r.t. the neighborhood branching setting. Summarizing, the results indicate that strong branching can be useful for certain instances that are hard to solve, but in the presented form it should not be used as default strategy. A careful fine-tuning will be necessary to keep the execution time of strong branching in balance.

Experiment 4: cutting planes For the cutting plane experiments, we use neighborhood branching for the MFPC and CKPC and bipartite branching for the TPESC instance

Table 5 Experiments on the CKPC instances (nodes and time in shifted geometric mean)

Setting	CKPC type 1 (30)			CKPC type 2 (30)			CKPC type 3 (20)		
	Limit	Nodes	Time	Limit	Nodes	Time	Limit	Nodes	Time
B-SOS1	30	1069555	7200.0	30	575127	7200.0	20	521383	7200.0
B-neigh	0	50291	119.0	0	360323	1133.3	0	551661	1780.1
B-bip	0	50270	126.0	0	360229	1195.8	0	551531	1880.4
B-nonzero	0	50291	119.3	0	360324	1139.3	0	551661	1776.7
B-neigh-comp	0	50290	119.7	0	360317	1134.9	0	551652	1782.4
B-bip-comp	0	50268	126.8	0	360221	1197.5	0	551519	1876.0
B-neigh-strong	0	33423	184.0	0	306294	1336.1	0	480013	2018.6
B-bip-strong	0	33416	190.2	0	306230	1392.9	0	479926	2109.3
B-neigh-noC-bds	0	13949991	1622.2	30	52249068	7200.0	20	51160687	7200.0
B-neigh-noC-strBds	–	–	–	–	–	–	–	–	–
B-neigh-C-bds10	0	28038	92.3	0	195702	827.7	0	303616	1307.3
B-neigh-C-edge	0	38523	141.5	0	286272	1298.4	0	438290	2013.2
B-neigh-C-neigh	0	37116	151.9	0	274376	1328.0	0	420269	2049.5
B-neigh-C-clique	0	37157	144.7	0	277114	1326.7	0	418592	2032.4
B-neigh-C-bip	0	37116	152.0	0	274376	1328.3	0	420269	2052.5
B-neigh-C-box	–	–	–	–	–	–	0	472244	1986.7
B-neigh-C-cover	–	–	–	–	–	–	–	–	–
B-neigh-favor	0	24979	118.3	0	171534	1004.6	0	262269	1547.4

Table 6 Different options for cutting planes

Shortcut	Explanation
bds	Bound cuts (Sect. 5.2)
bds10	Bound cuts separated with a node-depth frequency of 10
strBds	Strengthened bound cuts (Sect. 5.2)
edge	Disjunctive cuts w.r.t. edges (Sect. 5.1.1)
neigh	Disjunctive cuts w.r.t. neighborhoods (Sect. 5.1.1)
clique	Disjunctive cuts w.r.t. cliques (Sect. 5.1.1)
bip	Disjunctive cuts w.r.t. bipartite subgraphs (Sect. 5.1.1)
box	Disjunctive cuts w.r.t. boxes (Sect. 5.1.2)
cover	Cover cuts (Sect. 5.3)

sets. In the related part of Tables 3, 4 and 5, we report on the results with nine settings "B-<branching>-noC-<cutting>" and "B-<branching>-C-<cutting>". Here, the term <branching> takes the choice from {neigh, bip} and <cutting> from the options listed in Table 6. The terms "C" and "noC" declare whether the corresponding cutting plane is turned on or off, in contrast to the default setting. All cutting planes were separated only in the root node, with the exception of "B-<branching>-C-bds10",

where we separate bound inequalities as local cuts with a node-depth frequency of 10 by using locally valid bounds on the variables.

The separation time of all classes of cutting planes for MFPC and CKPC was negligible compared to the total CPU time. For TPESC, the separation time was significant, e.g., for "B-bip-C-edge" separation used 35.9% (7.1 s), 10.9% (8.3 s), and 9.4% (13.7 s) of the total time on average for types 1, 2, and 3, respectively. This is partly due to the fact that these instances are easier to solve and thus take less nodes, increasing the relevance of the time for the root node.

In our experiments, "stalling" was an issue: A separation round is said to *stall*, if it does not cause a change in the dual objective bound. Nevertheless, sometimes it is necessary to overcome some stalling rounds in order to generate efficient cuts; reasons can be symmetry or structures like bottlenecks in flow problems. We therefore allow for a maximal number of 200 stalling rounds.

In order to avoid numerical instabilities, we discard disjunctive cuts that have a high rank: We say a disjunctive cut has rank $r + 1$ if it is constructed from simplex tableau rows whose coefficients are affected by some rank-r cut from a former separation round, where r is maximal. In our default setting, we limit the maximum permissible rank to be at 20.

If separation of bound inequalities is turned off ("B-<branching>-noC-bds"), we observe a significant increase in the CPU time. Moreover, the same holds for strengthened bound inequalities for the MFPC instances, see row "B-neigh-noC-strBds" in Table 3. For the other instances, strengthened bound cuts are not relevant, and the respective rows "B-<branching>-noC-strBds" are left empty in Tables 4 and 5.

It is important to note that box disjunctive cuts have relevance for CKPC type three, but no relevance for CKPC type one and two, see the way the instances of these types are generated at the end of Sect. 6.1.3. Summarizing the results of all classes of disjunctive cuts, we see that they bring benefit in the CPU time for the harder (type two and three) MFPC and TPESC instances. For the CKPC instances, they have a very positive effect w.r.t. the number of nodes, but not w.r.t. the CPU time. Obviously, here the cutting planes cannot compensate the larger solving time of the LP-relaxations. If we compare among the different classes of disjunctive cuts, it is easy to identify edge disjunctive cuts as a clear favorite among them. Moreover, separation of cover inequalities only has significance for the TPESC instance sets. However, they do not have a measurable effect on the performance. In the following, we analyze this behavior in more detail with the help of Table 7.

The columns in Table 7 have the same names as the <cutting> choices. Each of them refers to the setting in which the corresponding class of cutting planes is turned on, in addition to (strengthened) bound cuts, which are always applied here. For each instance class, we present the number of applied cuts in shifted geometric mean with a shift of 10. We highlight the following aspects:

– As expected, a relatively large number of the very efficient (strengthened) bound inequalities were added. Clearly, this number rigorously increases if separation is performed with a node-depth frequency of 10.

Table 7 Number of cutting planes applied (in shifted geometric mean)

Problem	Type	bds	bds10	Edge	Neigh	Clique	Bip	Box	Cover
MFPC	1	361.8	731.5	52.2	32.2	37.2	28.5	4.2	–
	2	248.7	2936.9	52.7	29.6	36.8	25.5	6.8	–
	3	375.4	5102.9	40.6	21.5	20.9	19.4	3.0	–
TPESC	1	532.9	1272.9	47.7	36.3	43.2	27.0	–	59.3
	2	607.3	9630.5	58.3	35.7	48.6	30.4	–	85.3
	3	856.0	11238.0	62.6	39.4	54.0	30.4	–	146.9
CKPC	1	780.4	38225.7	20.2	20.0	20.0	20.0	–	–
	2	1032.0	196744.8	20.1	20.1	20.0	20.1	–	–
	3	1079.1	273866.2	20.1	20.1	20.0	20.1	20.1	–

- For the CKPC instances, almost always 20 disjunctive cuts were added. This is due to the restriction of the maximal possible rank number to be at most 20. Because there is only a single linear (knapsack) inequality, disjunctive cuts often recursively depend on the previously separated cuts. Furthermore the LP solution often violates only one complementarity constraint, see the discussion in Sect. 5.1.2.
- For the MFPC and TPESC instances, from all classes of disjunctive cuts, the most inequalities were added for edge disjunctive cuts. This has several reasons:
  First, the rank of cuts from the other classes of disjunctive inequalities grows faster, since they are usually generated from more rows of the simplex tableau.
  However, even if there are no rank restrictions, edge disjunctive cuts empirically produced the best results: edge cuts tend to produce stronger cuts, since they do not aggregate inequalities as in (5.3). By solving a (relatively large) cut generating linear program, as investigated in the article of Balas and Perregaard [9], one can avoid this aggregation. However, until now, we have not tested this in practice.
  A further reason for the low number of added neighborhood, clique, and bipartite cuts is that they aggregate several edge cuts in one single inequality.
- The fewest disjunctive inequalities were added for box cuts. This can be explained by the fact that they only can be applied to a very special structure, appearing less often for the considered problem instances.
- Finally, we have a closer look at the number of added cover inequalities. Although this number is relatively large, our tests have shown that the separation of cover inequalities does not yield significant changes in the root gap. One possible reason for this could be that for separation, we reduce the conflict graph to a clique cover and do not take the overlapping structure of the SOS1 constraints into account. A second reason could be that we do not make use of a lifting procedure to improve our cutting planes.

Experiment 5: heuristics After evaluation of the former experiments, we decided to use the combination of "B-neigh", "C-bds10", and "C-edge" as our favorite setting for the MFPC and CKPC instances and the combination of "B-bip-comp", "C-bds10", and "C-edge" as the one for the TPESC instances. On average, these settings clearly pro-

duced the fewest number of nodes and seem to provide a relatively good compromise, although they are not the fastest for all instances. In the last row of each of the Tables 3, 4 and 5, we report on the results using our favorite settings ("B-<branching>-favor"). We also use these favorite settings in our fifth experiment, which we explain in the following.

For each test run of this experiment, we turn different heuristics on and do not initialize the algorithm with a precomputed optimal solution. We use the following abbreviations as column labels for Tables 8 and 9: "none" for using none of the heuristics, "MWIS" for maximum weighted independent set fixing, "viol" for violation diving, "obj" for objective shift diving, "frac" for fractional diving, and "frac-LS" for fractional diving in combination with local search.

To compare the performance of the heuristics, we make use of the *primal-dual integral* (see Berthold [12]). The primal-dual integral is the value we obtain by integrating the gap g between the primal bound p and dual bound d over time, where $g := 100 \cdot |p - d| \,/\, \max(|d|, |p|)$. To ensure a fair comparison, we divide the primal-dual integral by the maximum total CPU time among the six variants from the column labels. Thus, a smaller scaled primal-dual integral indicates a higher solution quality.

Table 8 Scaled primal-dual integral for test runs with different heuristics

Problem	Type	None	MWIS	Viol	Obj	Frac	Frac-LS
MFPC	1	4.4	4.3	4.8	5.0	4.3	4.5
	2	5.0	5.1	5.1	5.0	4.8	5.0
	3	3.2	3.2	3.4	3.5	3.3	3.4
TPESC	1	56.7	47.2	45.6	46.1	44.4	44.7
	2	69.5	14.3	15.8	15.0	13.3	14.9
	3	64.0	14.5	15.3	14.4	12.3	12.4
CKPC	1	19.6	17.9	18.4	18.4	16.7	16.9
	2	17.2	16.8	17.1	17.1	16.5	16.4
	3	17.0	16.9	16.9	17.0	16.7	16.5

Table 9 Percentage of relative execution time of heuristics w.r.t. the total CPU time

Problem	Type	MWIS	Viol	Obj	Frac	Frac-LS
MFPC	1	0.3	2.2	2.0	0.9	1.4
	2	0.3	0.9	0.8	0.6	1.3
	3	0.1	0.9	1.0	0.5	1.1
TPESC	1	22.56	28.1	26.8	23.1	25.3
	2	7.5	21.8	20.7	6.9	12.8
	3	10.9	23.5	22.4	8.1	13.4
CKPC	1	1.8	2.8	2.8	0.8	1.6
	2	0.4	0.7	0.7	0.1	0.3
	3	0.3	0.5	0.5	0.1	0.2

Table 8 shows the scaled primal-dual integral on average over all the instances of each problem type. The provided information can be summarized as follows:

- For the MFPC instances, the solution quality is already relatively high for the test run using none of the heuristics. This partly explains the fact why the heuristics do not bring a significant benefit on these instances. Nevertheless, there is still a potential for improvement. We think that one needs a heuristic which takes greater account of the equality system $Ax = b$, i.e., prefer variable fixings that not only reduce the violation of the complementarity constraints, but also maintain the feasibility for the linear constraints.
- On the other hand, for the TPESC instances, all heuristics seem to perform well, especially for the harder type two and three instances. This motivates us to accept the relatively high execution time (see Table 9 below).
- For the CKPC instances, only slight improvements can be achieved. This can be explained with the help of Table 10. Here, column "heur" denotes the ratio of the CPU time without precomputed optimal solutions, but with the use of fractional diving as primal heuristic, to the CPU time with precomputed optimal solutions. Since the ratio is nearly 1.1 for the CKPC instance types, this shows that the use of heuristics generally cannot lead to major improvements on these instance sets.
- Fractional diving seems to perform best among the presented heuristics. If one combines it with local search, then this only leads to slight improvements for the CKPC type two and three instances. In all other cases, the solution does not significantly improve, but the increased run time leads to a deterioration of the primal integral. For the future, one could investigate a tabu search approach (see Glover [26]) which might have the potential to enhance the performance of our local search algorithm.

The data of Table 9 refers to the relative execution time of the heuristics in percent. For MFPC and CKPC the execution time is negligible compared to the total CPU time. On the other hand, for TPESC, it has a much larger share of the run time. Therefore, we recommend to use only one of the heuristics and not multiple of them at once. Among the diving heuristics, fractional diving is the fastest, since it usually fixes more variables to zero in each diving LP iteration, see Sect. 4.1.

Experiment 6: comparison with other MIP-solvers Finally, in Table 10 we compare the performance of SCIP and CPLEX with default settings on our instance sets. We apply both to (LPCC) formulated with SOS1 constraints and (MIPPC) formulated as an MIP with packing constraints. The resulting four variants are denoted as "SCIP-SOS1", "SCIP-MIP", "CPLEX-SOS1", and "CPLEX-MIP". Especially for "SCIP-SOS1", we make use of the new solver components using the "B-<branching>-favor" settings with fractional diving as primal heuristic.

It is noticeable that SCIP-SOS1 considerably outperforms the other branch-and-cut solvers. Surprisingly, CPLEX-MIP performs better in comparison to CPLEX-SOS1, which for many instances does not even find a feasible solution.

Note that the results presented in Table 10 are meant to ensure that black-box solvers do not outperform our newly developed methods. In this sense, the comparison is unfair, since neither CPLEX nor SCIP-MIP was tuned towards the used instances.

Table 10 Results of SCIP and CPLEX applied to (LPCC) in column "SCIP-SOS1"/"CPLEX-SOS1" and MIPPC in column "SCIP-MIP"/"CPLEX-MIP", where "SCIP-SOS1" denotes the implementation of this article

Problem	Type	#	SCIP-SOS1				SCIP-MIP			CPLEX-SOS1			CPLEX-MIP		
			Limit	Nodes	Time	Heur	Limit	Nodes	Time	Limit	Nodes	Time	Limit	Nodes	Time
MFPC	1	20	0	28858	112.7	1.4	20	473690	7200.0	20	6195286	7200.0	20	519160	7200.0
	2	20	0	96279	495.8	1.5	20	1224679	7200.0	20	7128394	7200.0	18	1668921	6774.0
	3	20	0	33048	611.7	1.8	20	30200	7200.0	20	3552629	7200.0	20	37304	7200.0
TPESC	1	20	0	455	23.1	1.2	1	6855	4590.3	20	2829542	7200.0	0	991	749.8
	2	40	0	1925	66.1	1.2	32	9398	6379.2	40	2794235	7200.0	18	5346	3967.2
	3	30	0	2138	125.8	1.3	30	3834	7200.0	30	1880248	7200.0	27	3102	6476.4
CKPC	1	30	0	28115	130.2	1.1	0	232593	1539.8	30	6565227	7200.0	0	61741	325.4
	2	30	0	202542	1133.3	1.1	30	797825	7200.0	30	5563063	7200.0	0	417630	3029.3
	3	20	0	305769	1730.7	1.1	20	730071	7200.0	20	65452334	7200.0	1	594269	5019.8

The number of nodes and the CPU time are presented in shifted geometric mean

Table 11 Results of SCIP-SOS1 applied to (LPCC) as well as SCIP-MIP and CPLEX-MIP applied to (MIPPC) on 27 additional instances

Problem	#	SCIP-SOS1			SCIP-MIP			CPLEX-MIP		
		Limit	Nodes	Time	Limit	Nodes	Time	Limit	Nodes	Time
MFPC	9	0	62030	339.1	8	53967	3587.5	7	110269	3442.0
TPESC	9	0	555	25.3	2	1254	854.5	2	546	288.3
CKPC	9	0	140735	469.3	6	509160	2370.5	0	328548	1377.0

While our solver has evaluated on the given instances in the previous experiments, we tried not to overtune the code on these particular instances.

In order to support this claim, we randomly generated nine additional instances for MFPC, TPESC, and CKPC, each, where three are of type 1, 2 or, 3 each. Thus, these instances use the model, but contain different coefficients and SOS1 constraints. We then repeated the experiment of Table 10 on these instance, but also using a different computer (an Intel i7-5820 CPU running at 3.30GHz). The tests were run with a time limit of one hour. The results are given in Table 11 and reflect the same behavior as in Table 10. The only difference is that the new instances can be solved a bit faster on average. Note that for the previous experiments, we removed instances that can be solved very quickly by our solver.

Moreover, note that SCIP-SOS1 is faster than the other solvers, as long as bound cuts and either neighborhood or bipartite branching is used. On the other hand, there are instances, where the other solvers outperform SCIP-SOS1. In fact, the main reason why the LPCC formulation often performs better than the MIPPC formulation is the smaller problem size. The LPCC formulation is particularly advantageous if the bounds of the variables are not tight. In this case, only a few number of bound inequalities are separated, and although the LP-relaxation of (LPCC) has less variables and constraints, it is approximately as tight as the LP-relaxation of (MIPPC). However, even if the bounds are not tight, the LPCC formulation can be beneficial. This is demonstrated for example by the CKPC instances: As knapsack instances, they always have an optimal solution in which all but one variable are either zero or at their upper bound. Nevertheless, the large number of set packing constraints deteriorates the solving time for the LP-relaxation of (MIPPC) for CKPC (the conflict graph density is 35%, see Table 1); in (LPCC) the conflict graph is used for separation and usually not completely represented in the LP-relaxation.

7 Conclusion

In this article, we discussed a branch-and-cut algorithm for solving LPs with overlapping SOS1 constraints. We exploited the structure of the corresponding conflict graph for developing branching rules, preprocessing, primal heuristics, and cutting planes. The algorithm was tested with different settings on randomly generated problem instances from three different applications.

The computational experience can be summarized as follows: The performance of a branching rule strongly depends on the specific structure of the problem instance. For conflict graphs consisting of large cliques that predominantly overlap with one another (as it is the case for the MFPC instances), neighborhood branching performs best. The same is true for completely random and disordered conflict graphs (as for the CKPC instances). For conflict graphs that consist of groups of vertices with identical or similar neighborhood properties (as for the TPESC instances), the more balanced bipartite branching performs better.

Moreover, locally adding complementarity constraints reduces the number of nodes for some instances, but does in general not reduce the solving time in comparison to the best branching rule. Furthermore, using strong branching significantly reduces the number of nodes, but in its current version often increases the solution time.

Among the variety of tested cutting planes, bound inequalities (not surprisingly) are the most important, since they provide a representation of the complementarity constraints in the LP relaxation. The disjunctive cuts clearly reduce the number of nodes, but sometimes increase the solution time.

For the considered problem instances, our algorithm outperforms SOS1 branching by Beale and Tomlin and the big-M reformulation of (LPCC) both in SCIP and CPLEX. This indicates the effectiveness of directly enforcing the SOS1 constraints without introducing auxiliary binary variables for the considered instances. Recall, however, that there exist problem instances for which the techniques considered in this paper do not improve w.r.t. the big-M model.

Most of our implementation is already part of the recent SCIP release (SCIP 3.2). With the insights gained from this article, we are currently working on the handling of cardinality constraints. In the future, we will investigate how to derive strong cover inequalities taking the overlapping structure of the SOS1 constraints directly into account. Furthermore, one could think of combining our branching strategy with the standard reliability branching approach (see Achterberg et al. [3]). Since bipartite branching is directly applicable to MIPs with conflict constraints, it would be interesting to test how this branching rule performs on a general test set.

Acknowledgements We are grateful to the referees for their detailed comments, which helped us to improve the paper. Furthermore, we thank Norbert Fabritius for the implementation of test instance generators. The work of Tobias Fischer is supported by the Excellence Initiative of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt.

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. Thesis, Technical University Berlin (2007)
2. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)
3. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**, 42–54 (2004)
4. Agra, A., Doostmohammadi, M., de Souza, C.C.: Valid inequalities for a single constrained 0–1 MIP set intersected with a conflict graph. Discrete Optim. **21**, 42–70 (2016)
5. Atamtürk, A., Nemhauser, G.L., Savelsbergh, M.W.P.: Conflict graphs in solving integer programming problems. Eur. J. Oper. Res. **121**(1), 40–55 (2000)
6. Atamtürk, A., Nemhauser, G.L., Savelsbergh, M.W.P.: The mixed vertex packing problem. Math. Program. **89**(1), 35–53 (2000)

7. Audet, C., Savard, G., Zghal, W.: New branch-and-cut algorithm for bilevel linear programming. J. Optim. Theory Appl. **134**(2), 353–370 (2007)
8. Baker, B.S., Coffman Jr., E.G.: Mutual exclusion scheduling. Theor. Comput. Sci. **162**(2), 225–243 (1996)
9. Balas, E., Perregaard, M.: A precise correspondence between lift-and-project cuts, simple disjunctive cuts and mixed integer gomory cuts for 0–1 programming. Math. Program. **94**(2), 221–245 (2003)
10. Beale, E.M.L., Tomlin, J.A.: Special facilities in general mathematical programming system for non-convex problems using ordered sets of variables. In: Lawrence, J. (ed.) Proceedings of the 5th International Conference on Operations Research, pp. 447–454. Travistock Publications, London (1970)
11. Benichou, M., Gauthier, J.M., Hentges, G., Ribiere, G.: The efficient solution of large-scale linear programming problems-some algorithmic techniques and computational results. Math. Program. **13**(1), 280–322 (1977)
12. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013)
13. Berthold, T.: Heuristic Algorithms in Global MINLP Solvers. Ph.D. Thesis, TU Berlin (2014)
14. Bonami, P., Gonçalves, J.P.: Heuristics for convex mixed integer nonlinear programs. Comput. Optim. Appl. **51**(2), 729–747 (2012)
15. Borndörfer, R., Kormos, Z.: An algorithm for maximum cliques. Zuse Institute Berlin, 1997 (Unpublished Manuskript)
16. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Commun. ACM **16**(9), 575–577 (1973)
17. Cao, B.: Transportation problem with nonlinear side constraints a branch and bound approach. Z. Oper. Res. **36**(2), 185–197 (1992)
18. Dowsland, K.A.: Nurse scheduling with tabu search and strategic oscillation. Eur. J. Oper. Res. **106** (2–3), 393–407 (1998)
19. de Farias, I.R., Johnson, E.L., Nemhauser, G.L.: Branch-and-cut for combinatorial optimization problems without auxiliary binary variables. Knowl. Eng. Rev. **16**(1), 25–39 (2001)
20. de Farias, I.R., Johnson, E.L., Nemhauser, G.L.: Facets of the complementarity knapsack polytope. Math. Oper. Res. **27**(1), 210–226 (2002)
21. de Farias, I.R., Kozyreff, E., Zhao, M.: Branch-and-cut for complementarity-constrained optimization. Math. Program. Comput. **6**(4), 365–403 (2014)
22. de Farias, I.R., Nemhauser, G.L.: A polyhedral study of the cardinality constrained knapsack problem. Math. Program. **96**(3), 439–467 (2003)
23. Fischer, T., Pfetsch, M.E.: On the structure of linear programs with overlapping cardinality constraints. Technical report, Available on Optimization Online (2017)
24. Forrest, J.J.H., Hirst, J.P.H., Tomlin, J.A.: Practical solution of large mixed integer programming problems with UMPIRE. Manag. Sci. **20**(5), 736–773 (1974)
25. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
26. Glover, F.: Future paths for integer programming and links to artificial intelligence. Comput. Oper. Res. **13**(5), 533–549 (1986). Applications of Integer Programming
27. Gupta, P., Kumar, P.R.: The capacity of wireless networks. IEEE Trans. Inf. Theory **46**(2), 388–404 (2000)
28. Hamdi, K., Labadi, N., Yalaoui, A.: An iterated local search algorithm for the vehicle routing problem with conflicts. In: 8th International Conference of Modeling and Simulation—MOSIM10, pp. 1203–1211 (2010)
29. Hifi, M., Michrafy, M.: Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. Comput. Oper. Res. **34**(9), 2657–2673 (2007)
30. Hoheisel, T., Kanzow, C., Schwartz, A.: Theoretical and numerical comparison of relaxation methods for mathematical programs with complementarity constraints. Math. Program. **137**, 257–288 (2013)
31. Hu, J., Mitchell, J.E., Pang, J.S., Bennett, K.P., Kunapuli, G.: On the global solution of linear programs with linear complementarity constraints. SIAM J. Optim. **19**(1), 445–471 (2008)
32. Hu, J., Mitchell, J.E., Pang, J.S., Yu, B.: On linear programs with linear complementarity constraints. J. Glob. Optim. **53**(1), 29–51 (2012)
33. Hummeltenberg, W.: Implementations of special ordered sets in MP software. Eur. J. Oper. Res. **17**(1), 1–15 (1984)
34. Ibaraki, T.: Approximate algorithms for the multiple-choice continuous knapsack problem. J. Oper. Res. Soc. Jpn. **23**(1), 28–62 (1980)

35. Ibaraki, T., Hasegawa, T., Teranaka, K., Iwase, J.: The multiple-choice knapsack problem. J. Oper. Res. Soc. Jpn. **21**(1), 59–95 (1978)
36. Jain, K., Padhye, J., Padmanabhan, V.N., Qiu, L.: Impact of interference on multi-hop wireless network performance. Wirel. Netw. **11**(4), 471–487 (2005)
37. Jansen, K.: An approximation scheme for bin packing with conflicts. In: Arnborg, S., Ivansson, L. (eds.) Algorithm Theory—SWAT'98. Lecture Notes in Computer Science, vol. 1432, pp. 35–46. Springer, Berlin, Heidelberg (1998)
38. Jeroslow, R.G.: Representability in mixed integer programming, I: characterization results. Discrete Appl. Math. **17**(3), 223–243 (1987)
39. Júdice, J.J., Sherali, H.D., Ribeiro, I.M., Faustino, A.M.: A complementarity-based partitioning and disjunctive cut algorithm for mathematical programming problems with equilibrium constraints. J. Glob. Optim. **36**(1), 89–114 (2006)
40. Lin, E.Y.H.: Multiple choice knapsack problems and its extensions on capital investment. Oper. Res. Appl. **2**, 406–417 (1998)
41. Murty, K.G.: Linear Complementarity, Linear and Non Linear Programming. Sigma Series in Applied Mathematics. Heldermann Verlag, Berlin (1988)
42. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley Series in Discrete Mathematics and Optimization. Wiley, London (1999)
43. Owen, G.: Cutting planes for programs with disjunctive constraints. J. Optim. Theory Appl. **11**(1), 49–55 (1973)
44. Pferschy, U., Schauer, J.: The knapsack problem with conflict graphs. J. Graph Algorithms Appl. **13**(2), 233–249 (2009)
45. Sakai, S., Togasaki, M., Yamazaki, K.: A note on greedy algorithms for the maximum weighted independent set problem. Discrete Appl. Math. **126**(2–3), 313–322 (2003)
46. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA J. Comput. **6**(4), 445–454 (1994)
47. SCIP: Solving Constraint Integer Programs. http://scip.zib.de
48. Shi, Y., Hou, Y.T., Liu, J., Kompella, S.: How to correctly use the protocol interference model for multi-hop wireless networks. In: MobiHoc '09: Proceedings of the Tenth ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 239–248. ACM, New York, USA (2009)
49. Sun, M.: A tabu search heuristic procedure for solving the transportation problem with exclusionary side constraints. J. Heuristics **3**(4), 305–326 (1998)
50. Syarif, A., Gen, M.: Solving exclusionary side constrained transportation problem by using a hybrid spanning tree-based genetic algorithm. J. Intell. Manuf. **14**(3–4), 389–399 (2003)
51. Tomlin, J.A.: Special ordered sets and an application to gas supply operations planning. Math. Program. **42**(1–3), 69–84 (1988)