A Branch-and-Cut Algorithm for Mixed-Integer Bilinear Programming

Matteo Fischetti[1] and Michele Monaci[2]

[1] DEI, Università di Padova, via Gradenigo 6/A, 35100 Padova (Italy)

[2] DEI "Guglielmo Marconi", Università di Bologna, Viale Risorgimento 2, 40136 Bologna (Italy)

e-mail: *matteo.fischetti@unipd.it, michele.monaci@unibo.it*

February 22, 2019; Revised, 20 September 2019

Abstract

In this paper we consider the Mixed-Integer Bilinear Programming problem, a widely-used reformulation of the classical mixed-integer quadratic programming problem. For this problem we describe a branch-and-cut algorithm for its exact solution, based on a new family of intersection cuts derived from bilinear-specific disjunctions. We also introduce a new branching rule that is specifically designed for bilinear problems. We computationally analyze the behavior of the proposed algorithm on a large set of mixed-integer quadratic instances from the MINLPlib problem library. Our results show that our method, even without intersection cuts, is competitive with a state-of-the-art mixed-integer nonlinear solver. As to intersection cuts, their extensive use at each branching node tends to slow down the solver for most problems in our test bed, but they are extremely effective for some specific instances.

Keywords: (O) mixed-integer quadratic programming, bilinear programming, branch-and-cut algorithms, intersection cuts, computational experiments.

1 Introduction

We consider a *Mixed-Integer Bilinear Programming* problem (MIBLP) of the following form

$$(MIBLP) \qquad \min_x c^T x \tag{1}$$
$$Ax = b \tag{2}$$
$$\ell_j \le x_j \le u_j, \quad j = 1, \dots, n \tag{3}$$
$$x_j \text{ integer}, \quad j \in \mathcal{I} \tag{4}$$
$$x_{r_k} = x_{p_k} x_{q_k}, \quad k = 1, \dots, K, \tag{5}$$

where $x \in \Re^n$, matrix (A, b) has m rows and $n+1$ columns, and ℓ and u are lower/upper bound vectors in $\Re^n$, and $\mathcal{I} \subseteq N := \{1, \ldots, n\}$ is the (possibly empty) index set of the integer variables. Finally, a triple of indices (p_k, q_k, r_k) in N is given for each $k = 1, \ldots, K$, for which the bilinear equation (5) is imposed. Case $p_k = q_k$ is allowed, i.e., the model can handle quadratic terms of the type $x_j = x_i^2$. We assume that variables x_{pk} and x_{qk} have finite bounds—possibly implied by other constraints.

Problem (1)–(4), obtained removing constraints (5) from the model, will be referred to as the *bilinear relaxation*. This relaxation turns out to be a Mixed-Integer Linear Programming problem (MILP). A solution x that is feasible for the bilinear relaxation and does not satisfy inequalities (5) will be called *bilinear infeasible*.

Problem MIBLP is stated in a form where the bilinear and quadratic terms are isolated from the linear ones, and are defined through the bilinear equations (5). It is self-evident however that MIBLP can model (among others) a completely-general mixed-integer quadratic programming problem of the form

$$(MIQP) \qquad \min a_0^T x + x^T Q^0 x \tag{6}$$
$$a_k^T x + x^T Q^k x \mathbin{@} b, \quad k = 1, \ldots, m \tag{7}$$
$$\ell_j \le x_j \le u_j, \qquad j = 1, \ldots, n \tag{8}$$
$$x_j \text{ integer}, \qquad j \in \mathcal{I}, \tag{9}$$

with $@ \in \{\le, =, \ge\}$, hence it is strongly NP-hard.

MIBLPs play a relevant role in mixed-integer nonconvex optimization. A general approach for deriving convex relaxations to bilinear terms has been introduced by McCormick [30], allowing the development of branch-and-bound algorithms. Many efforts have been done in the literature to derive tight relaxations. The Reformulation-Linearization Technique (RLT), introduced in [36] for solving continuous bilinear problems, computes lower bounds based on the solution of linear programming problems in which some additional variables are used to linearize bilinear terms. Based on RLT, a branch-and-cut algorithm for nonconvex quadratically constrained quadratic programming problems was introduced in [5]. In [37], convex envelopes that involve all the bilinear terms simultaneously were considered; this characterization is based on disjunctive programming and does not introduce additional variables. In [1] real applications of MIBLPs to location and product distribution situations were considered, and solved by means of an exact algorithm that generates Benders' cuts on the fly and uses them to define a Lagrangian relaxation of the problem. Recently, MIBLPs in which all nonlinear terms involve the product of one continuous and one integer variable were addressed in [24]. For this special case, a reformulation obtained replacing an integer variable with its binary expansion is presented, and the convex hull of the underlying mixed integer linear set is analyzed. A branch-and-cut algorithm for a family of mixed-integer quadratic programming problems was computationally analyzed in [9], while intersection cuts for polynomial optimization were addressed in [8].

MIBLPs have a number of real-world applications and arise as subproblems in more complex optimization problems. A typical application of bilinear optimization is represented by the pooling problem (see, Foulds, Haugland and Jörnsten [21]), that can be seen as a combination of the classical network flow problem and of the blending problem (which is a special minimum cost network flow problem). This problem is fundamental in

many contexts, including petrochemical refining, wastewater treatment and mining, and requires to optimize the flow rates over a given network satisfying some side constraints, related to the quality on the final product composition. The network may also include some inter-pool links, in which case discrete decisions about the use of some intermediate components have to be taken. Alternative formulations, reformulation-linearization techniques and heuristic algorithms for different variants of the pooling problem were given, e.g., in [4, 26, 32, 15, 16], and [25].

Finally, we mention that other applications of MIBLPs can be found, e.g., in farm management [10], cutting and packing [13], and supply chain [34].

Solvers for generic (mixed-integer) nonlinear problems include SCIP [38], BARON [35], BONMIN [11], ANTIGONE [33], COUENNE [7], MOSEK [3], IPOPT [39], and Knitro [12], among others.

The paper is organized as follows. In Section 2 we present a branch-and-cut algorithm for the exact solution of MIBLP, embedding valid inequalities from the literature and a new branching rule based on spatial branching. Section 3 introduces a new family of valid inequalities belonging to the family of intersection cuts, and discusses possible ways to derive those cuts. Section 4 gives some implementation details of our algorithms, that are computationally evaluated in Section 5 using a large benchmark of instances from the literature. Finally, some conclusions are drawn in Section 6

2 A branch-and-cut algorithm for MIBLP

Our solution algorithm for MIBLP relies on the solution of the bilinear relaxation (1)–(4), in which nonlinear constraints (5) are initially removed. To this end, an effective branch-and-cut MILP solver is used, which computes an optimal solution of the Linear Programming (LP) relaxation (1)–(3) at the various nodes of the decision tree, and uses branching to impose the integrality requirements. The solution algorithm is enriched by the addition of locally-valid inequalities based on the bilinear conditions (5), that are generated on the fly, to improve the LP relaxation at the current node. In case of branching, we rely on the default branching decision of the MILP solver whenever this is possible, and use a bilinear-specific (spatial) branching strategy only as a last resort, i.e., in case the current solution satisfies integrality conditions (4) but is bilinear infeasible.

The main ingredients of our solution scheme are described in the next subsections. To simplify our presentation, with a little abuse of notation, we will sometimes denote the generic (single) bilinear equation (5) and the associated variables' bounds by

$$z = x\,y \tag{10}$$

$$\ell_x \leq x \leq u_x \tag{11}$$

$$\ell_y \leq y \leq u_y \tag{12}$$

where $x, y, z \in \Re$ denote scalar variables (possibly, $x \equiv y$).

2.1 McCormick inequalities

As already mentioned, our solution algorithm is based on the dynamic separation of locally valid inequalities which are used to strengthen the formulation and ensure that

the returned solution satisfies bilinear conditions (5). In this section we describe the classical McCormick inequalities ([31, 2]) associated with a triple of variables involved in the bilinear equations (5). A new class of inequalities, that contains intersection cuts ([6]) based on different bilinear-free polyhedra, will be described in Section 3.

It is known [31, 2] that the convex hull of the points $(x, y, z) \in \Re^3$ satisfying (10)–(12) is defined by the following *McCormick inequalities*:

$$\begin{aligned}
\text{mc1)} \quad z &\geq \ell_y x + \ell_x y - \ell_x \ell_y && (13)\\
\text{mc2)} \quad z &\geq u_y x + u_x y - u_x u_y && (14)\\
\text{mc3)} \quad z &\leq u_y x + \ell_x y - \ell_x u_y && (15)\\
\text{mc4)} \quad z &\leq \ell_y x + u_x y - u_x \ell_y && (16)
\end{aligned}$$

which are in turn obtained from the obvious condition

$$\begin{aligned}
(x - \ell_x)(y - \ell_y) &\geq 0 && (17)\\
(x - u_x)(y - u_y) &\geq 0 && (18)\\
(x - \ell_x)(y - u_y) &\leq 0 && (19)\\
(x - u_x)(y - \ell_y) &\leq 0 && (20)
\end{aligned}$$

by replacing xy by z. As such, these linear inequalities are tight only when x and/or y are at their lower or upper bound (in which case the left-hand side of one of (17)–(20) is exactly equal to zero), while in all other cases they give a lower or upper approximation of $z = xy$ whose quality depends on the distance of x and y from their bounds. These considerations show the importance of having reduced domains for the x and y variables, obtained either by preprocessing or by branching.

While McCormick inequalities are the best possible linear cuts in the (x, y, z) space for the general case, improved linear inequalities can be obtained when the feasible x and y variables must satisfy additional conditions such as $x \leq y$ or alike. A particularly relevant case arises when $x = y$, i.e., in the quadratic case $z = x^2$. Here, (17) and (18) are useless as they are dominated by the bound conditions, and can be replaced by the family of gradient cuts

$$z \geq x_0^2 + 2x_0(x - x_0), \quad \text{for each } x_0 \in \Re \qquad (21)$$

that describe the convex set $z \geq x^2$. As to (19), it coincides with (20) and gives the best-possible linear (upper) approximation of the nonconvex region $z \leq x^2$ over the close segment $[l_x, u_x]$.

2.2 Bilinear-specific branching rule

As already mentioned, our algorithm performs branching on integer variables, when possible. At every branching node where the current LP solution is integer but bilinear infeasible and cannot be cut by our cut separation procedures, we need to perform a branching operation. A typical branching criterion, known as *spatial branching*, consists of reducing the domain of a certain variable x_{p_k} (say) involved in the right-hand side of a violated bilinear equation (5); see, e.g., [7]. This is obtained by imposing the branching

condition $(x_{p_k} \leq \theta) \vee (x_{p_k} \geq \theta)$, where θ is a threshold value to be determined. Typical choices proposed in the literature are, e.g., $\theta = x^*_{p_k}$ or $\theta = (\overline{\ell}_{p_k} + \overline{u}_{p_k})/2$, where $\overline{\ell}_{p_k}$ and $\overline{u}_{p_k}$ are the lower and upper bounds of x_{p_k} at the current branching node, respectively. As already noticed, the rationale of these choices is that the reduced domain of the branching variable x_{p_k} will tighten the McCormick inequalities in the child nodes. We next propose an alternative definition of the threshold θ, based on the following argument.

To simplify notation, we will concentrate again on a single bilinear equation in the system (5), written as in (10)–(12). Given (x^*, y^*, z^*) at the current node, we compute the error

$$\rho^* := z^* - x^* y^*$$

whose absolute value is strictly positive in the case of interest. Assume $\rho^* > 0$ (the case $\rho^* < 0$ is similar). Our order of business is to branch on the condition

$$(x \leq \theta) \vee (x \geq \theta)$$

where $\theta = x^* - \delta$ and $\delta > 0$ is a parameter to be defined later (the cases with x and y swapped and/or $\theta = x^* + \delta$ are analogous).

By definition, in the left-hand branch the new upper bound condition $x \leq \theta$ will be violated by $\delta > 0$ by the current point (x^*, y^*, z^*). As to the right-hand branch, we will update the lower bound ℓ_x on variable x as follows

$$\ell_x = x^* - \delta$$

thus improving the McCormick inequalities involving ℓ_x because we are reducing the domain of x by removing the open segment $[l_x, \theta)$ from it—recall that smaller domains result into improved McCormick inequalities. Our idea is to require that at least one of these inequalities be violated by the same amount δ, so as to "balance" the two child nodes.

As an illustration, the McCormick inequality (19) becomes

$$(x - x^* + \delta)(y - u_y) \leq 0$$

in the right-hand branch (the other McCormick inequalities can be dealt with in a very similar way). Rewriting the above condition as

$$xy - xu_y - x^* y + x^* u_y + \delta y - \delta u_y \leq 0$$

and replacing xy by z leads to the McCormick inequality for the right-hand branch

$$z - xu_y - x^* y + x^* u_y + \delta y - \delta u_y \leq 0.$$

By imposing this inequality be violated by δ w.r.t (x^*, y^*, z^*) we obtain

$$z^* - x^* u_y - x^* y^* + x^* u_y + \delta y^* - \delta u_y = \delta$$

i.e.

$$(z^* - x^* y^*) + \delta y^* - \delta u_y = \delta$$

hence

$$\rho^* = \delta\,(1 + u_y - y^*)$$

Algorithm 1: Our branching procedure

Input : The bilinear-infeasible point x^* and the variable bounds $(\overline{\ell}, \overline{u})$ at the current node; the tolerance value ε for constraint violation;

Output: The branching variable $x_{\texttt{bvar}}$ and the corresponding threshold value θ for spatial branching;

1 $\delta := -\infty$; $\texttt{bvar} := -1$; $\theta := 0$;
2 **for** *each* $k \in \{1, \dots, K\}$ *with* $|x^*_{r_k} - x^*_{p_k} x^*_{q_k}| > \varepsilon$ **do**
3 $\quad$ `branch_score`(x^*, $\overline{\ell}$, $\overline{u}$, p_k, q_k, r_k, bvar, δ, θ);
4 $\quad$ `branch_score`(x^*, $\overline{\ell}$, $\overline{u}$, q_k, p_k, r_k, bvar, δ, θ);
5 **end**
6 **if** $(\theta < \overline{\ell}_{\texttt{bvar}} + 10\,\varepsilon$ **or** $\theta > \overline{u}_{\texttt{bvar}} - 10\,\varepsilon)$ **then** $\theta := (\overline{\ell}_{\texttt{bvar}} + \overline{u}_{\texttt{bvar}})/2$ **endif**;
7 **return** *(*bvar, θ*);*

Algorithm 2: function `branch_score`(x^*, $\overline{\ell}$, $\overline{u}$, ix, iy, iz, bvar, δ, θ)

$\rho^* = x^*_{iz} - x^*_{ix}\, x^*_{iy}$;
if *($\rho^* < -\varepsilon$)* **then** // x^*_{iz} too small: use McCormick ineq.s mc1–mc2 to increase it
$\quad$ $d := -\rho^*/(1 + x^*_{iy} - \overline{\ell}_{iy})$;
$\quad$ **if** *($d > \delta$)* **then** $\texttt{bvar} := ix$, $\theta := x^*_{ix} - d$, $\delta := d$ **endif**;
$\quad$ $d := -\rho^*/(1 + \overline{u}_{iy} - x^*_{iy})$;
$\quad$ **if** *($d > \delta$)* **then** $\texttt{bvar} := ix$, $\theta := x^*_{ix} + d$, $\delta := d$ **endif**;
end
if *($\rho^* > \varepsilon$)* **then** // x^*_{iz} too large: use McCormick ineq.s mc3–mc4 to reduce it
$\quad$ $d := \rho^*/(1 + \overline{u}_{iy} - x^*_{iy})$;
$\quad$ **if** *($d > \delta$)* **then** $\texttt{bvar} := ix$, $\theta := x^*_{ix} - d$, $\delta = d$ **endif**;
$\quad$ $d := \rho^*/(1 + x^*_{iy} - \overline{\ell}_{iy})$;
$\quad$ **if** *($d > \delta$)* **then** $\texttt{bvar} := ix$, $\theta := x^*_{ix} + d$, $\delta := d$ **endif**;
end

that finally yields

$$\delta = \rho^*/(1 + u_y - y^*) > 0$$

(recall that we assume $\rho^* > 0$). Given a set of violated bilinear conditions (5), we determine the branching condition (i.e., branching variable and value of θ) as the one associated with maximum value of δ.

Our detailed branching procedure is given in Algorithm 1, where the update function `branch_score` described in Algorithm 2 is used to update the best (i.e., maximum) achievable δ. Thus, the so-far best δ is initialized at Step 1 of Algorithm 1, and is updated at Steps 3 and 4 for each violated bilinear equation. Note that, at Step 4, the role of p_k and q_k (i.e., of x and y) is just swapped. As to Step 6, it avoids that the threshold θ is too close to one of the bounds. As to the violation tolerance, in our implementation we used $\varepsilon = 10^{-6}$.

3 Intersection cuts for bilinear programming

In this section we introduce a new class of valid inequalities for MIBLP based on intersection cuts. The separation of these inequalities relies on the fact that the solution to be cut corresponds to a vertex of the current LP relaxation.

3.1 Intersection cuts

Intersection cuts have been introduced in [6] in the 1970's, and are widely used in Mixed-Integer Programming; the reader is referred to Ch. 6 "Intersection Cuts and Corner Polyhedra" in [14] for a recent in-depth treatment of the subject.

The definition of an intersection cut (IC) violated by a given point $x^* \in \Re^n$ (say) requires the definition of two sets:

1. a cone pointed at x^* that contains all the bilinear-feasible solutions; and
2. a convex set (a polyhedron, in our case) S containing x^* but including no bilinear-feasible solutions in its interior.

Note that the latter condition replaces the usual "lattice free" property (exploited in classical MILPs) with the bilinear-free one.

It is well known that, in the definition of an IC, the larger S, and the smaller the cone, the better the resulting IC. As to the first requirement, we observe that, as customary in mixed-integer programming, our ICs are generated for vertices of the current relaxation, so a suitable cone is just the corner polyhedron associated with the corresponding optimal basis. All relevant information about this cone is readily available in the "optimal tableau". As the LP relaxation at a given branching node exploits locally-valid information (notably, the reduced variable domain resulting from branching), our ICs are locally (as opposed to globally) valid.

The second point (namely, the definition of a proper set S), will be discussed in Subsection 3.2.

We next describe how we compute ICs in a numerically reliable way, by exploiting their "disjunctive interpretation" [22, 23]; see [18] for more details.

To ease exposition, let the LP model at the given branch-and-cut node be formulated in its standard form as

$$\min\{\hat{c}^T x : \hat{A}x = \hat{b}, x \geq 0\}.$$

Now let x^* be an optimal vertex of the above LP, associated with a certain basis $\hat{B}$ (say) of $\hat{A}$, and let the bilinear-free polyhedron S of interest be defined as

$$S = \{x : g_i^T x \leq g_{0i},\ i = 1, \ldots, k\}.$$

We aim at deriving a valid inequality, violated by x^*, from the feasibility condition "x cannot belong to the interior of S". To this end, we observe that the latter condition can be reformulated as the k-term disjunction:

$$\bigvee_{i=1}^{k} (g_i^T x \geq g_{i0}) \tag{22}$$

where we write $\geq$ instead of $>$ as a feasible x can in fact belong to the frontier of S. ICs are then obtained by the 3-step procedure below:

1. For $i = 1, \ldots, k$, take the equivalent "reduced form" of $g_i^T x \geq g_{i0}$ with respect to $\hat{B}$, namely $\overline{g}_i^T x \geq \overline{g}_{i0}$ where $(\overline{g}_i^T, \overline{g}_{i0}) = (g_i^T, g_{i0}) - u_i^T(\hat{A}, \hat{b})$ and $u_i^T = (g_i)_{\hat{B}}^T \hat{B}^{-1}$ guarantees that all basic variables have 0 coefficient in the reduced inequality.
2. Normalize all right-hand sides by dividing each reduced-form inequality by $\overline{g}_{i0}$, the latter term being strictly positive as $\overline{g}_i^T x \geq \overline{g}_{i0}$ is strictly violated by x^*, and $\overline{g}_i^T x^* = 0$ by construction.
3. Combine the resulting $\geq$ inequalities by taking, for each variable x_j, the maximum left-hand-side coefficients (while the right-hand-side is 1): the resulting inequality is a weakening of the single inequalities $g_i^T x \geq g_{i0}$, hence it is valid for (22) and violated (by 1) by x^*.

It is important to observe that, at Step 1, the validity of the final IC does *not* require the vectors $u_i^T = (g_i)_{\hat{B}}^T \hat{B}^{-1}$ be computed with a very high numerical precision, as the cut coefficients are computed by using the original data $(\hat{A}, \hat{b})$ and not the (possibly inaccurate) tableau information. This property is very important in practice, to ensure the correctness of the method—numerical issues in computing u_i^T can reduce cut violation but do not affect the validity of the final IC.

3.2 Bilinear-free sets

Given the solution to be separated, for each (x^*, y^*, z^*) such that $|z^* - x^*y^*|$ is larger than a given tolerance $\epsilon > 0$ (equal to 10^{-6} in our implementation), one can generate a bilinear-free set S by simulating a branching operation on x (or y) and by collecting a violated inequality for each branch.

To be more specific, assuming a k-way branch, the bilinear-free set S can be defined as

$$S := \{(x, y, z) : \alpha^t x + \beta^t y + \gamma^t z \leq \delta^t, \text{ for } t = 1, \ldots, k\} \tag{23}$$

where, for each $t = 1, \ldots, k$,

$$\alpha^t x + \beta^t y + \gamma^t z \geq \delta^t \tag{24}$$

is a valid inequality for the t-th branch which is strictly violated by (x^*, y^*, z^*). Note that the sense of the inequalities in (23) is reverted with respect to (24), which implies that no bilinear-feasible solution can belong to the interior of S, as required.

Our first bilinear-free set, S_1, is based on the customary 2-way branch

$$(x \leq x^*) \vee (x \geq x^*). \tag{25}$$

(the case with x and y swapped is analogous). For each branch, we separate the corresponding McCormick inequalities (13)–(16) for the updated lower/upper bounds on x, and pick the one that is most violated with respect to (x^*, y^*, z^*).

Our second bilinear-free set, S_2, is based on the 4-way branch

$$(x \leq x^*, y \leq y^*) \vee (x \leq x^*, y \geq y^*) \vee (x \geq x^*, y \leq y^*) \vee (x \geq x^*, y \geq y^*) \tag{26}$$

For each branch, we separate again the corresponding McCormick inequalities (13)–(16) for the updated lower/upper bounds on x and y, and pick the one that is most violated with respect to (x^*, y^*, z^*).

Other bilinear-free set S could be defined in an analogous way, by just changing the branching disjunction. E.g., one can use the branching rule described in Subsection 2.2, or even branch on more elaborated conditions like

$$((y - \ell_y)(x - y - x^* + y^*) \geq 0) \vee ((\ell_x - x)(x - y - x^* + y^*) \geq 0)$$

or

$$((x - \ell_x)(x + y - x^* - y^*) \leq 0) \vee ((x - u_x)(x + y - x^* - y^*) \leq 0)$$

where the term $x\,y$ is replaced by z, while x^2 and y^2 can be approximated by the gradient cuts $x^2 \geq (x^*)^2 + 2x^*\,(x - x^*)$ and $y^2 \geq (y^*)^2 + 2y^*\,(y - y^*)$, respectively. According to our preliminary tests, however, the corresponding intersection cuts do not seem to improve the performance of our solver in a significant way. Thus, we removed them from our code, to keep our implementation as clean as possible.

4 Branch-and-cut Implementation

In this section we briefly describe some implementation details that play an important role in the design of an effective code. Our description is based on the actual MILP solver we used (IBM ILOG CPLEX 12.8), but it extends easily to other solvers.

Our implementation solves a generic quadratic problem (6)–(9) written in its equivalent form (1)–(5), and infers simple bounds on the auxiliary x_{r_k} variables appearing in (5) from those of the corresponding x_{p_k} and x_{q_k} variables.

Our branch-and-cut algorithm is applied to the bilinear relaxation (1)–(4).

Due to the presence of callbacks, we need to work on the original variable space and disable dual presolve reductions and nonlinear presolve reductions. In addition, if IC separation is active we need to retrieve the LP basis at each node, which requires the complete deactivation of all preprocessing operations (CPX_PARAM_PREIND=0).

Our implementation is thread safe, so we can select 1-thread or multi-thread (deterministic or opportunistic) parallel mode depending on the kind of tests we want to perform.

Each time a new integer solution x^* is found and is going to update the incumbent, a specific "lazyconstraint callback" function is automatically invoked by CPLEX to let the user possibly discard this solution for whatever reason, and possibly add a cut that makes this solution infeasible. In this callback function, we first determine whether the solution is in fact bilinear-feasible. If this is not the case, we try to cut it by using a suitable McCormick/gradient or intersection cut. In case internal CPLEX's heuristics produced a bilinear-infeasible candidate solution, for which an associated LP basis is not available (hence, ICs cannot be derived), we just discard it. In our implementation, for each violated bilinear equation (5) we first look for a violated McCormick/gradient cut. If no such violated cut is found, we resort to IC separation by trying the bilinear-free sets S_1 first, and then the sets S_2—as defined in Subsection 3.2. In any case we generate, at most, one violated IC for each bilinear equation (5), discarding cuts that are numerically unreliable and hence error prone.

Immediately before branching, at each node CPLEX automatically invokes an "user-cut callback" function where violated McCormick/gradient or intersection cuts can be generated. In our default setting, however, this callback is not installed, meaning that we do not generate bilinear-specific cuts for fractional solutions but only rely on the internal CPLEX's cuts.

Note that CPLEX is not aware of the existence of bilinear constraints, so it would not branch on a solution that satisfies all the linear and integrality constraints (even if this solution is bilinear infeasible). To circumvent this issue, one needs to change the problem type to CPXPROB_MILP (even if the problem does not involve any integer variable) and to install a "branching callback" function that handles the bilinear-specific cases. In our implementation, this function receives on input the branching condition chosen by CPLEX: if this condition is not undefined (meaning that CPLEX knows how to branch), then we just let CPLEX branch as usual; otherwise we check whether the current solution is bilinear infeasible, in which case we apply our own branching rule.

5 Computational results

All the procedures described in the previous sections were coded in C language. To asses the performance of our branch-and-cut algorithm, we performed an extensive computational analysis on MIBLP instances from the literature. All the experiments were executed on an Intel Xeon E3-1220V2 running at 3.10 GHz, with 16 GB of RAM each, in single thread mode.

In the following, we compare the following approaches:

- `SCIP`: the direct application of the general-purpose solver SCIP [38] (version 5.0.1);
- `basic`: a branch-and-cut algorithm which implements our bilinear-specific branching rule (see Section 2.2), and separates McCormick inequalities only (see Section 2.1), without ICs;
- `with-IC`: a more sophisticated branch-and-cut algorithm obtained from the previous one by separating intersection cuts at each node where the LP solution x^* is integral, as described in Section 3.1.

As one of the main goals of our tests was the evaluation of the role of ICs for bilinear programming, and since our branch-and-cut schemes do not implement any bilinear-specific primal heuristic, to reduce performance variability [20, 29] we decided to provide on input to all solvers the optimal (or best-known) solution; this is customary when analyzing the effect of cutting planes, and has been done, among many others, in [19, 27].

We ran each algorithm with a time limit of 3,600 CPU seconds. While `SCIP` was applied directly to the quadratic model (6)–(9), algorithms `basic` and `with-IC` were executed on the bilinear reformulation (1)–(5) of the problem. All the three algorithms use IBM ILOG CPLEX (version 12.8) as LP solver; in addition, `SCIP` uses IPOPT [39] (version 3.12.9) as NLP solver.

`SCIP` and CPLEX have different default values for what concerns their internal parameters. In addition, differently from our branch-and-cut implementations, `SCIP` does not support multi-thread runs. Therefore, to have a fair comparison we had to find a

common setting for running the three codes. To this end, SCIP was run with its default parameters but for the relative and absolute optimality gap that were set to 10^{-4} and 10^{-6}, respectively (those being the default values for CPLEX). As for CPLEX, all internal parameters were left to their default value, but the integrality and feasibility tolerances (namely, CPX_PARAM_EPINT and CPX_PARAM_EPRHS) were set to 10^{-6} (as in the SCIP's default). In addition, the sequential (1-thread) CPLEX's mode was activated (CPX_PARAM_THREADS = 1), internal CPLEX's cuts were activated at their aggressive level (namely, level 2), and CPLEX's strong branching was selected.

Our benchmark was derived from the MINLPlib library, available at `http://www.minlplib.org/index.html`, and containing instances coming from different sources. We considered all the 768 instances that are available in .lp format, and removed all the instances for which the root-node bilinear relaxation (1)–(3) is unbounded—as already stated, our branch-and-cut scheme being not applicable in those situations. This produced our final testbed, containing 620 instances.

5.1 Preliminary experiments

During the development of our codes, we made a number of preliminary experiments with several possible variants. We next report some of those preliminary computational results.

Our first order of business was to evaluate the effectiveness of our new branching rule, as described in Algorithm 1, with respect to a more classical rule from the literature. We therefore implemented, using our CPLEX callback framework, the following branching strategy taken from [7]. Given the current bilevel-infeasible solution x^*, we define for each variable x_j a branching score σ^*_j that gives the total contribution of x_j to the violation of the bilinear equations (5), namely:

$$\sigma^*_j := \sum_{k \in \{1,\ldots,K\}: j \in \{p_k, q_k\}} |x^*_{r_k} - x^*_{p_k} x^*_{q_k}|.$$

We then compute $bvar := \arg\max\{\sigma^*_j : j \in N\}$ and branch on the disjunction $(x_{bvar} \leq \theta) \vee (x_{bvar} \geq \theta)$, where

$$\theta = \alpha\, x^*_{bvar} + (1-\alpha)\, \frac{\bar{\ell}_{bvar} + \bar{u}_{bvar}}{2}$$

and $\bar{\ell}_{bvar}$ and $\bar{u}_{bvar}$ are, respectively, the local lower and upper bound of x_{bvar} at the current branching node, with $\alpha = 0.25$ as in [7].

Figure 1 (top) plots a performance profile comparing `basic` with its modified version `mod-basic` (say) obtained by just replacing our branching rule with the branching strategy described above. The two versions hit the time limit more or less on the same MINLPlib instances in our testbed. Removing the time limits for both methods left 339 MINLPlib instances, on which `basic` had a significantly better performance in terms of wins, i.e., of numbers of instances for which one algorithm required strictly less computing time than the other one (`basic` had 222 wins, while `mod-basic` only 93 wins). All in all, as shown in the figure, our new branching rule performed better, although the difference was not striking for the hardest cases, hence we decided to use it in our codes.

Also related to branching, we implemented a variant of our code where McCormick inequalities (that are separated only for integer solutions) were used as local constraints

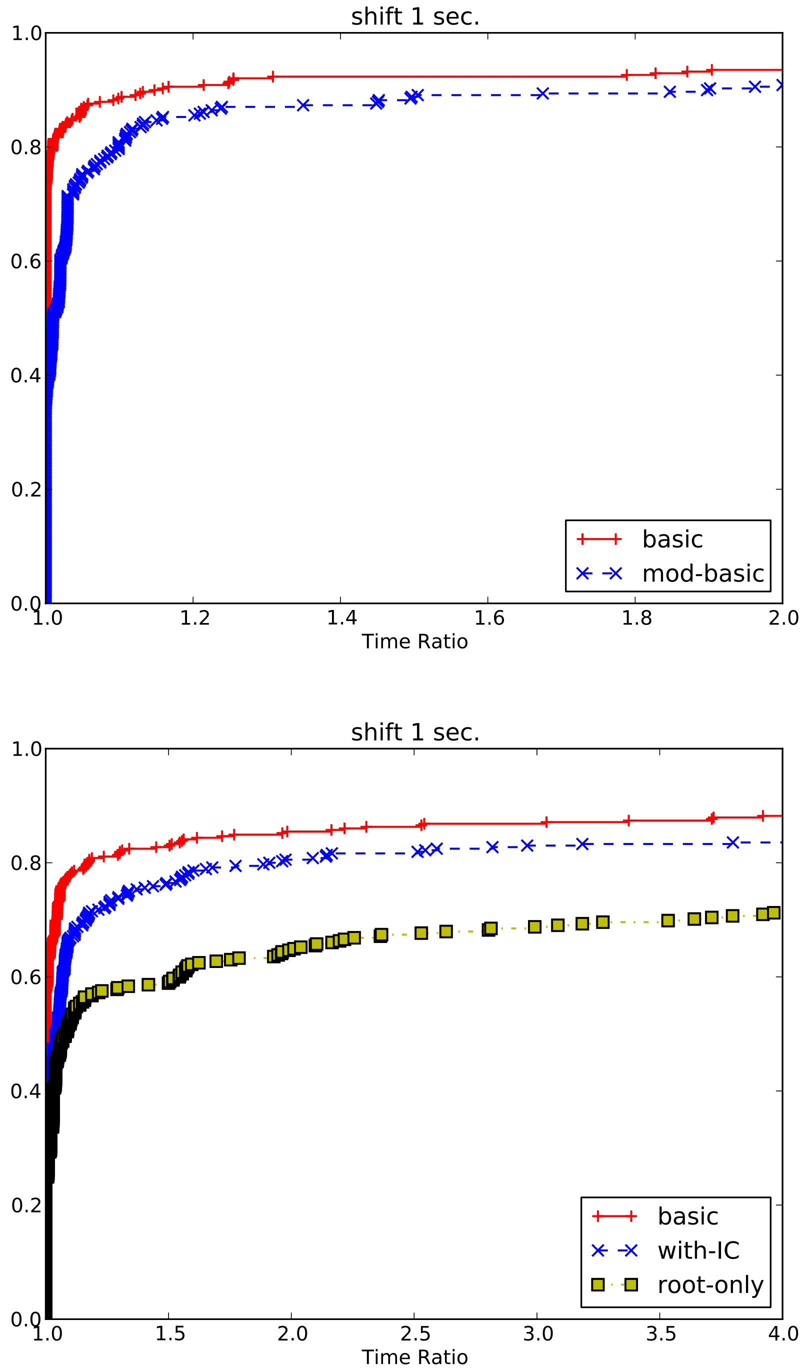

Figure 1: Performance profile comparisons (time shift of 1 sec.): alternative branching rule (top) and fractional cut separation only at the root node (bottom).

at the branching level. To be more specific, when we build each child node, besides updating the lower/upped bound of the branching variable x_{bvar}, we scan all the bilinear inequalities (5) involving x_{bvar} and separate the associated McCormick inequalities (with the new lower or upper bound on x_{bvar}) with respect to the current branching solution x^*: all the violated McCormick inequalities that we find are then added as local constraints

for the child node. In this way, we are sure that also the *fractional* optimal solutions of the child subproblem are affected by the improved McCormick inequalities (while in the original version this holds true only for the integer solutions). This new version of the code was however not significantly better than the original one, so in the end we preferred to stay with the simplest version without the above local constraints.

We also tried a variant of `with-IC`, called `root-only`, where all cuts—including ICs—are separated also for fractional solutions (i.e., in both lazyconstraint and usercut callbacks) at the root node; after the root node, ICs are deactivated and McCormick/gradient cuts are only separated for integer solutions, as in `basic`. Figure 1 (bottom) plots the corresponding performance profile on the MINLPlib instances. The cases where all three versions hit the time limit have been removed; on the remaining 365 ones, both `basic` and `with-IC` reached the time limit 36 times, while `root-only` hit it 64 times. Although implementing a more clever tailing-off strategy could perhaps improve those figures, we decided to stay with the safer option of avoiding the separation of fractional solutions—even at the root node.

5.2 Final experiments

Table 1 reports the outcome of our final experiments on the 620 instances of our testbed. For each algorithm we give:

- the number of instances solved to proven optimality within the given time limit (#opt);
- the number of instances for which the algorithm proved optimality within the smallest computing time, including ties (column "#fast"); instances for which all algorithms hit the time limit are not counted in this figure;
- the arithmetic and the geometric means of the computing time (T_{ari}, and T_{geo}, respectively); computing times are reported in CPU seconds, and geometric means are shifted by 1 second; the reported figures include the instances for which all solvers hit the time limit.

SCIP				basic				with-IC			
#opt	#fast	T_{ari}	T_{geo}	#opt	#fast	T_{ari}	T_{geo}	#opt	#fast	T_{ari}	T_{geo}
378	224	1480.58	34.45	328	156	1744.56	53.04	323	58	1787.33	67.93

Table 1: Results on 620 instances from the MINLPlib.

The performance of our branch-and-cut algorithms turns out to be not too far from that of a highly-sophisticated solver such as `SCIP`. Note however that SCIP and CPLEX are very different in terms of technology. Furthermore, our implementations use callbacks, hence they are completely different from the SCIP's one. Thus, it is not possible to assess a direct comparison of our algorithms with `SCIP`—as a matter of fact, our use of SCIP was only intended to show that our basic callback-based implementation, `basic`, is not far from the state of the art.

Figure 2 gives additional information about the relative performance of `SCIP`, `basic` and `with-IC`. The figure plots the corresponding performance profile [17] on the 408 MINLPlib instances that could be solved by al least one of the three methods in the 1-hour time limit (i.e., we only removed the instances for which *all* methods hit the time limit). It confirms that `basic` and `SCIP` have a quite close performance, while `with-IC` is slightly worse.

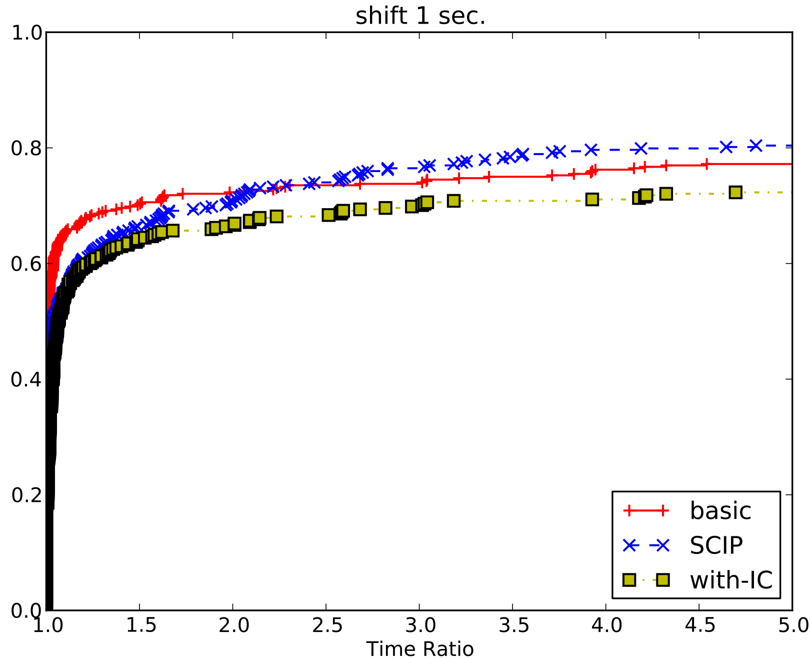

Figure 2: Performance profile comparison of `basic`, `SCIP` and `with-IC`, on the 408 MINLPlib instances that could be solved by at least one method in the 1-hour time limit (time shift of 1 sec.)

To gain additional computational insights, we considered the subset of instances for which all algorithms converged, without numerical issues, to a proven optimal solution within the time limit—although this introduces a bias in disfavor of `SCIP`, which solved more problems within the time limit. This subset contains 248 instances of various difficulty, that we partitioned into 5 non-overlapping classes $C_1, \dots, C_5$ defined as follows. Given an instance i and an algorithm a, let t_{ai} be the computing time of algorithm a on instance i. For each instance i we define $\bar{t}_i := \max_a \{t_{ai}\}$, and assign instance i to the class $k(i)$ such that

$$k(i) = 1 \text{ if } \bar{t}_i \leq 1, \text{ and } \quad 10^{k(i)-2} \leq \bar{t}_i < 10^{k(i)-1} \text{ otherwise.}$$

In other words, C_1 contains the easiest instances that can be solved by all the algorithms within 1 second; C_2 contains all instances not in C_1 that can be solved by all the algorithms within 10 seconds, and so on. This is a fair policy for classifying instances because the competing methods play an indistinguishable role in the class definition, so no bias is expected.

			SCIP			basic			with-IC		
Class	Time Range	#inst	#fast	T_{ari}	T_{geo}	#fast	T_{ari}	T_{geo}	#fast	T_{ari}	T_{geo}
C_1	(0,1]	130	86	0.07	0.05	46	0.06	0.05	12	0.09	0.08
C_2	(1,10]	40	11	2.58	1.22	24	1.23	0.70	6	1.74	1.10
C_3	(10,100]	37	6	26.95	12.34	23	12.30	2.84	10	21.57	9.07
C_4	(100,1000]	29	15	179.26	18.38	10	309.03	97.06	4	296.96	115.79
C_5	(1000, 3600]	12	5	1153.15	154.94	5	789.24	66.83	2	954.65	84.14
ALL	(0,3600]	248	123	81.23	0.90	108	76.39	0.77	34	84.46	1.15

Table 2: Results on the 248 MINLPlib instances than can be solved by all methods within the 1-hour time limit.

Table 2 reports the performance of the three algorithms on the 5 classes defined above and, in the last line, on the entire benchmark. For each class we report the number of instances in the class and, for each algorithm, the average computing time and the number of instances for which the algorithm was the fastest one.

Table 2 shows that `basic` outperforms `SCIP` for all classes but C_1 and C_4. The better performance of `SCIP` on the very easy instances of class C_1 can be explained by observing `SCIP` includes many bilinear-specific tools—including specific preprocessing and the use of a nonlinear solver—that evidently are instrumental to quickly solve (sometimes during presolving) these very easy instances.

Table 3 gives the same information as in Table 2, but with a different grouping of the instances intended to study the behavior of the three algorithms on subsets of harder-and-harder instances. To this end, the testbed is partitioned into 5 classes $G_1, \ldots, G_5$, where $G_k = C_k \cup \cdots \cup C_5$ $(k = 1, \ldots, 5)$. In other words, G_1 contains all the instances, G_2 contains all instances but the easiest ones (i.e., those that can be solved by all algorithms in at most 1 second), and so on.

			SCIP			basic			with-IC		
Class	Time Range	#inst	#fast	T_{ari}	T_{geo}	#fast	T_{ari}	T_{geo}	#fast	T_{ari}	T_{geo}
G_1	(0,3600]	248	123	81.23	0.90	108	76.39	0.77	34	84.46	1.15
G_2	(1,3600]	118	37	170.65	8.19	62	160.48	6.05	22	177.42	10.67
G_3	(10,3600]	78	26	256.84	21.14	38	242.15	17.36	16	267.51	33.02
G_4	(100,3600]	41	20	464.30	34.33	15	449.58	87.02	6	489.46	105.46
G_5	(1000,3600]	12	5	1153.15	154.94	5	789.24	66.83	2	954.65	84.14

Table 3: Results on the 248 MINLPlib instances than can be solved by all methods within the 1-hour time limit.

Results in Table 3 confirm that `basic` is the best algorithm if the very easy instances in class C_1 (which can be solved by all algorithms within just 1 second) are removed. Indeed, for class G_2, algorithm `basic` turns out to be the fastest method in 62 (os opposed to 37 for `SCIP`) cases, and has a better average computing time.

As to algorithm `with-IC`, it seldom produces significant improvements over `basic` in terms of average computing time. This is not really surprising, in that it is a common (somehow frustrating) experience in mixed-integer programming that sophisticated classes of cuts such as ICs and disjunctive cuts are not beneficial when a large and varied testbed is considered. In addition, as observed in the implementation section, the performance of `with-IC` (but not of `basic`) is affected in a negative way by the need of deactivating preprocessing completely. However, there are a number of problems for which ICs do produce a considerable speedup. Table 4 reports the computing times for the three algorithms on a selected subset of nine instances for which `with-IC` is by far the best algorithm, i.e., the addition of intersection cuts in the branch-and-cut algorithm plays a relevant role in the possibility to solve these instances to optimality—or in reducing computing time. This behavior was confirmed (for those instances) by re-running our codes with 5 different random seeds, so as to take performance variability into account.

Instance	Application	`SCIP`	`basic`	`with-IC`
blend531	Multiperiod Blend Scheduling	234.21	3600.00	31.05
crudeoil_lee4_09	Crude Oil Scheduling	89.12	9.83	2.21
portfol_classical050_1	Portfolio Optimization	57.03	54.37	33.26
powerflow0009r	Electricity Network	3600.00	3600.00	969.12
powerflow0014r	Electricity Network	3600.00	3600.00	302.77
sporttournament14	Sports Tournament	3600.00	182.41	125.50
squfl015-080	Facility Location	3600.00	238.53	137.32
squfl025-030	Facility Location	3600.00	44.46	18.72
turkey	Agriculture	61.19	3600.00	0.11

Table 4: Selected instances for which adding intersection cuts is highly beneficial.

When analyzing the results reported in Figure 2, we realized that many of the 408 MINLPlib instances therein considered are in fact quite small: 144 (respectively, 188) of them involve no more than 50 (respectively, 100) variables even after reformulation. It turns out that some of them—including some toy instances named `ex*`—are just trivial for `SCIP` (that leverages on an internal NLP solver), while they are quite hard for our LP-based branch-and-cut solvers. On the other hand, removing *a posteriori* those small instances from our testbed would have been unfair to `SCIP` as it would have produced a biased dataset, so we decided to include all of them in our statistics. Nevertheless, it can be of some interest to see a comparison of the three solvers on the instances that, after reformulation, involve more than 50 (resp. 100) variables, as shown in the top (resp., bottom) plot in Figure 3.

All the reported results so far refer to sequential runs but, as already observed, our implementation can easily handle multi-thread runs—the only requirement being that the callbacks are implemented in a tread-safe mode. In Figure 4 we report a performance profile comparing the running times of `basic` when run sequentially (i.e., with 1 thread, as in the previous experiments) and in parallel mode (opportunistic with 4 threads). Instances where both versions hit the time limit have been eliminated; on the remaining 349 instances, the sequential version hits the time limit 19 times, while the 4-thread version only 2 times. According to the plot, a significant speedup can be achieved when

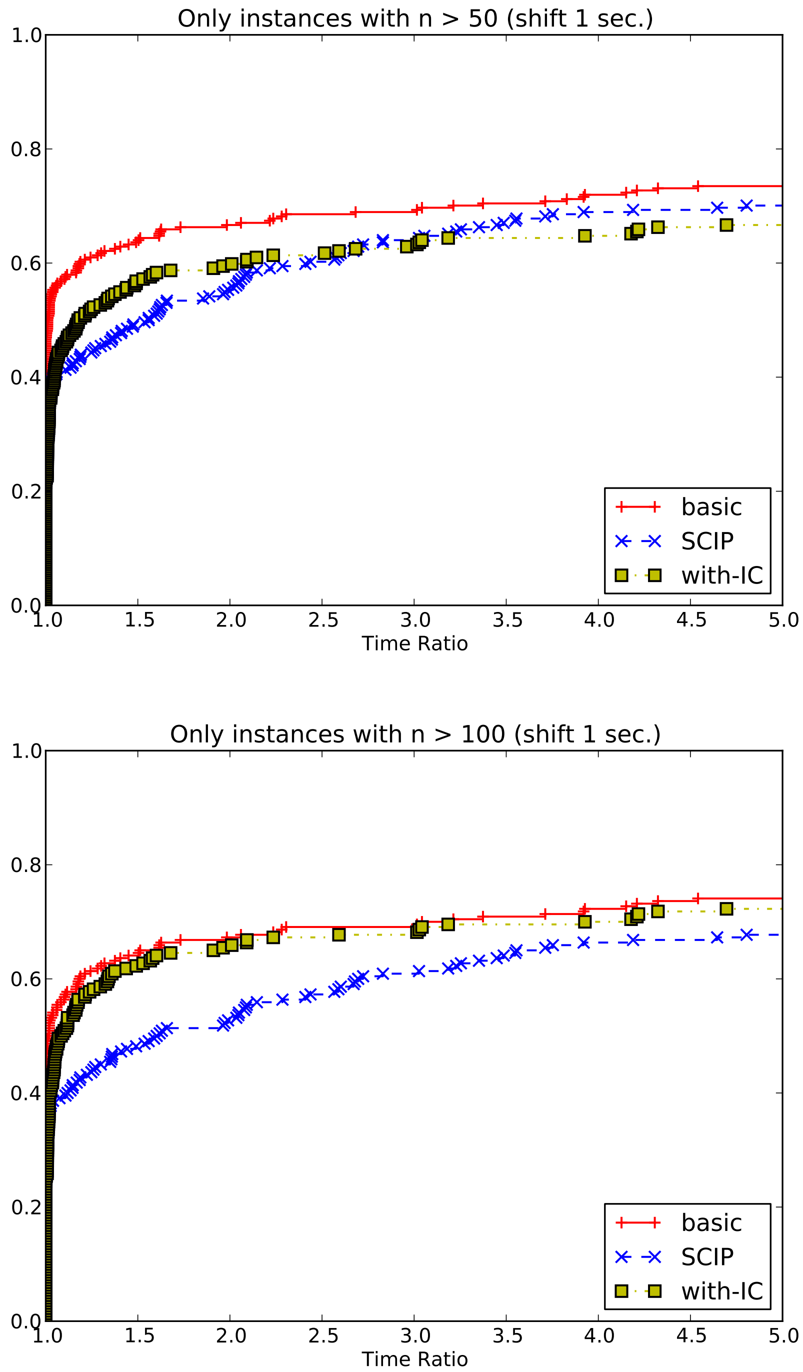

Figure 3: Performance profile comparison of `basic`, `SCIP` and `with-IC` as in Figure 2, when small instances are removed (time shift of 1 sec.)

exploiting parallelism. In particular, in about 20% of the cases we had a speedup of 3 or more—speedups larger than 4 (about 10% of the cases) are mainly due to performance variability [28, 20].

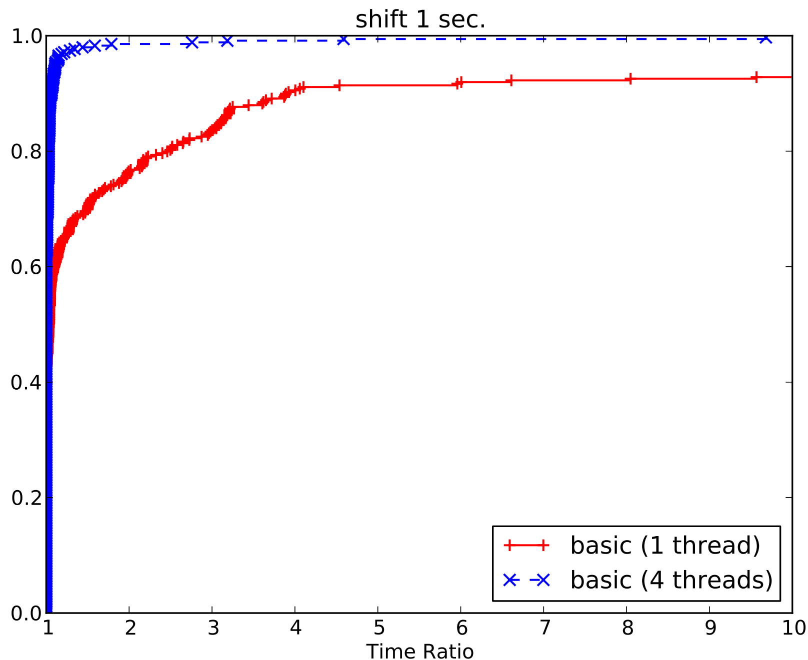

Figure 4: Performance profile comparison of `basic` in sequential vs parallel (4-thread) mode, with a time shift of 1 sec.

6 Conclusions

In this paper we have considered Mixed-Integer Bilinear Programming (MIBLP) problems, and proposed branch-and-cut algorithms for their exact solution. We have introduced a new branching rule that is specifically designed for bilinear problems, and a family of intersection cuts that can strengthen the formulation and possibly reduce the computational effort required for solving the problem. We have computationally tested our algorithms on a large set of instances from the MINLPlib problem library.

Our results show that the proposed algorithms are competitive with a general-purpose state-of-the-art nonlinear solver (SCIP) for a large subset of instances, and that a considerable speedup can be obtained for some specific problems because of the addition of intersection cuts.

Future research should be be devoted to the definition of alternative bilinear-free sets, that are required for deriving the intersection cuts. Understanding the relevant features of a MIBLP instance that influence the performance of our algorithm is also an interesting research topic, that could lead to an automated tuning of the main parameters of our branch-and-cut algorithm—deciding, in particular, if IC separation must be activated or not for the specific instance at hand.

Acknowledgments

This research was funded by MiUR, Italy (PRIN2015 project "Nonlinear and Combinatorial Aspects of Complex Networks") and by the Vienna Science and Technology Fund

(WWTF) through project ICT15-014. We thank the organizers of Dagstuhl Seminar 18081 “Designing and Implementing Algorithms for Mixed-Integer Nonlinear Optimization” (Pierre Bonami, Ambros M. Gleixner, Jeff Linderoth, and Ruth Misener), held in Dagstuhl (Germany) on February 18–23, 2018, for inspiring discussions during the workshop. Thanks are also due to Domenico Salvagnin for his help in using the SCIP code, and to three anonymous referees for their constructive comments.

References

[1] W.P. Adams and H.D. Sherali. Mixed-integer bilinear programming problems. *Mathematical Programming*, 59:279–305, 1993.

[2] F.A. Al-Khayyal and H.D. Sherali. On finitely terminating branch-and-bound algorithms for some global optimization problems. *SIAM Journal on Optimization*, 10:1049–1057, 2000.

[3] E.D. Andersen and K.D. Andersen. The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm. In H. Frenk *et al.*, editor, *High Performance Optimization*, pages 197–232. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[4] C. Audet, J. Brimberg, P. Hansen, S. Le Digabel, and N. Mladenović. Pooling problem: Alternate formulations and solution methods. *Management Science*, 50(6):761–776, 2004.

[5] C. Audet, P. Hansen, B. Jaumard, and G. Savard. A branch and cut algorithm for nonconvex quadratically constrained quadratic programming. *Mathematical Programming*, 87(1):131–152, 2000.

[6] E. Balas. Intersection cuts–a new type of cutting planes for integer programming. *Operations Research*, 19(1):19–39, 1971.

[7] P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4-5):597–634, 2009.

[8] D. Bienstock, C. Chen, and G. Muñoz. Outer-Product-Free Sets for Polynomial Optimization and Oracle-Based Cuts. *arXiv e-prints*, page arXiv:1610.04604, Oct 2016.

[9] Daniel Bienstock. Computational study of a family of mixed-integer quadratic programming problems. In Egon Balas and Jens Clausen, editors, *Integer Programming and Combinatorial Optimization*, pages 80–94, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[10] J. M. Bloemhof-Ruwaard and E.M.T. Hendrix. Generalized bilinear programming: An application in farm management. *European Journal of Operational Research*, 90(1):102 – 114, 1996.

[11] P. Bonami and J. Lee. BONMIN user's manual. Technical report, IBM Corporation, 2007.

[12] R.H. Byrd, J. Nocedal, and R.A. Waltz. *Knitro: An Integrated Package for Nonlinear Optimization*, pages 35–59. Springer US, Boston, MA, 2006.

[13] A. Caprara, M. Locatelli, and M. Monaci. Bidimensional packing by bilinear programming. *Lecture Notes in Computer Science*, 3509:377–391, 2005. Proceedings of IPCO 2005.

[14] M. Conforti, G. Cornuéjols, and G. Zambelli. Corner polyhedron and intersection cuts. *Surveys in Operations Research and Management Science*, 16(2):105–120, 2011.

[15] C. D'Ambrosio, J. Linderoth, and J. Luedtke. Valid inequalities for the pooling problem with binary variables. *Lecture Notes in Computer Science*, 6655:117–129, 2011. Proceedings of IPCO 2011.

[16] S.S. Dey and A. Gupte. Analysis of milp techniques for the pooling problem. *Operations Research*, 63(2):412–427, 2015.

[17] E. D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[18] M. Fischetti, I. Ljubić, M. Monaci, and M. Sinnl. A new general-purpose algorithm for mixed-integer bilevel linear programs. *Operations Research*, 65(6):1615–1637, 2017.

[19] M. Fischetti and M. Monaci. Branching on nonchimerical fractionalities. *Operations Research Letters*, 40:159–164, 2012.

[20] M. Fischetti and M. Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014.

[21] L.R. Foulds, D. Haugland, and K. Jörnsten. A bilinear approach to the pooling problem. *Optimization*, 24(1-2):165–180, 1992.

[22] F. Glover. Polyhedral convexity cuts and negative edge extensions. *Zeitschrift für Operations Research*, 18(5):181–186, 1974.

[23] F. Glover and D. Klingman. Improved convexity cuts for lattice point problems. *Journal of Optimization Theory and Applications*, 19(2):283–291, 1976.

[24] A. Gupte, S. Ahmed, M.S. Cheon, and S.S. Dey. Solving mixed integer bilinear problems using MILP formulations. *SIAM Journal on Optimization*, 23(2):721–744, 2013.

[25] A. Gupte, S. Ahmed, S.S. Dey, and M.S. Cheon. Relaxations and discretizations for the pooling problem. *Journal of Global Optimization*, 67(3):631–669, 2017.

[26] R. Karuppiah and I.E. Grossmann. Global optimization for the synthesis of integrated water systems in chemical processes. *Computers and Chemical Engineering*, 30:650–673, 2006.

[27] F.K. Karzan, G.L. Nemhauser, and M.W.P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1:249–293, 2009.

[28] A. Lodi and A. Tramontani. *Performance Variability in Mixed-Integer Programming*, chapter Chapter 1, pages 1–12.

[29] A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. In *Theory Driven by Influential Applications*, chapter 2, pages 1–12. 2014.

[30] G.P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I - convex underestimating problems. *Mathematical Programming*, 10(1):146–175, 1976.

[31] G.P. McCormick. *Nonlinear Programming: Theory, Algorithms and Applications.* John Wiley & Sons, 1983.

[32] R. Misener and C.A. Floudas. Advances for the pooling problem: modeling, global optimization, and computational studies. *Applied and Computational Mathematics*, 8(1):3–22, 2009.

[33] R. Misener and C.A. Floudas. Antigone: Algorithms for continuous / integer global optimization of nonlinear equations. *J. Global Optimization*, 59(2-3):503–526, 2014.

[34] A.G. Nahapetyan. *Bilinear programming: applications in the supply chain management*, pages 282–288. Springer US, Boston, MA, 2009.

[35] Nikolaos V. Sahinidis. BARON: A general purpose global optimization software package. *Journal of Global Optimization*, 8(2):201–205, Mar 1996.

[36] H. D. Sherali and A. Alameddine. A new reformulation-linearization technique for bilinear programming problems. *Journal of Global Optimization*, 2(4):379–410, 1992.

[37] M. Tawarmalani, J.-P. Richard, and K. Chung. Strong valid inequalities for orthogonal disjunctions and bilinear covering sets. *Mathematical Programming*, 124(1):481–512, 2010.

[38] S. Vigerske and A. Gleixner. SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework. *Optimization Methods and Software*, 33(3):563–593, 2018.

[39] A. Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, Mar 2006.