Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TIMO BERTHOLD AND AMBROS M. GLEIXNER

Undercover Branching

This paper is to appear in the Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013) held June 5–7, 2013, in Rome, Italy.

ZIB-Report 13-14 (April 2013)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Undercover Branching*

Timo Berthold and Ambros M. Gleixner†

April 2013

Abstract

In this paper, we present a new branching strategy for nonconvex MINLP that aims at driving the created subproblems towards linearity. It exploits the structure of a *minimum cover* of an MINLP, a smallest set of variables that, when fixed, render the remaining system linear: whenever possible, branching candidates in the cover are preferred.

Unlike most branching strategies for MINLP, Undercover branching is not an extension of an existing MIP branching rule. It explicitly regards the nonlinearity of the problem while branching on integer variables with a fractional relaxation solution. Undercover branching can be naturally combined with any variable-based branching rule.

We present computational results on a test set of general MINLPs from MINLPLib, using the new strategy in combination with reliability branching and pseudocost branching. The computational cost of Undercover branching itself proves negligible. While it turns out that it can influence the variable selection only on a smaller set of instances, for those that are affected, significant improvements in performance are achieved.

1 Introduction

State-of-the-art solvers for generic mixed integer linear programs (MIPs) and mixed integer nonlinear programs (MINLPs) are based on the branch-and-bound paradigm [23]. The question of how to split a given MIP or MINLP into subproblems, commonly referred to as the *branching* step, lies at the heart of any branch-and-bound algorithm. Its main purpose is to improve the *dual bound* by, e.g., eliminating fractionality of the integer variables and, for MINLP, reducing the convexification gap between the nonconvex constraint functions and the relaxation. In MIP solving, typically an LP relaxation is solved for the *bounding* step. For MINLP, although an NLP relaxation is a natural choice, most state-of-the-art solvers also rely on an LP relaxation.

The branching rule is one of the components with highest impact on the overall performance of MIP solvers [12, 2]. Consequently, the literature has seen many publications on efficient branching rules, which will be reviewed in the next

*This paper is to appear in the Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013) held June 5–7, 2013, in Rome, Italy.

†Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, berthold,gleixner@zib.de.
The authors gratefully acknowledge the support of the DFG Research Center Matheon *Mathematics for key technologies* in Berlin and the Berlin Mathematical School.

paragraphs. For MINLP, up to now, research has mainly focused on adopting MIP branching rules [29, 7].

In mixed integer programming, the most common methodology is variable-based branching (an exception being [21]), i.e., considering integer variables with a fractional LP solution value as *branching candidates*. State-of-the-art branching rules, sometimes also called variable selection heuristics, estimate the impact that splitting a variable's domain has on the dual bound and the solvability of the created subproblems. A very prominent approach is the usage of so-called *pseudocosts* [8], an estimate of the increase that branching on a variable has on the optimum of the LP relaxation.

In [25], it is shown that initializing pseudocosts by strong branching [5, 6] is beneficial, an approach further refined in reliability branching [4]. Hybrid branching [3] combines reliability branching with VSIDS [26] and inference values [24], two common branching dichotomies in satisfiability testing and constraint programming, respectively. Methods that combine pseudocost and strong branching information can be considered to be the state-of-the-art for MIP solvers.

In recent years several publications have investigated new paradigms for variable-based branching schemes that show superior performance on important classes of hard MIPs. Kılınc et. al. [22] suggest to use conflict learning information for branching on 0-1 integer programs. To this end, they run a sampling phase of 500 branch-and-bound nodes during which they collect conflict constraints, restart the solution process, and prefer branching on variables that appear in short conflict constraints during the second phase.

Backdoor branching [15] goes one step further: it applies multiple restarts, attempting to find a good approximation of a *backdoor*. Here, a backdoor is a (preferably small) set of variables such that, whenever these variables get assigned integer values, solving an LP on the remaining variables gives a proof of feasibility or infeasibility. After each restart, the approximated backdoor is computed by solving a set covering problem. Branching is exclusively performed on backdoor variables until all of them are fixed. *Non-chimerical branching* [16] is a criterion to rule out candidates for strong branching which are not promising.

For nonconvex MINLP, it is possible that the LP relaxation is integral and cannot be strengthened further by gradient cuts (see Footnote 3), while some of the nonconvex constraints are still violated. In this case, spatial branching can be applied, i.e., branching on variables contained in violated nonconvex constraints, including continuous variables. Subsequently, the relaxation can be tightened in the created subproblems; thereby, the infeasible relaxation solution is cut off.

To select a branching variable for spatial branching, Tawarmalani and Sahinidis [28] suggest performing a so-called *violation transfer*. This estimates the impact of each variable on the problem by minimizing and maximizing a Lagrangian function over a neighborhood of the current relaxation solution when holding all other variables fixed. For a linear relaxation, this is similar to selecting variables with large reduced cost.

In [7], the concept of pseudocosts has been extended to continuous variables by investigating suitable counterparts for the violation of integrality, which is used in pseudocost formulas for MIP. Their computational analysis suggests

that pseudocost-based branching is superior for hard MINLPs, while for easy instances and nonconvex NLPs it is outperformed by violation transfer or even simpler violation-based rules.

In this paper, we suggest a branching strategy that aims at driving the subproblems towards linearity. To this end, *Undercover branching* restricts the set of branching candidates to a *minimum cover* [9] of an MINLP, i.e., a smallest set of variables that, when fixed, linearizes all constraints. It builds on the ideas of the Undercover heuristic [9, 10], which computes feasible solutions for MINLPs by solving a sub-MIP defined via a minimum cover.

Whereas many branching rules are history-based and share heuristic components, Undercover branching exploits structural information of the problem in an exact manner. In the spirit of backdoor branching, it features a pre-selection rule for branching candidates: independent of the current subproblem, Undercover branching globally separates a set of variables with a certain predicate from others. Consequently, it can be combined with any variable-based branching rule.

A major characteristic of Undercover branching is that it respects information on the nonlinearity of the problem already in the branching decisions for fractional integer variables, not only during spatial branching. From a computational point of view, Undercover branching has the benefit that it costs little additional time. In the way that we suggest, a minimum cover has to be computed only once in the beginning of the solution process. Our experiments show this to be computationally cheap in practice. In contrast to backdoor branching, Undercover branching does not require repeated restarts of the main solution procedure.

The remainder of the article is organized as follows. Section 2 states a formal definition of a minimum cover, explains how it can be computed, and analyzes minimum cover sizes of the test problems in MINLPLib [13]. In Section 3, we present the general idea and implementational details of the newly proposed branching strategy. In Section 4, we evaluate the applicability and the impact of Undercover branching on instances from MINLPLib. Finally, we discuss the results and give an outlook on future work in Section 5.

2 Covers of mixed integer nonlinear programs

The branching strategy investigated in this paper relies on the concept of a minimum cover, a structural feature of an MINLP that is a measure for its "grade of nonlinearity". This notion has been introduced in [9] and utilized for the design of a primal heuristic. In the following, we give a brief summary of the main results from [9, 10].

Definition 1 (cover of an MINLP)**.** *Let P be an MINLP of form*

$$\begin{array}{lll} \min & c^{\mathsf{T}}x & \\ \text{s.t.} & g_k(x) \leqslant 0 & \text{for } k = 1, \ldots, m, \\ & \ell_i \leqslant x_i \leqslant u_i & \text{for } i = 1, \ldots, n, \\ & x_i \in \mathbb{Z} & \text{for } i \in \mathcal{I}, \end{array} \tag{1}$$

where $c \in \mathbb{R}^n$, $g_k : \mathbb{R}^n \to \mathbb{R}$, $\ell_i \in \mathbb{R} \cup \{-\infty\}$, $u_i \in \mathbb{R} \cup \{+\infty\}$, $\ell_i < u_i$, *and* $\mathcal{I} \subseteq \{1, \ldots, n\}$.[1] *We call a set of variable indices* $\mathcal{C} \subseteq \{1, \ldots, n\}$ *a* cover of the function g_k *if and only if for all* $x^* \in [\ell, u]$ *the set*

$$\{(x, g_k(x)) : x \in [\ell, u], x_i = x_i^* \text{ for all } i \in \mathcal{C}\} \tag{2}$$

is an affine set intersected with $[\ell, u] \times \mathbb{R}$. *We call* $\mathcal{C}$ *a* cover of P *if and only if* $\mathcal{C}$ *is a cover of all constraint functions* $g_1, \ldots, g_m$.

Trivial examples of covers are the set of all variables or the set of all variables appearing in nonlinear terms. As will be shown at the end of this section, however, many instances of practical interest allow for significantly smaller covers. Minimum covers can be computed generically by solving a vertex covering problem. This is a crucial observation for exploiting them in an MINLP solver.

Definition 2 (co-occurrence graph)**.** *Let* P *be an MINLP of form* (1) *with* $g_1, \ldots, g_m$ *twice continuously differentiable on the interior of* $[\ell, u]$. *We call* $G_P = (V_P, E_P)$ *the* co-occurrence graph *of* P *with node set* $V_P = \{1, \ldots, n\}$ *given by the variable indices of* P *and edge set*

$$E_P = \big\{ij : i, j \in V, \exists k \in \{1, \ldots, m\} : \frac{\partial^2}{\partial x_i \partial x_j} g_k(x) \not\equiv 0\big\},$$

i.e., an edge connects nodes i *and* j *if and only if the Hessian matrix of some constraint has a structurally nonzero entry* (i, j).

This leads to the following result:

Theorem 1. *Let* P *be an MINLP of form* (1) *with* $g_1, \ldots, g_m$ *twice continuously differentiable on the interior of* $[\ell, u]$. *Then* $\mathcal{C} \subseteq \{1, \ldots, n\}$ *is a cover of* P *if and only if it is a vertex cover of the co-occurrence graph* G_P.

Proof. See [10].

Since vertex covering is $\mathcal{NP}$-hard [18] and any graph can be interpreted as the co-occurrence graph of a suitably constructed MINLP, we obtain

Corollary 1. *Computing a minimum cover of an MINLP is* $\mathcal{NP}$*-hard.*

In practice, however, minimum covers can be computed rapidly by solving a binary programming formulation of the vertex covering problem with a state-of-the-art MIP solver as has been argued already in [10]. As an example, we have computed minimum covers for 255 instances[2] from MINLPLib [13] for the present paper. Using the MINLP solver SCIP 3.0 [27] and the expression interpreter CppAD 20120101.3 [14] for obtaining the sparsity patterns of the Hessians, the binary programs were all solved within the root node and took at most 0.2 seconds on the hardware described in Sec. 4.

[1] W.l.o.g. we may assume a linear objective, because for a nonlinear objective $f(x)$, we can always append a constraint $f(x) \leqslant x_0$ and minimize the auxiliary variable x_0.

[2] This excludes 18 instances that cannot be handled by SCIP 3.0, e.g., because they contain trigonometric functions: `blendgap`, `deb{6,7,8,9,10}`, `dosemin{2,3}d`, `prob10`, `var_con{5,10}`, `water{3,ful2,s,sbp,sym1,sym2}`, and `windfac`.

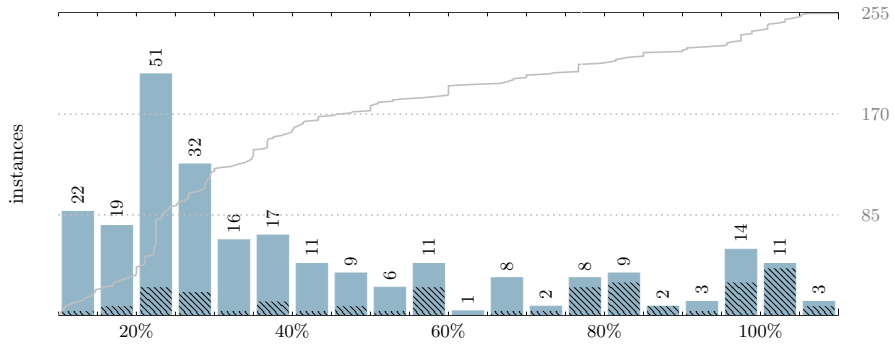

Figure 1: Distribution of the sizes of a minimum cover relative to the total number of variables over 255 instances from MINLPLib. Numbers above the bars state how many instances fall in the corresponding 5% interval. Shaded bars indicate the proportion of minimum covers with integer variables only. The cumulative distribution function refers to the right-hand scale.

The distribution of the sizes of minimum covers is depicted in Fig. 1. One third of the instances allows for covers consisting of less than 14% of the variables and another third of the instances has covers with less than 36% of the variables. As indicated by the shaded bars, 65 instances have a minimum cover with only integer variables. For the vast majority of 163 instances it contains only continuous variables. The minimum covers for the remaining 27 instances are formed by continuous variables complemented by a small fraction of less than 1% integer variables.

To summarize, we observe that the majority of problems from MINLPLib features small covers. Note that this even holds for many instances that have almost all variables contained in nonlinear terms. The latter underlines that the size of a minimum cover valuably complements other measures of nonlinearity such as number of nonlinear nonzeros, constraints, or variables appearing in nonlinear terms.

3 Using MINLP covers for branching

Although MIP and MINLP are both $\mathcal{NP}$-hard, arguably MIPs are computationally easier than MINLPs. For MIP, it is possible to compute a relaxation solution in polynomial time that only drops the integrality requirements, but respects all constraint functions. For MINLP, solving a (nonconvex) NLP relaxation is already $\mathcal{NP}$-hard. Also, generic cutting plane algorithms, which contribute a lot to the practical success of MIP solvers, do not have a direct equivalent in MINLP. Of course, they can be used to strengthen a MIP relaxation, but they do not yield a finite algorithm. Last, but not least, considering today's state-of-the-art in optimization software, MIP codes have reached an impressive maturity and have become a standard industry tool, whereas MINLP software has just recently evolved and only few codes are available by now.

From this point of view it is an important observation that a cover of an

MINLP presents a structure that turns an MINLP into a MIP for any assignment of the variables in the cover. Branching shrinks variables' domains, ideally fixes them, and therefore, branching on cover variables offers itself as a promising strategy to drive an MINLP towards linearity. If a pure branch-and-bound algorithm is applied without domain propagation techniques, the size of the cover corresponds to the minimum number of branching decisions that have to be taken before obtaining a linear subproblem. In particular, the following observation holds:

Lemma 1. *Let P be an MINLP of form (1) and $\mathcal{C} \subseteq \mathcal{I}$ a cover of P with $\ell_i, u_i \in \mathbb{Z}$ for all $i \in \mathcal{C}$. Then, P can be solved by solving a sequence of at most $\prod_{i \in \mathcal{C}}(u_i - \ell_i + 1)$ MIPs.*

In the case of variables with infinite domain, i.e., continuous or unbounded integer variables, branching on cover variables does not necessarily enforce linearity in a bounded number of steps. Nevertheless, branching on such a variable, and thereby tightening its domain, is likely to produce better underestimators than, e.g., branching on an integer variable which is not even part of a nonlinear expression. Better underestimators lead to better relaxation bounds which lead to earlier pruning (or feasibility) of the created subproblems.

In particular, Undercover branching explicitly regards the nonlinearity of the problem also when branching on integer variables with a fractional relaxation solution.

We therefore suggest to use a branching strategy that prefers cover variables over others as depicted in Fig. 2. The methods branch_int and branch_spat in lines 7 and 10, respectively, are black box methods for which any standard variable-based rule for branching on fractional integer variables and for spatial branching can be used.

```
1  input MINLP P as in (1) with cover C
2        set of fractional variables F
3        set of candidates for spatial branching S
4  begin
5  |  if F ≠ ∅ then
6  |  |  if F ∩ C ≠ ∅ then F ← F ∩ C
7  |  |  branch_int (P, F)
8  |  else
9  |  |  if S ∩ C ≠ ∅ then S ← S ∩ C
10 |  |  branch_spat (P, S)
11 end
```

Figure 2: Undercover branching algorithm.

To perform Undercover branching, a cover of the MINLP P has to be computed once before the branch-and-bound process starts. This global structure can be exploited also by other solver components, e.g. an Undercover heuristic as in [9, 10].

As a distinguishing feature, the structure used for branching is computed exactly, at negligible cost (see previous section), and no sampling phase as, e.g., in [22] or [15], is required. Furthermore, we do not enforce branching on cover variables via strict branching priorities: if the candidate set lies completely outside of the cover, we do not continue branching on unfixed cover variables, but stick with the candidates proposed by the solver.

4 Experimental results

In this section we investigate the computational impact of Undercover branching when combined with standard branching rules implemented in the MINLP solver SCIP 3.0 [2, 27].

SCIP implements a branch-and-bound algorithm based on an LP relaxation that is constructed via gradient cuts[3] for convex constraints and linear over- and underestimators of the nonconvex terms. Further algorithmic components comprise primal heuristics, cutting planes applied to the MIP relaxation, an extensive presolving and propagation engine, conflict analysis, and several reformulation steps to detect convex or convexifiable constraints at the beginning. For details, we refer to [29, 11].

By default, SCIP applies binary branching, i.e., it splits the current node into two subproblems. Branching on integer variables is prefered, however, not categorically: if all integer variables have integral value in the LP solution (and nonlinearities are still violated), SCIP continues with spatial branching even if not all of the integer variables are fixed.

For branching on integer variables with a fractional LP value, the default variable selection rule is *hybrid reliability branching* [3], which uses pseudocosts that are initialized by multiple strong branches per variable as in reliability branching [4]. VSIDS [26] and inference values [24, 1], two scores from satisfiability testing and constraint programming, are taken into account for tie breaking. For spatial branching, SCIP implements the pseudocost strategy "rb-int-br"[4] from [7] weighted by the violations of the constraints in which a variable appears.

All experiments were conducted on a cluster of 64bit Intel Xeon X5672 CPUs at 3.2 GHz with 12 MB cache and 48 GB main memory and a time limit of one hour. Hyperthreading and Turboboost were disabled. For the latter experiment, we ran only one job per node to avoid random noise in the measured running time that might be caused by cache-misses if multiple processes share common resources. As subroutines, SCIP was linked to the LP solver CPLEX 12.4 [20], the expression interpreter CppAD 20120101.3 [14], and the NLP solver Ipopt 3.10.2 [30]. To avoid interactions, we deactivated the Undercover heuristic. To measure tree sizes accurately, we deactivated restarts.

As test set we chose the MINLPLib [13, 17] featuring 273 instances. We excluded 18 instances that cannot be parsed or handled by SCIP 3.0, see Footnote 2. 13 instances were linearized during presolving; 42 further instances could

[3] If a convex constraint $g(x) \leqslant 0$, $g \in \mathcal{C}^1([\ell, u], \mathbb{R})$, is violated at some x^*, then x^* can be cut off by the *gradient cut* $\nabla g(x^*)^\mathsf{T}(x - x^*) + g(x^*) \leqslant 0$.

[4] For this strategy, the pseudocosts are multiplied by the distance of the current relaxation solution to the bounds of a variable when computing the branching score.

be solved already during root node processing, hence no branching was applied; for two instances, branching had not started after one hour. We also removed three instances for which SCIP 3.0 suffers from numerical inaccuracies which lead to inconsistent solution values (independent of applying Undercover branching). All in all, this leaves a test set of 195 instances. Note that in MINLP, even more than in MIP, standard test sets often decompose into very easy and extremely difficult instances, with very few medium hard problems. Also, they are typically not as heterogeneous as, e.g., the MIPLIBs.

How often? As an initial experiment, we analyzed how often the Undercover pre-selection can be applied to reduce the candidate set. Note that there are two general cases when Undercover branching does not make a difference. If all candidates lie outside the cover (e.g., when the cover is entirely continuous, but there are integer variables with fractional LP solution), the algorithm will select a non-cover variable; if all candidates lie inside the cover (e.g., when the set of fractional variables is a subset of the cover), our algorithm does not yield any impact at that node of the tree.

To this end, we ran SCIP with its default branching rules, and enforced the pre-selection of cover variables as in Fig. 2 whenever possible. At each branching decision taken, we recorded whether all candidates were inside the cover, all candidates were outside, or whether the intersection was nontrivial.

It turned out that on 23 of the instances Undercover branching actually affected the branching decisions. For a further 26 instances the candidates from $\mathcal{F}$ were always included in the cover; all of these had a cover of at least 40% of the variables. For the majority of instances, no candidates were contained in the minimum cover used.[5]

This result is not overly surprising: it is explained by the fact that we are employing *minimum* covers and consequently increase the likelihood that the branching candidates are all outside the cover. However, since the computational overhead of Undercover branching is negligible, it does not degrade the performance on the unaffected instances. It remains to be analyzed whether it is helpful for the set of instances for which it actually affects branching decisions.

How good? The goal of our second experiment was to compare the performance impact of Undercover branching on the 23 instances that are affected. Before discussing the complete results, consider the small instance ex1264a as an illustrative example, modelling a nonconvex trim-loss problem from [19]. Figure 3 shows the actual search trees explored by SCIP 3.0 default and SCIP with Undercover branching added. As can be seen, both trees have similar structure, but the number of nodes is reduced significantly because subtrees can be pruned earlier. This instance indeed confirms our hope that Undercover branching helps to drive the subproblems towards linearity faster: while without, only three nodes are linear, with Undercover branching eleven nodes become linear. Without Undercover branching, linear nodes only appear in depth twelve and below,

[5] For 6 instances, SCIP did not branch on integer variables at all; for 140 instances, $\mathcal{F} \cap \mathcal{C}$ was always empty. Spatial branching did not get performed for 61 instances, 111 times $\mathcal{S} \cap \mathcal{C}$ was always empty.

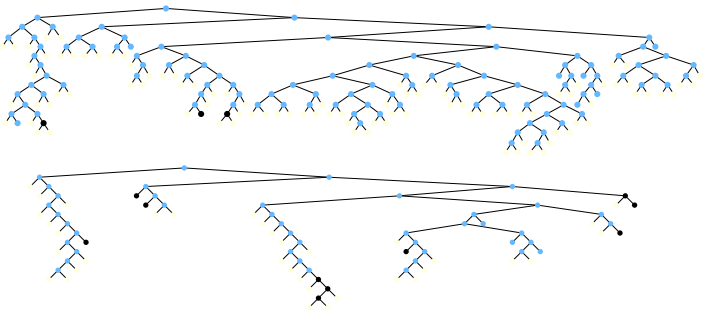

Figure 3: Search trees explored by SCIP 3.0 for instance `ex1264a` with default branching reliability/pseudocost (top, 111 nodes processed) and Undercover branching (bottom, 52 nodes processed). White nodes pruned unprocessed. Linear nodes marked black.

whereas with Undercover branching, linear nodes can be observed from depth four onwards.

Our main hope is to reduce the number of branch-and-bound nodes processed by exploiting the global perspective provided by a cover in addition to the local perspective of a branching rule at a specific node. However, note that if the base rule employed in the branch_int procedure applies strong branching on its candidates such as the hybrid reliability rule in SCIP, then the restriction of the candidate set affects the solving process beyond the branching variable selection. On the one hand, computation time for solving strong branching LPs on the excluded candidates is saved; on the other hand, variable fixings that can be learned from strong branching might be lost.

Hence, we evaluated the impact of Undercover branching w.r.t. two base rules: SCIP's default reliability/pseudocost as described above, and pseudocost/pseudocost, i.e., exchanging the hybrid reliability rule for branching on integer variables by a pure pseudocost rule without any strong branching. The results for those instances that could be solved by at least one variant can be seen in Tab. 1 and Tab. 2, respectively. Using Undercover branching, for both cases two more instances could be solved as compared to not using it.

On four instances in Tab. 1 solved by both variants, Undercover branching increases the number of branch-and-bound nodes. This might be due to the side effect on strong branching described above. For the majority of cases, however, it reduces the number of branch-and-bound nodes significantly. The impact is even more visible on the number of performed strong branches, which is decreased ten times and only increased once. Since the main goal of Undercover branching is to restrict the set of branching candidates, this could be expected. Finally, considering computation time, Undercover branching helps nine times, four times the performance deteriorates. Regarding the shifted geometric means,[6] Undercover branching reduces the computation time by 79% and the number of branch-and-bound nodes by 45%. Note that also when excluding the two positive outliers `tln4` and `tln5` and one negative outlier `tltr`, Undercover branching still yields an overall reduction of all considered performance measures.

[6]We used a shift of 100 nodes, 100 strong branchings, and 10s to compute the means.

Table 1: Impact of Undercover branching on number of nodes, time, and strong branches performed for affected instances solved to optimality within one hour.

	reliability/pseudocost			with undercover			relative [%]		
instance	nodes	strbrs	time [s]	nodes	strbrs	time [s]	nodes	strbrs	time
ex1263a	273	292	0.56	132	135	0.17	−52	−54	−70
ex1264a	111	218	0.33	52	120	0.08	−53	−45	−76
ex1265a	135	171	0.18	75	109	0.11	−44	−36	−39
ex1266a	59	220	0.38	101	177	0.35	+71	−20	−8
fac1	5	6	0.04	5	2	0.06	0	−67	+50
fac3	15	54	0.30	9	5	0.24	−40	−91	−20
nvs15	4	5	0.02	4	2	0.02	0	−60	0
pump	509	110	2.96	773	113	3.57	+52	+3	+21
st_e36	206	0	0.79	200	0	0.77	−3	0	−3
st_e40	19	22	0.08	25	22	0.10	+32	0	+25
tln4	2518	291	1.99	171	41	0.55	−93	−86	−72
tln5	500254	2621	373.73	7977	463	5.89	−98	−82	−98
tloss	78	227	0.19	77	166	0.12	−1	−27	−37
tltr	4	57	0.18	17	88	0.20	+325	+54	+11
shifted mean	291	144	3.68	141	79	0.77	−52	−45	−79
timed out:									
tln6	4101772	4007	*58% gap*	17005	471	22.28	−99	−88	·
tln7	2609508	3946	*124% gap*	9703667	5296	3156.38	+272	+34	·

Table 2: Impact of Undercover branching on number of nodes and time for affected instances solved to optimality within one hour.

	pseudocost/pseudocost			with undercover			relative [%]		
instance	nodes	strbrs	time [s]	nodes	strbrs	time [s]	nodes	strbrs	time
ex1263a	166	·	0.20	163	·	0.18	−2	·	−10
ex1264a	38	·	0.20	85	·	0.18	+124	·	−10
ex1265a	138	·	0.21	104	·	0.40	−25	·	+90
ex1266a	70	·	0.08	141	·	0.20	+101	·	+150
fac1	7	·	0.05	5	·	0.05	−29	·	0
fac3	23	·	0.52	9	·	0.23	−61	·	−56
nvs15	4	·	0.02	4	·	0.03	0	·	+50
pump	1007	·	4.38	827	·	3.62	−18	·	−17
st_e36	206	·	0.79	200	·	0.73	−3	·	−8
st_e40	23	·	0.07	25	·	0.10	+9	·	+43
tln4	1454	·	1.49	202	·	0.67	−86	·	−55
tln5	622318	·	446.86	11013	·	8.39	−98	·	−98
tloss	73	·	0.09	77	·	0.11	+5	·	+22
tltr	18	·	0.17	4	·	0.16	−78	·	−6
shifted mean	287	·	3.85	159	·	0.91	−45	·	−76
timed out:									
tln6	4963908	·	*79% gap*	16382	·	21.54	−99	·	·
tln7	3207274	·	*226% gap*	468231	·	593.74	−85	·	·

For pure pseudocost branching, see Tab. 2, we observe a similar behavior: Undercover branching shows an improvement in the performance measures significantly more often than a deterioration, in shifted geometric mean we see improvements of 76% w.r.t. computation time and 45% w.r.t. the number of branch-and-bound nodes.

5 Conclusion and outlook

In this paper, we have introduced Undercover branching, a new branching strategy for MINLP that exploits minimum covers to drive the subproblems created faster towards linearity. We showed that a combination of Undercover branching with either hybrid reliability branching or pseudocost branching outperforms the corresponding branching rules without Undercover information, yielding savings of 45% w.r.t. the number of branch-and-bound nodes and more than 70% w.r.t. running time in geometric mean for affected instances. The time consumed by the branching rule itself proved negligible.

Currently, the main limitation of Undercover branching is that the fraction of affected instances is relatively small (23 out of 195 instances). Therefore, the main goals of our future research are to employ alternative (not minimum) covers and to investigate the trade-off between cover size and number of affected instances. Furthermore, we work on identifying linear subproblems with large sub-trees, which could then be solved more efficiently using a pure MIP solver.

References

[1] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007.

[2] Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, TU Berlin, 2007.

[3] Tobias Achterberg and Timo Berthold. Hybrid branching. In Willem Jan van Hoeve and John N. Hooker, editors, *Proc. of the 6th CPAIOR*, volume 5547 of *LNCS*, pages 309–311. Springer, 2009.

[4] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[5] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. Finding cuts in the TSP (A preliminary report). Technical Report 95-05, DIMACS, 1995.

[6] David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, USA, 2007.

[7] Pietro Belotti, Jon Lee, Leo Liberti, François Margot, and Andreas Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods & Software*, 24:597–634, 2009.

[8] M. Benichou, J.M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer programming. *Math. Prog.*, 1:76–94, 1971.

[9] Timo Berthold and Ambros M. Gleixner. Undercover – a primal heuristic for MINLP based on sub-MIPs generated by set covering. In Pierre Bonami, Leo Liberti, Andrew J. Miller, and Annick Sartenaer, editors, *Proc. of the EWMINLP*, pages 103–112, April 2010.

[10] Timo Berthold and Ambros M. Gleixner. Undercover: a primal MINLP heuristic exploring a largest sub-MIP. *Math. Prog.*, 2013. doi:10.1007/s10107-013-0635-2.

[11] Timo Berthold, Stefan Heinz, and Stefan Vigerske. Extending a CIP framework to solve MIQCPs. In Jon Lee and Sven Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *The IMA Volumes in Mathematics and its Applications*, pages 427–444. Springer, 2012.

[12] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Edward Rothberg, and Roland Wunderling. MIP: Theory and practice – closing the gap. In M.J.D. Powell and S. Scholtes, editors, *Systems Modelling and Optimization: Methods, Theory, and Applications*, pages 19–49. Kluwer Academic Publisher, 2000.

[13] Michael R. Bussieck, Arne S. Drud, and Alexander Meeraus. MINLPLib – a collection of test models for mixed-integer nonlinear programming. *INFORMS J. Comput.*, 15(1):114–119, 2003.

[14] CppAD. A Package for Differentiation of C++ Algorithms. http://www.coin-or.org/CppAD.

[15] Matteo Fischetti and Michele Monaci. Backdoor branching. In Oktay Günlük and Gerhard J. Woeginger, editors, *Proc. of the 15th IPCO*, pages 183–191. Springer, 2011.

[16] Matteo Fischetti and Michele Monaci. Branching on nonchimerical fractionalities. *OR Letters*, 40(3):159–164, 2012.

[17] GAMS. MINLP Library. http://www.gamsworld.org/minlp/minlplib.htm.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[19] Iiro Harjunkoski, Tapio Westerlund, Ray Pörn, and Hans Skrifvars. Different transformations for solving non-convex trim-loss problems by MINLP. *Eur. J. Oper. Res.*, 105(3):594–603, 1998.

[20] IBM. CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[21] Miroslav Karamanov and Gérard Cornuéjols. Branching on general disjunctions. *Math. Prog.*, 128(1-2):403–436, 2011.

[22] Fatma Kılınç Karzan, George L. Nemhauser, and Martin W. P. Savelsbergh. Information-based branching schemes for binary linear mixed-integer programs. *Math. Prog. Computation*, 1(4):249–293, 2009.

[23] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[24] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proc. of CP*, pages 342–356, Autriche, 1997. Springer.

[25] Jeffrey T. Linderoth and Martin W.P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS J. Comput.*, 11:173–187, 1999.

[26] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the DAC*, July 2001.

[27] SCIP. Solving Constraint Integer Programs. http://scip.zib.de.

[28] Mohit Tawarmalani and Nikolaos V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Math. Prog.*, 99:563–591, 2004.

[29] Stefan Vigerske. *Decomposition in Multistage Stochastic Programming and a Constraint Integer Programming Approach to MINLP*. PhD thesis, HU Berlin, 2012.

[30] Andreas Wächter and Lorenz T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Math. Prog.*, 106(1):25–57, 2006.