Backdoor branching

Matteo Fischetti and Michele Monaci
DEI, University of Padova
via Gradenigo 6/A
35131 Padova, Italy
{matteo.fischetti,michele.monaci}@unipd.it

Submitted: April 29, 2011; Revised: December 18 2011

Abstract

We present an exact Mixed Integer Programming (MIP) solution scheme where a set covering model is used to find a small set of first-choice branching variables. In a preliminary "sampling" phase, our method quickly collects a number of relevant low-cost fractional solutions that qualify as obstacles for the Linear Programming (LP) relaxation bound improvement. Then a set covering model is solved to detect a small subset of variables (a "backdoor", in the AI jargon) that "cover the fractionality" of the collected fractional solutions. These backdoor variables are put in a priority branching list, and a black-box MIP solver is eventually run—in its default mode—by taking this list into account, thus avoiding any other interference with its highly-optimized internal mechanisms. Computational results on a large set of instances from the literature are presented, showing that some speedup can be achieved even with respect to a state-of-the-art solver such as `IBM ILOG Cplex` 12.2.

1 Introduction

Consider a generic Mixed-Integer linear Program (MIP) of the form:

$$
\begin{aligned}
(P)\quad v(P) :=\quad & \min c^T x && (1)\\
& Ax \geq b, && (2)\\
& x_j \text{ integer}, \ \forall j \in \mathcal{I}, && (3)\\
& x_j \text{ continuous}, \ \forall j \in \mathcal{C}, && (4)
\end{aligned}
$$

where A is an $m \times n$ input matrix, and b and c are input vectors of dimension m and n, respectively. The variable index set $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $(\mathcal{I}, \mathcal{C})$, where $\mathcal{I}$ is the index set of the integer variables, while $\mathcal{C}$ indexes the continuous variables, if any.

Bounds on the variables, if any, are assumed to be part of system (2). For a given $S \subseteq \mathcal{I}$, let

$$L(S) := \min\{c^T x : Ax \geq b,\, x_j \text{ integer } \forall j \in S\}$$

denote the lower bound on $v(P)$ obtained by dropping the integrality requirement for all variables but for those in S. Hence $L(\emptyset)$ corresponds to the LP relaxation bound, whereas $L(\mathcal{I}) = v(P)$.

With a little abuse of terminology, in what follows we will say that a point x is *integer* if x_j is integer for all $j \in \mathcal{I}$ (no matter the value of the other components), *fractional* otherwise.

The exact solution of (P) is typically performed by using enumerative schemes where branching is used to partition the solution space; see, e.g., [12]. A large amount of research has been devoted the way branching is actually performed. A customary approach is to branch on a single variable, to be selected according to its impact in the LP bounds of the descendant nodes ([4, 2]), or in the binding constraints ([14, 15]). A different strategy was recently proposed in [11], where a restart scheme is adopted to gather suitable branching information.

The final goal of enumerative methods is to build a tree whose leaf nodes satisfy the following *certificate condition*: the LP relaxation optimal solution x^* is either integer, or $c^T x^* \geq v(P)$—the LP cost being $+\infty$ in case of infeasibility. A branching tree whose leaves satisfy the certificate condition is called a *certificate tree*. For the sake of simplicity, in the above definition we consider a pure branch-and-bound scheme, i.e., cut generation as well as variable fixing and node preprocessing are not taken into account when checking the certificate conditions. In addition, in our definition we assume the optimal solution value $v(P)$ is known in advance, though this hypothesis will be relaxed when actual solution algorithms will be presented.

The computational effort for solving (P) depends on the number of nodes of the certificate tree, that in turn depends on the number of branching variables involved. So, it makes sense to define the *compactness* of a branching tree as the number of distinct branching variables associated to it—the fewer the variables, the more compact the tree. Note however that this measure is not perfect, in the sense that not all tree nodes need to be evaluated explicitly, hence the final computational effort also depends on the tree "shape" and not just on the number of involved variables. In addition, branching on a general integer (as opposed to binary) variable might not fix its fractionality, hence these variables should be counted, e.g., in a way proportional to the logarithm of their range.

A set of variables leading to a compact certificate tree is called a *backdoor* in the AI community. As a matter of fact, different backdoor definitions are possible. We consider the following definition, strictly related to the concept of *strong backdoor* in [16]:

Definition 1 *A set $S \subseteq \mathcal{I}$ is a backdoor if $L(S) = v(P)$.*

Roughly speaking, a backdoor can be viewed as a (hopefully small) set of branching variables leading to certificate tree, i.e., it is enough to require integrality for these variables to get the optimal solution value $v(P)$.

In [9] backdoors are applied to optimization problems, and the backdoor size for some problems from MIPLIB 2003 is reported for the first time. The outcome of this analysis is that, for some problems, fixing values for a very small fraction of the decisional variables is enough to easily solve the problem to optimality.

Our first order of business is to define a model that allows us to exactly compute the minimum size of a set of integer variables leading to a certificate tree. To this end, we first compute the optimal MIP value, $v(P)$, and then treat all the vertices x of P with $c^T x < v(P)$ as *obstacles* whose fractionality has to be *covered* by the branching variable set. This approach is reminiscent of Chvátal *resolution search* [7], where obstacles are associated with minimal sets of integer variables whose fixing leads to infeasibility.

Our second step is more ambitious: we try to use backdoor information on the fly, within the solution procedure, so as to reduce the total computational effort of the MIP solver at hand. As a simple proof-of-concept of the potential of the idea, we implemented the following multiple restart scheme. We make a sequence of short enumeration runs in a "sampling mode" intended to gather information relevant for branching. This approach is in the spirit of the recent work in [11], but in our case we intend to discover and store a collection of relevant low-cost fractional solutions qualifying as obstacles that block the lower bound improvement. More specifically, we use a dynamically-updated cost threshold, and maintain a list of fractional solutions whose cost does not exceed the current threshold. At each sampling run, we start by solving a set covering model to determine a small-cardinality set of branching variables "covering" all fractional solutions in our current list, and treat them as first-level variables with priority 1 for branching (all other variables having priority 0). We then run the MIP solver by using the above branching priorities, and collect additional low-cost fractional solutions. After a while, we abort the current run, and repeat. In this way, more and more low-cost fractional solutions are collected and used to find a clever set of first-level branching variables. After a certain number of restarts, the "final run" is executed by using the MIP solver as a black box, without interfering with its highly-optimized criteria but just giving branching priority 1 to all the variables in the solution of the last set covering problem, and 0 to the remaining ones.

Computational results on a large set of instances from the literature (possibly involving general-integer variables, treated by our set covering model in a heuristic way) are reported, with a comparison with IBM ILOG Cplex 12.2. The outcome is that even a simple proof-of-concept implementation of backdoor branching can lead to a performance improvement.

An early version of the present paper was presented at the IPCO XV meeting, Yorktown Heights, New York, June 15–17, 2011.

2 A basic set covering model

We next show how to compute the *compactness* of a given MIP, defined as size of its minimum-cardinality backdoor—i.e., as the minimum number of branching variables leading to a certificate tree. To have a more meaningful setting, in this section we restrict our attention to 0-1 MIPs, i.e., all integer variables are assumed to be binary, hence the smaller the number of variables in the backdoor, the smaller the associated (complete) certificate tree—assuming again that all the nodes of this tree need to be generated explicitly.

Branching on 0-1 variables (or, more generally, on faces of P) has the nice property of not introducing new vertices of P; see, e.g., [5]. The key property here is that branching could in fact be implemented by just adding (resp. subtracting) a large constant $M > 0$ to the cost c_b of all branching variables fixed to 0 (resp., to 1), so the LP relaxation polyhedron at any node does not change. More formally, if x^k is a vertex of the LP relaxation polyhedron P^k (say) at any given branching node k, then there exists an objective function $w^k x$ that attains its unique minimum over P^k at x^k. By adding/subtracting M to the weight w^k_b of the branching variables x_b along the path from the root to node k, we then obtain a new weight function $\overline{w}^T x$ that has x^k as its unique minimum with respect to P, hence x^k is also a vertex of P.

In view of the above property, we can model our compactness problem as follows. For each $j \in \mathcal{I}$, we introduce a binary variable such that $y_j = 1$ if x_j is in the minimum backdoor set, $= 0$ otherwise. For any vertex x^* of P, let $frac(x^*) := \{j \in \mathcal{I} : x^*_j \text{ is fractional}\}$ denote its *fractional support* w.r.t. $\mathcal{I}$. The problem of computing MIP compactness then calls for a minimum-cardinality set of variables that cover all the fractionalities of all the vertices of P having a cost strictly less than $v(P)$, and can be rephrased as the following set covering problem.

$$\min \sum_{j \in \mathcal{I}} \gamma_j y_j \tag{5}$$

$$\sum_{j \in frac(x^k)} y_j \geq 1, \ \forall \text{ vertex } x^k \text{ of } P : c^T x^k < v(P) \tag{6}$$

$$y_j \in \{0,1\} \ \forall j \in \mathcal{I}, \tag{7}$$

where we set $\gamma_j = 1$ for all $j \in \mathcal{I}$, though a different cost definition can be used to obtain approximate backdoors; see below. Whenever (6) holds for a certain pair y and x^k, we say that "y covers the fractionality of x^k".

The set covering problem above can be tackled through a run-time constraint generation scheme. To this end, for a given set F of points x^k, let $SCP(F)$ denote the set covering model (5)–(7) where (6) is imposed for $x^k \in F$ only. A possible Benders-like solution scheme is then sketched in Figure 1. In this algorithm, we iteratively solve a relaxation of the set covering model where constraints (6) are imposed for a subset F of vertices x^k,

```
1.  compute $v(P)$ and initialize $F := \emptyset$;
2.  Let $y^*$ be an optimal sol. of $SCP(F)$, and define $S := \{j \in \mathcal{I} : y^*_j = 1\}$;
3.  Compute $L(S)$ by an enumerative scheme, and let $\tilde{x}$ be an optimal solution;
4.  **if** $c^T \tilde{x} < v(P)$ **then** add $\tilde{x}$ to $F$, and repeat from 2.
```

Figure 1: Conceptual algorithm for solving model (5)–(7).

while the integrality condition (7) is preserved. For a given optimal set covering solution y^*, our order of business is to prove that the complete branching tree associated with y^* is in fact a certificate tree. This in turn amounts to solving a relaxed MIP obtained from the original one by relaxing the integrality requirement for all x_j with $y_j^* = 0$. If a fractional optimal solutions $\tilde{x}$ of this relaxed MIP is found and $c^T\tilde{x} < v(P)$, the associated constraint (6) is added to the set covering model, and the method is iterated.

As already observed, the method requires the knowledge of $v(P)$. A modified version can also be designed, that does not compute $v(P)$ at step 1 and adds $\tilde{x}$ to F at step 4 after having verified it is not integer. In this way, however, one can only guarantee that each $\tilde{x}$ inserted in F satisfies $c^T\tilde{x} = L(S) \leq v(P)$, meaning that the method can actually over-estimate the backdoor size in the degenerate case where $\tilde{x}$ is fractional but $c^T\tilde{x} = v(P)$.

3 Backdoor branching

As already mentioned, our *backdoor branching* is a multi-restart strategy inspired by the set covering model outlined in the previous section. Its goal is to heuristically use the information associated with the set covering model (5)-(7) to produce a solution scheme for the original MIP that can outperform the standard one, at least on certain classes of hard instances.

Our underlying hypothesis here is that even hard MIPs could be solved by a compact tree (in terms of number of branching variables involved), if an appropriate set of branching variables is chosen. So, we afford spending a certain amount of computing time just to heuristically determine a small set of branching variables leading to a (possibly approximate) certificate tree.

Our proposal can be cast into the following master-slave scheme.

The set covering problem (5)-(7) written for a certain set of vertices x^k acts as our *master problem*: it is solved (possibly heuristically) to get a candidate branching set y^*. In our computational tests (as reported the next section) we did not impose a time limit for its solution, as its optimal solution always required a very short computing time.

If $\sum_{j\in\mathcal{I}} y_j^*$ is reasonably small, then the branching set is compact and we try to use it. To be specific, we solve our MIP as a *slave problem* to possibly generate new x^k's needed to extend (6). To this end, instead of relaxing the integrality conditions on the x_j's with $y_j^* = 0$ we prefer to use a black-box MIP solver on the *original* problem (P) by giving all variables x_j with $y_j^* = 1$ a high priority for branching. This is obtained by just using y^* as a branching priority vector (=1 for the variables to branch first, =0 for the other variables to be branched only as a last resort), so as to let the MIP-solver choose among these variables according to its favorite strategy, e.g., strong branching or alike.

All fractional solutions x^k encountered during the solution of the slave problem are stored and added to the list in (6) to be possibly used at the next iteration. In order to favor the collection of low-cost solutions x^k, a best-bound-first tree exploration strategy is used.

When a prefixed number of new solutions x^k whose fractionality is not covered by y^*

has been collected (say, $K_{max} = 10$) we have the evidence that the current y^* is not an accurate estimate of a good branching set, and we return to the master to get another y^*, i.e., a new tentative branching set.

The master-slave cycle above is iterated until either (i) too many slaves have been solved (say R, where $R = 10$ in our implementation), or (ii) the current master solution y^* has too many 1's (more than, say, Γ, where $\Gamma = 5$ in our implementation), meaning that the MIP likely does not admit a compact branching set. In the latter case, instead of giving up we relax the requirement of having an exact backdoor, and reduce the threshold value of the vertices x^k to be taken into account in (6). In particular, denoting by θ_c and $v(LP)$ the current threshold value and the value of the LP relaxation, respectively, we reduce the threshold by setting

$$\theta_c = \theta_c - (\theta_c - v(LP))/10. \tag{8}$$

At the first iteration, θ_c is initialized to the value of the best integer solution known.

After the solution of the last slave problem, we make the final choice of the branching priorities by solving our last set covering model (5)-(7) over the current set of fractional solutions x^k whose cost is smaller than the current θ_c, and just re-run the MIP solver from scratch (in its default setting) by using the optimal set covering solution y^* as priority vector—this phase being called "the final run".

As the solution of the original MIP (acting as a slave in our scheme) is restarted several times, our method can be interpreted as a multi-restart strategy under the control of a set covering model (the master) that guides the branching process.

Due to its heuristic nature, the backdoor definition above can be quite risky in practice because it may favor variables that happen to cover well the collected fractionalities, but have a minor impact for branching. An extreme case arises when the backdoor includes variables whose integrality requirement is in fact redundant. To mitigate this drawback, we try to exploit the information available for free after each slave solution. To this end, we count how many times each variable has been selected for branching in any of the slaves, and also store the pseudocost vector (see, e.g., [6] and [1]) when the slave is interrupted. Each time a set covering is solved, we treat the never-branched variables as if they were continuous, and modify the set covering objective function to favor the choice of the variables that are likely to have a large impact for branching. To be specific, the cost of each variable y_j ($j \in \mathcal{I}$) in model (5)–(7) is redefined as

$$\gamma_j = M - p_j \tag{9}$$

where M is a sufficiently large positive number, and p_j measures the importance of variable j for branching (the larger the more important). In our proof-of-concept implementation, we set $M = 1000$, whereas coefficient p_j is computed as

$$p_j = \lfloor 100 \cdot \frac{\Psi_j^- + \Psi_j^+}{\Psi_{\max}} \rfloor \tag{10}$$

where Ψ_j^- and Ψ_j^+ denote the average (across all slaves solved so far) pseudocost of variable j in the downwards and upwards branching, respectively, and $\Psi_{\max} = \max\{\Psi_j^- + \Psi_j^+ : j \in$

$\mathcal{I}\}$. According to our computational experience, pseudocosts proved useful to deal with degenerate cases with a lot of ties, for which the set covering model without pseudocosts would have chosen branching variables essentially at random. Different definitions of the p_j's are also possible, that take into account how many times a variable has been selected for branching.

The overall algorithm is sketched in Figure 2, where S, F and θ denote the current (tentative) backdoor, set of fractional solutions to be used in (6), and threshold, respectively.

4 Computational results

In this section we compare our backdoor branching scheme with the state-of-art MIP solver `IBM ILOG Cplex` 12.2. All experiments were performed on an Intel(R) i5 CPU 750 running at 2.67 GHz, in single-thread mode, with time and memory limits equal to 10,000 CPU seconds and 8GB RAM, respectively, for each run.

Testbed and settings
Our testbed was constructed to contain "not too easy nor hopeless" instances, selected according to the following procedure intended to reduce biasing in favor of one of two codes under comparison.

We first considered all instances in the MIPLIB 2003 [3], COR@L [8] and MIPLIB 2010 [13] libraries and solved them by `IBM ILOG Cplex` 12.2 (in its single-thread default setting, with no upper cutoff). Then, we disregarded the instances that could not be solved in the time limit, along with the "too easy" ones that could be solved within just 1,000 nodes or

```
1.   S := ∅; F := ∅; θ := upper bound on v(P);
Sampling Phase
2.   repeat
3.        solve (P) with branching priorities defined by S,
             until K_max new fractional solutions are encountered;
4.        if (P) has been solved to optimality then stop;
5.        add the new fractional solutions to F;
6.        get the current pseudocost values Ψ^- and Ψ^+, and define the cost
             of each set-covering variable according to (9) and (10);
7.        solve SCP(F), and let S be its optimal set;
8.        if |S| > Γ then reduce θ according to (8);
9.   until R iterations have been performed;
Long Run
10.  Solve (P) with branching priorities defined by S.
```

Figure 2: Pseudocode of the Backdoor Branching algorithm.

100 CPU seconds.

Following [11], to have a fair comparison of alternative branching rules with no side effects related to primal heuristics, in the subsequent runs we provided the optimal solution value as the upper cutoff to both solvers and disabled all heuristics, which became useless in this context.

Again to reduce side effects, in all runs we deactivated cut generation for both codes—resulting into a significant slowdown or even a time-limit condition for some instances. In addition, we also had to deactivate (again for both codes) variable aggregation in the preprocessing phase because of its interference with the branching priority mechanism.

Within these settings, some instances in our testbed became too easy, since they were solved to optimality at the root node; thus, we discarded these problems. Finally, we disregarded all instances that were not solved to proven optimality by neither `IBM ILOG Cplex` 12.2 (in our settings) nor backdoor branching. At the very end, the procedure ended up with a testbed of 70 instances.

Results
Table 1 provides the outcome of our experiments on the instances in our testbed. For each instance we give the results of the following algorithms:

- `CPX_default`, i.e., `IBM ILOG Cplex` 12.2 (single thread) in its default setting with no upper cutoff;
- `Cplex`, i.e., `IBM ILOG Cplex` 12.2 (single thread) with our settings about upper cutoff, cut generation, heuristics, and variable aggregation;
- `Backdoor`, i.e., our backdoor branching algorithm (single thread, same setting as `Cplex`).

For each algorithm we give the computing time and number of branching nodes; `t.l.` indicates that the instance was not solved to proven optimality within the 10,000-second time limit.

Computing times and number of nodes reported for our backdoor method sum up the corresponding figures for all (sampling and final) runs; the root node computing time is instead counted only once in that all these runs start exactly from the same root-node information. In addition, we report the size $|B|$ of the "approximate backdoor" used for the last run of the MIP solver, as well as the computing time (column T_{last}) spent by backdoor branching in its final run.

The last lines of the table report the average values (arithmetic and geometric means) of computing time and number of nodes, along with the number of instances solved to proven optimality. Instances that reached the time limit are counted by taking the computing time and number of nodes at the time limit.

According to Table 1, `CPX_default` solves more instances than both `Cplex` and `Backdoor` and requires fewer branching nodes and computing time (in arithmetic mean). This outcome was largely expected due to the use of variable aggregation and cuts that were inhibited in the other two methods. `Backdoor` compares favorably with respect to its direct competitor, `Cplex`, as it solves three more instances and has a computing time and number of branching nodes reduced of about 15% and 25%, respectively (again, in arithmetic mean).

Surprisingly enough, `Backdoor` turns out to be even faster than `CPX_default` and just comparable with `Cplex` when geometric (as opposed to arithmetic) means are considered. This behavior can however be explained by the different number of timelimits incurred by the three methods, that tends to penalize the more clever methods that are able to solve more instances within the given limit. To counteract this distorted behavior of geometric means, it was recently proposed in [10] to penalize the unsolved instances by multiplying by 10 their computing time. If this approach is taken, the geometric means for computing time become 1108, 985, and 813 seconds for `CPX_default`, `Cplex`, and `Backdoor`, respectively.

Table 1: Detailed results for `IBM ILOG Cplex` 12.2 and backdoor branching on a set of hard instances. `t.l.` indicates that the 10,000-sec. time limit has been reached.

	`CPX_default`		`Cplex`		`Backdoor`			
	Time	#nodes	Time	#nodes	Time	#nodes	$\lvert B\rvert$	T_{last}
aflow40b	820.54	116,277	7,191.21	4,253,462	5,252.32	3,185,782	3	5,248.71
app1_2	2,082.97	138,739	163.90	6,064	599.94	12,224	2	521.20
bienst2	135.39	105,238	103.70	85,199	180.44	143,731	4	179.65
csched-010	4,289.02	402,709	2,740.09	2,174,057	3,179.48	2,469,581	5	3,178.10
eilD76.2	6,089.58	20,810	2,069.35	135,801	2,005.36	119,656	3	1,927.05
fast0507	1,746.24	12,746	139.64	3,329	271.55	3,848	2	152.59
glass.lp.sc	1,457.10	54,254	1,495.68	46,123	1,393.02	40,397	2	1,341.66
glass4	1,058.56	940,410	5.11	25,654	23.54	143,656	4	23.36
gmu35_40	1,925.08	1,212,700	`t.l.`	60,707,426	89.29	439,918	5	89.00
markshare_5_0	7,425.75	131,757,859	5,164.27	141,061,124	4,458.08	122,768,127	6	4,457.83
mas74	484.34	2,929,728	218.79	2,993,274	285.56	3,922,440	5	285.45
mcsched	363.65	62,200	682.38	72,102	882.97	162,209	2	862.53
mine_90_10	5,477.70	60,280	721.93	217,682	1,243.69	436,330	2	1,219.02
miplib1	1,084.28	8,447	1,498.72	16,086	1,920.00	16,218	3	1,569.73
neos-1053234	127.98	15,074	24.64	7,405	60.13	17,159	3	56.25
neos-1126860	3,127.37	16,677	3,450.15	18,519	2,962.07	15,993	5	2,821.20
neos-1215891	194.14	7,629	2,068.09	442,404	907.59	178,128	3	895.74
neos-1324574	2,635.90	21,777	5,887.34	52,413	`t.l.`	132,731	3	`t.l.`
neos-1330346	4,797.75	120,265	`t.l.`	516,148	9,984.18	532,478	2	9,963.31
neos-1337307	2,227.09	20,021	8,341.73	475,838	`t.l.`	400,775	5	`t.l.`
neos-1427181	2,395.74	751,589	`t.l.`	4,472,330	2,612.93	934,040	6	2,605.26
neos-1439395	18.16	6,228	21.40	16,866	140.32	109,735	10	138.08
neos-1440460	1,571.19	877,619	2,299.76	1,657,377	935.44	764,126	6	935.01
neos-1451294	2,244.29	10,951	9,630.03	71,278	`t.l.`	85,140	3	`t.l.`
neos-1595230	1,946.95	159,514	347.10	79,340	263.08	66,921	3	259.40
neos-1616732	4,827.34	1,617,471	6,735.23	2,747,954	5,197.28	1,996,385	2	5,187.79
neos-548047	6,763.29	113,081	`t.l.`	108,367	6,571.39	130,069	2	6,521.23
neos-603073	186.50	20,940	2.29	5,318	2.96	6,094	3	2.54
neos-785912	380.49	26,238	`t.l.`	588,153	3,634.81	242,110	4	3,621.32
neos-859770	1,134.46	47,765	2,608.10	147,225	1,524.75	67,867	6	1,513.40
neos-863472	685.15	279,040	20.59	57,617	39.66	114,276	2	39.45

neos-886822	131.93	22,669	82.67	15,360	75.09	15,383	2	65.04
neos-905856	439.43	27,684	894.77	57,865	1,124.87	66,622	3	1,117.66
neos-916792	1,557.94	234,010	586.49	121,654	631.00	139,152	2	622.63
neos13	187.32	14,906	21.20	940	50.49	1,468	2	29.60
neos18	84.84	26,290	235.04	143,654	194.48	122,214	3	189.98
neos5	57.82	362,243	131.34	1,199,529	54.35	447,350	3	53.85
neos788725	4,026.49	1,019,915	218.53	767,350	200.90	755,774	3	199.43
neos818918	2,317.66	254,046	32.10	7,003	154.59	38,194	3	148.65
neos823206	5,073.19	119,609	350.38	205,241	241.86	167,653	3	236.93
noswot	725.30	3,771,681	253.77	2,353,215	127.23	1,154,341	8	127.09
ns1324654	2,560.29	21,777	5,921.72	52,413	t.l.	129,720	3	t.l.
ns1702808	1,477.67	665,309	292.84	251,583	432.82	377,091	5	431.95
ns1766074	500.52	1,067,844	145.98	1,212,703	152.49	1,275,784	5	151.84
ns1830653	666.45	18,269	448.50	42,542	330.59	24,859	4	305.52
ns25-pr3	1,374.60	92,418	1,443.58	219,964	1,427.41	207,664	4	1,421.89
ns25-pr9	256.00	21,137	130.91	23,230	86.35	15,752	3	83.46
ns60-pr3	3,779.10	102,147	t.l.	1,106,164	9,441.38	1,066,696	2	9,433.23
ns60-pr9	661.17	29,275	266.90	38,056	293.26	38,491	2	286.34
opm2.z8.s1_frm00	3,973.91	8,558	2,429.07	10,578	4,473.18	11,564	2	3,540.56
p2m2p1m1p0n100	1,432.92	64,471,749	1,386.26	64,618,970	1,388.07	64,620,437	30	1,387.69
pdh-DBM	41.91	17,566	16.63	56,602	17.77	57,454	2	17.14
pigeon-09	132.69	573,577	65.34	467,325	86.77	621,353	3	86.56
pigeon-10	1,492.41	5,909,259	720.60	4,594,047	739.80	4,676,256	3	739.23
pima.lp.sc	2,758.77	52,452	1,089.11	21,310	532.01	9,680	2	466.52
prob.15.80.100.4.sc	8,302.89	177,646	8,099.56	195,991	8,689.43	187,229	2	8,581.19
prob.20.90.100.0.sc	2,655.14	27,027	2,453.65	23,035	2,382.57	20,743	2	2,151.52
prob.25.80.100.1.sc	3,322.35	23,186	1,240.13	8,050	1,813.49	10,279	2	1,477.92
prob.25.90.100.2.sc	3,028.89	49,164	2,330.25	45,809	3,349.22	45,746	2	3,199.34
prob.5.100.100.0.sc	8,297.75	461,830	2,632.43	185,749	1,993.06	162,617	2	1,961.79
prob.5.100.100.3.sc	4,855.73	325,811	4,213.81	270,359	1,996.72	139,086	2	1,957.00
prod2	144.52	84,898	8.29	21,023	8.98	20,881	5	8.40
pw-myciel4	1,579.25	281,385	191.42	26,842	363.68	60,610	7	357.38
ran16x16	136.84	72,326	1,978.80	14,953,792	2,277.81	16,702,697	4	2,277.63
rocII_4_11	1,336.53	380,593	386.82	158,374	290.50	124,170	5	284.61
rococoC10-001000	601.25	17,587	2,507.89	1,631,422	3,112.39	1,608,535	2	3,108.86
SING290	3,724.97	8,260	t.l.	416,442	5,491.15	123,885	3	5,384.72
sp98ic	1,074.47	44,630	956.75	528,380	1,017.49	536,194	3	1,004.20
tic-tac-toe.lp.sc	2,553.22	47,159	2,759.44	56,423	3,639.74	71,357	2	3,547.06
wpbc.lp.sc	5,043.34	45,406	4,077.87	37,076	3,893.01	45,658	2	3,685.25
arithmetic mean	2,172.45	3,184,080	2,637.60	4,562,971	2,274.25	3,365,088		2,224.94
geometric mean	1,108.64	106,238	711.33	162,999	712.78	158,443		679.27
number of opt.	70		63		66			

We want to stress here that a tighter integration of our restart philosophy into `IBM ILOG Cplex` routines is likely to lead to even higher savings, as the column T_{last} suggests. Also, a tighter integration would allow one to access branching decisions directly, without the need to resort to our very rigid scheme based on static branching priorities.

An interesting outcome of our experiments is that even approximate backdoors of very small size may be quite useful in shaping the enumeration tree in a more effective way—evidently, the first branching variables are so crucial that one can afford spending a non-negligible computing time to better choose them.

We also performed additional experiments on a variant of our backdoor branching where the MIPs solved during the sampling phase use strong branching for a more careful choice of the branching variables, thus making the slave information more reliable and useful for the backdoor choice. This policy turned out to be rather effective in reducing the time spent in the final run, though this did not compensate for the sampling time increase due to strong branching, at least in our current implementation.

5 Conclusions and future directions of work

We have proposed a new branching strategy whose main ingredients are (i) a preliminary sampling phase intended to detect low-cost fractional solutions that act as obstacles for lower bound improvement, and (ii) the definition of a small set of crucial branching variables through the solution of an additional set covering model.

The actual implementation of the above scheme is of course instrumental for its practical success. In this paper we have investigated a simple (proof-of-concept) implementation and have shown the potential of the approach.

Future work should investigate the actual performance of the method when fully integrated within state-of-the-art solvers, a task that however would require a complete access to source codes.

Acknowledgments

Research supported by the *Progetto di Ateneo* on “Computational Integer Programming” of the University of Padova, and by MiUR, Italy (PRIN project “Integrated Approaches to Discrete and Nonlinear Optimization”). Thanks are due to two anonymous referees for their constructive comments.

References

[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin; Fakultät II - Mathematik und Naturwissenschaften. Institut für Mathematik, 2007.

[2] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[3] T. Achterberg, T. Koch, A. Martin. MIPLIB 2003. *Operations Research Letters*, 34:361–372, 2006.

[4] D. Applegate and R.E. Bixby and V. Chvátal and W. Cook. Finding cuts in the TSP. *DIMACS*, Technical report 95-09, March 1995.

[5] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

[6] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[7] V. Chvátal. Resolution search. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 73, 1997.

[8] COR@L: MIP instances. Available at `http://coral.ie.lehigh.edu/ mip-instances/`.

[9] B. Dilkina, C. P. Gomes, and Y. Malitsky. Backdoors to Combinatorial Optimization: Feasibility and Optimality. In Willem Jan van Hoeve and John N. Hooker, editors, *CPAIOR 2009, Proceedings of 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 56–70. Springer, 2009.

[10] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Automated Configuration of Mixed Integer Programming Solvers. In Andrea Lodi and Michela Milano and Paolo Toth, editors, *CPAIOR 2010, Proceedings of 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 186–202. Springer, 2010.

[11] 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.

[12] J.T. Linderoth and M.W.P. Savelsbergh. A Computational Study of Branch and Bound Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[13] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R.E. Bixby, E. Danna, G. Gamrath, A.M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D.E. Steffy, and K. Wolter. MIPLIB 2010 mixed integer programming library version 5. *Mathematical Programming Computation*, 3:103–163, 2011.

[14] J. Patel and J.W. Chinneck. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming*, 110:445–474, 2007.

[15] J. Pryor and J.W. Chinneck. Faster integer-feasibility in mixed-integer linear programs by branching to force change. *Computers and Operations Research*, 38:1143–1152, 2011.

[16] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1173–1178. Morgan Kaufmann, 2003.