Branching on nonchimerical fractionalities

Matteo Fischetti and Michele Monaci

DEI, Università di Padova, Italy
{matteo.fischetti, michele.monaci}@unipd.it

Submitted: 6 April 2011; Revised: 26 October 2011

Abstract

In this paper we address methods for selecting the branching variable in an enumerative exact algorithm for Mixed-Integer Programs—a crucial step for the effectiveness of the resulting method. Many branching rules have been proposed in the literature, most of which are based on the impact of branching constraints on the LP solution values at the child nodes. Among them, strong branching turns out to be the most effective strategy in reducing the number of branching nodes, though its associated overhead may be substantial in most cases.

In this paper we present heuristics to speed-up the strong branching computation, and also to reduce the set of candidate branching variables by removing the variables whose fractionality is just *chimerical*, in the sense that it can be fixed by allowing for a little worsening of the objective function. Extensive computational results on instances from the literature are presented, showing that an average speedup of two can be achieved with respect to a standard full strong branching implementation. This is particularly encouraging if one considers the proof-of-concept nature of our implementation.

1 Introduction

In this paper we address branching strategies for the exact solution of a generic MIP, in minimization form

$$\min\{c^T x : Ax \leq b, x_j \text{ integer } \forall j \in I\}$$

where $x \in \Re^n$, $b \in \Re^m$, and $I \subseteq N := \{1, \cdots, n\}$ denotes the (nonempty) index set of the integer-constrained variables. At any given branching node, we denote by x^* the optimal solution available for the current LP relaxation, and by $F(x^*) := \{j \in I : x^*_j \text{ is fractional}\}$ its fractional support (restricted to the integer-constrained variables). When no confusion arises, we write F instead of $F(x^*)$.

Computationally speaking, branching is one of the most crucial steps in Branch&Cut methods for the optimum solution of general Mixed-Integer Linear Programs (MIPs). In this paper we will concentrate on branching on a single variable (still the most commonly used policy in practice), and will address the key problem of how to select the fractional variable to branch on. This topic has received considerable attention in the last years, starting from the computational study by Linderoth and Savelsbergh [10] on different branching rules.

Several branching criteria have been adopted in the literature. The simplest criterion is to branch on the most-fractional variable, i.e., to select a branching variable x_b ($b \in F$), whose fractional part in the LP relaxation solution x^* is as close as possible to 0.5. This is however well known to be a very poor choice in most cases, in that it performs not significantly better then just choosing x_b at random; see Achterberg, Koch, Martin [3]. This behavior can be illustrated in a qualitative way as follows. Consider a 0-1 knapsack problem with side constraints (possibly including previously-generated cutting planes), where both large and small items are present. Due to side constraints, many fractionalities might arise as a

consequence of the fact that an optimal LP solution has to squeeze 100% optimality, taking full advantage of any possible option in "breaking" the items. As a result, one could face a situation where a single large item with small fractionality is present, together with several small items with any kind of fractionality. A clever branching rule should of course be able to detect the large fractional item, but the presence of a lot of other "disturbing but less relevant" fractionalities might trick it and favor an ineffective branching on a small item. The picture is even worse for MIPs involving big-M coefficients to model conditional constraints, where an "important" binary variable activating a critical constraint often has an LP value very close to zero, so it is seldom used for branching.

A much more robust branching policy, known as *Full Strong Branching* (FSB), was proposed by Applegate, Bixby, Chvátal and Cook [4]. It consists in simulating branching on all fractional variables and in selecting the actual branching variable x_b among those that give the best progress in the dual bound. This approach typically leads to much smaller branching trees, but introduces a quite large overhead at each node because of the need of solving two LPs for each candidate branching variable. Cheaper versions are therefore preferred in practice, where strong branching is applied to a (small) subset of the fractional variables, LP reoptimizations are aborted after a small number of dual pivots, and other information such as *pseudocosts* [5] is used to avoid LP reoptimizations as much as possible. A quite effective scheme based on pseudocosts is known as *reliability branching* and was proposed by Achterberg, Koch, Martin [3]. The approach by Patel and Chinneck [12] and by Pryor and Chinneck [13], instead, does not use pseudocost information but is based on the impact of branching on the constraints that are active in the current LP relaxation; computational experiments show that this allows one to get an integer solution faster than in the traditional approaches. Achterberg [1, 2] proposed *inference branching*, a method strictly related to SAT solvers that considers domain reductions deriving from branching constraints. Karzan, Nemhauser, and Savelsbergh [8] recently proposed a more sophisticated approach where a restart strategy is used to collect information that is used in the branching variable selection. In [7] we analyzed a *backdoor branching* strategy also based on restart.

Extensive computational experiments on different branching policies have been performed in Achterberg [2] by using the open-source Branch&Cut software SCIP. A comparison between FSB and the best available branching rule (reliability branching) shows that FSB reduces the average number of nodes by 75%, while doubling computing time. As a consequence, a speedup of 2x in the FSB implementation would suffice to make it be the fastest option—at least, in the considered SCIP environment.

In the present paper we aim at finding a computationally cheaper way to gather the same information as FSB by solving fewer LPs. In order to test new research ideas and directions, we decided to try to escape the "local optimal" common practice where a clever mixture of strong branching and pseudocost information is used. In particular, we deliberately avoided to exploit any pseudocost information in our methods, so as to be forced to find valid alternatives—though future practical implementations of our ideas could benefit from exploiting pseudocost information as well, as outlined in our concluding remarks.

The paper is organized as follows. In Section 2 we discuss three FSB variants that allows one to obtain similar information as FSB by solving a significantly smaller number of LPs. In Section 3 we evaluate the effect of the new branching rules on a large testbed of hard MIPs from the literature, showing that a speedup of two can be achieved with respect to a standard FSB implementation, thus making the proposed variants comparable with more sophisticated branching rules implemented in the state-of-the-art codes. Some conclusions

are finally drawn in Section 4, with a preliminary analysis with a hybrid branching scheme making use of pseudocost information.

2 Chimerical vs nonchimerical fractionalities

According to the previous discussions, we found it useful to qualitatively distinguish between:

i) *chimerical* fractionalities x_j^* that can be fixed by only a small deterioration of the objective function, i.e., such that an almost-optimal LP feasible solution $\widetilde{x}$ exists satisfying either $\widetilde{x}_j \leq \lfloor x_j^* \rfloor$ or $\widetilde{x}_j \geq \lceil x_j^* \rceil$

ii) *nonchimerical* fractionalities x_j^*, for which branching on x_j has a large impact on the LP solution.

According to the discussion on knapsack problems of the previous section, we may think of chimerical fractionalities as being associated with small items, whereas nonchimerical fractionalities correspond to large items.

FSB can be viewed as a computationally-expensive way to detect nonchimerical fractionalities. (Another very interesting option to detect nonchimerical fractionalities is to study the implication of branching constraints through, e.g., constraint propagation, as suggested by Patel and Chinneck [12]).

In Section 2.1 we address the design of a specialized FSB implementation. This procedure turns out to produce the same information as the standard implementation of FSB, but requires the solution of a smaller number of LPs. Sections 2.2 and 2.3 present two computationally cheaper alternatives to FSB, that are still able to get rid of chimerical fractionalities that would confound branching.

2.1 Parametrized full strong branching

A standard FSB implementation works as follows. Recall that F denotes the fractional support of the optimal LP solution x^* found at a given branching node. Let $\texttt{score}(l_0, l_1)$ be the score function to be maximized when selecting the branching variable, e.g., $\texttt{score} = \min(l_0, l_1)$, or $\texttt{score} = l_0 * l_1$, or alike. We assume function `score` to be monotone, in the sense that decreasing argument l_0 and/or l_1 cannot improve the associated score—as it is the case in practice for all reasonable `score` functions one can think of. For each $j \in F$, one solves two LPs to compute $LB0_j$ (resp. $LB1_j$) as the LP-value increase when branching down (resp. up) on x_j, thus obtaining $\text{score}_j = \texttt{score}(LB0_j, LB1_j)$. In the end, a variable x_b with maximum score_b is selected for branching.

In order to save the solution of as many LPs as possible, a parametrized version can be designed along the following lines. Values $LB0_j$ and $LB1_j$ are treated as just upper bounds on the corresponding LP worsenings, and a flag $f0_j$ (respectively, $f1_j$) is used to indicate whether $LB0_j$ (resp., $LB1_j$) has been computed exactly. A key observation is that the LP solution computed when simulating branching on a certain variable x_k can be used to update the upper bound values $LB0_j$ and/or $LB1_j$ for other variables x_j, thus hopefully saving LP computations at a later time. (We became recently aware that a similar technique was independently suggested, for disjunction branching, by Ashutosh Mahajan in his PhD dissertation [11].)

Here is a rough sketch of how our parametrization works.

1. Initialize $LB0_j := LB1_j := \infty$ and $f0_j := f1_j :=$ `FALSE` for all $j \in F$
2. Updating algorithm
   (a) Look for the candidate variable x_k that has maximum score,
   (b) if ($f0_k = f1_k =$ `TRUE`) DONE
   (c) exactly compute $LB0_k$ (or $LB1_k$), update $f0_k$ (or $f1_k$), and repeat
   (d) KEY STEP: whenever a new LP solution $\widetilde{x}$ is available:
      i. possibly update $LB0_j$ if $\widetilde{x}_j \leq \lfloor x^*_j \rfloor$ for some $j \in F$
      ii. possibly update $LB1_j$ if $\widetilde{x}_j \geq \lceil x^*_j \rceil$ for some $j \in F$

A more detailed description is given in Algorithm 1.

Algorithm 1: Parametrized Full Strong Branching—the basic scheme.

input : the current LP solution x^* and its fractional support $F \neq \emptyset$;
output: index k of the branching variable;

initialize $LB0_j := LB1_j := \infty$ and $f0_j := f1_j :=$ `FALSE` $\forall j \in F$;
while `TRUE` **do**
  compute $\text{score}_j =$`score`$(LB0_j, LB1_j)$ for all $j \in F$;
  let $k := \arg\max\{\text{score}_j : j \in F\}$;
  if ($f0_k = f1_k =$ `TRUE`) **then** return(k);
  if ($f0_k =$ `FALSE`) **then**
    solve LP with the additional constraint $x_k \leq \lfloor x^*_k \rfloor$, thus obtaining solution $\widetilde{x}$;
    set $\delta := c^T\widetilde{x} - c^T x^*$, $LB0_k := \delta$, and $f0_k :=$ `TRUE`
  else
    solve LP with the additional constraint $x_k \geq \lceil x^*_k \rceil$, thus obtaining solution $\widetilde{x}$;
    set $\delta := c^T\widetilde{x} - c^T x^*$, $LB1_k := \delta$, and $f1_k :=$ `TRUE`
  end
  for $j \in F$ s.t. $\widetilde{x}_j \leq \lfloor x^*_j \rfloor$ **do**
    $LB0_j := \min\{LB0_j, \delta\}$;
    if ($\delta = 0$) **then** $f0_j :=$ `TRUE`
  end
  for $j \in F$ s.t. $\widetilde{x}_j \geq \lceil x^*_j \rceil$ **do**
    $LB1_j := \min\{LB1_j, \delta\}$;
    if ($\delta = 0$) **then** $f1_j :=$ `TRUE`
  end
end

As already mentioned in the introduction, the standard implementation of FSB embedded within almost all commercial solvers halts each LP reoptimization after a small number of dual pivots, so as to keep computing time under control. When such a limit is imposed, one can still use the parametrization algorithm, in which case some of the $LB0$ and $LB1$ values may not represent valid upper bounds on the associated LP worsenings. Indeed, computational experiments reported in Section 3 show that the use of such pivot limits does not affect the viability of our method.

2.2 Perseverant branching

A common practice to limit strong-branching overhead is to restrict the list F of the fractional variables candidate for branching. A rather effective heuristic is based on the use of pseudocost information collected in the previous nodes, that provide statistics on the effect of branching on certain variables; we refer the reader to Achterberg [1] for an in-depth discussion of this topic.

We next propose a different approach to reduce the candidate branching list, based on the observation that the use of a "robust" criterion such as strong branching makes it very unlikely to select a chimerical branching variable at the top of the branching tree. So, it makes sense to insist as much as possible on branching on a variable that was already selected as the branching variable at a previous node. A further motivation for such a choice is to hopefully reduce the size of the overall branching set needed to complete the enumeration, acting as a heuristic for the so-called *backdoor* introduced by Dilkina et al. [6].

The resulting policy, that we call *perseverant branching*, is a simple variant of FSB that consists of keeping in the candidate strong-branching list F only the indexes of the variables that were already selected for branching in a previous branching node. Of course, in the case the reduced list would be empty (as it happens, e.g., at the root node) we restore the original list F containing the indexes of all fractional variables, and apply FSB to it.

2.3 Asymmetric branching

Our third modification of standard FSB is based on the observation that, for most practical instances, the "down branching" $x_k \leq \lfloor x_k^* \rfloor$ is more critical than the "up branching" $x_k \geq \lceil x_k^* \rceil$. This is particularly true for 0-1 MIPs where setting a certain binary variable to 1 corresponds to a relevant choice that implies tight restrictions, whereas the opposite setting to 0 leaves the problem almost unchanged (as, e.g., in set covering/partitioning or set packing problems, or in network design problems modeled through big-M conditions triggered by the value 1 of certain binary variables). For these problems, just a few up branches in a diving path suffice in fathoming the current node, and the total number of branching nodes mainly depends of down-branching choices.

In this situation, one can argue that the entries of $LB1$ do not convey relevant branching information, in the sense that a sensible score would mainly depend on the entries of $LB0$. Therefore one would like to avoid solving the LPs needed for the exact computation of any $LB1_k$. With this aim, we define an "asymmetric" variant of our parametrized FSB where the flags $f1_j$ are all initialized to `TRUE`, meaning that values $LB1_j$ will be possibly updated (for free) but never be responsible for the overhead incurred when solving a new LP.

Finally, we observe that asymmetric branching is very much related to the structure of the problem at hand, hence it has to be used with some care as we cannot expect it to work well in all cases—e.g., flipping the binary variables of a 0-1 MIP model does not change the problem at hand, but likely would have a bad impact on the performance of asymmetric branching.

3 Computational results

The following two main `score` functions, denoted as `min` and `prod`, respectively, have been proposed in the literature:

- $\texttt{score}(LB0, LB1) = \min(LB0, LB1)$

- `score`$(LB0, LB1) = \max(LB0, \epsilon) * \max(LB1, \epsilon)$

where ϵ is a small value used to break ties in case one of the two terms $LB0, LB1$ is zero. The latter function was proposed by Achterberg [1], and is currently considered the best option for full strong branching. We also report results on the `min` score to see the effects of our criteria on top of this option.

All experiments were performed on a PC Intel Core i5 running at 2.67GHz. Our procedures were coded in C language and were embedded within the general solver `IBM ILOG Cplex` 12.2 using callback functions. All codes were run in single-thread mode. We addressed all the instances considered in Achterberg, Koch, Martin [3], plus all those from Karzan, Nemhauser, Savelsbergh [8] that are not classified as "easy" and all instances of the recent MIPLIB2010 library of instances [9] that belong to classes "benchmark", "hard" and "tree". For each such problem, we considered a modified instance obtained after the root-node preprocessing routines have been applied. Following [8], to have a fair comparison of branching rules, in all runs we disabled all further preprocessing, heuristic and cut generation procedures, and provided the solver with the optimal solution value as the upper cutoff.

Firstly, we solved all instances with our implementation of the standard FSB strategy with an upper bound of 20 dual pivots for each LP reoptimization inside the branch-selection procedure, `prod` score function, and a time limit of 10,000 CPU seconds. We removed all instances not solved to optimality within the given time limit, plus those just solved by the preprocessing routine, thus obtaining our final testbed containing 60 instances.

For each instance we compared:

- a standard full strong branching implementation (FSB); for a fair comparison, we did not use `IBM ILOG Cplex`'s internal procedure `CPXstrongbranch` for computing strong-branching information, because this procedure has direct access to the internal data-structure and does not need, e.g., to refactor the basis each time the optimal LP basis is restored and a new branching is simulated;
- the parametrized version of FSB described in Section 2.1 (PFSB);
- the perserverant (parametrized) branching procedure described in Section 2.2 (PPFSB);
- the asymmetric perseverant (parametrized) branching procedure described in Section 2.3 (APPFSB);
- our implementation of *Reliability Branching* (RB) [3, 1], intended as a proxy for the default branching rule adopted by state-of-the-art solvers. According to this rule, the `prod` score is used, the score for a given candidate variable being computed according to pseudocosts only if the variable is reliable, i.e., if it has been selected for branching at least $\eta_{\texttt{rel}}$ times; as suggested in [1], we used $\eta_{\texttt{rel}} = 8$. For non-reliable candidates, the highly-optimized `CPXstrongbranch` Cplex's function was used to compute strong branching information.

Table 1 gives the outcome of our experiments; more details are reported in the Appendix (Table 4). For each algorithm and score function we report the geometric mean of computing time (in CPU seconds) and, in parenthesis, the associated ratio with respect to the computing time of the reference algorithm (i.e., the standard implementation of FSB), the number *#uns.* of instances that are not solved to optimality within the time limit, and the number of

	min				prod			
	Time		#uns.	#nodes	Time		#uns.	#nodes
FSB	749.25	(1.59)	5	36,240	471.00	(1.00)	0	20,916
PFSB	446.82	(0.95)	4	39,096	355.68	(0.76)	0	20,852
PPFSB	262.36	(0.56)	2	39,193	214.08	(0.45)	0	21,563
APPFSB	212.62	(0.45)	1	45,923	199.41	(0.42)	1	43,956
RB					226.19	(0.48)	1	49,681

Table 1: Average results (geometric means) for different branching rules with at most 20 dual pivots for each LP reoptimization.

branching nodes. Note that our implementation of both FSB and PSFB uses exactly the same lexicographic rule to break ties in the choice of the branching variable, so the two methods produce the same enumeration tree—small variations in the total number of nodes only depend on very rare numerical issues that trick the lexicographic order. As the optimal solution value is provided on input and cut generation is deactivated, the number of nodes explored during enumeration should in principle only depend on the branching policy, i.e., node selection should play no role in our setting. However, numerical considerations related, e.g., to the frequency of LP basis refactorization may change the LP solution and hence the whole search path in an erratic way, so in our experiments we decided to use `IBM ILOG Cplex`'s default node selection rule.

As expected, Table 1 shows that the `min` function is much worse than its `prod` counterpart in the standard FSB setting, as it leads to an almost doubled number of branching nodes and to an increase of more than 50% of the overall computing time (on average, 749 instead of 471 seconds). However, our parametrization scheme proved very well suited for the `min` score, and the difference between the two scores is less pronounced in the PFSB setting (on average, 446 seconds for `min` instead of 355 seconds for `prod`). The difference between the two score functions becomes even smaller for PPFSB and for APPFSB. In any case, the `prod` score qualifies as the best option for all methods.

As to the computing time required by the four methods when using the same `prod` score, PSFB, PPFSB and APPFSB exhibit a speedup factor with respect to FSB of about 1.3, 2.2 and 2.3, respectively. Still with respect to FSB, the number of nodes is almost unchanged for PFSB (as expected), slightly increased by PPFSB, and doubled by APPFSB. As already mentioned, the number of FSB and PSFB nodes are not always identical because ties can be broken in a different way. The perseverant branching rule seems to be beneficial in reducing the time per node without significantly affecting the total number of nodes, whereas the main merit of the asymmetric rule is the reduction of the computational overhead spent at each node to detect the branching variable. For the instances in our testbed, RB produced slightly worse results than APPFSB, in terms of both computing time and number of nodes.

Tables 2 gives the same information as in Table 1, when the algorithms are executed with different limits on the number of dual pivots for each LP reoptimization within the branching procedure, namely 10, 50 and no upper bound, respectively. For the sake of comparison, the last line of the table reports the performance of our Reliability Branching implementation, taken from Table 1.

	pivot	min				prod			
	k	Time		#uns.	#nodes	Time		#uns.	#nodes
FSB	10	944.77	(2.01)	13	83,843	909.15	(1.93)	16	71,964
PFSB	10	734.77	(1.56)	12	94,345	721.98	(1.53)	13	82,268
PPFSB	10	580.32	(1.23)	8	92,776	574.34	(1.22)	12	84,536
APPFSB	10	485.77	(1.03)	9	122,676	496.69	(1.05)	9	125,035
FSB	50	775.92	(1.65)	8	31,635	431.90	(0.92)	0	16,419
PFSB	50	432.25	(0.92)	4	35,209	312.30	(0.66)	0	16,489
PPFSB	50	254.52	(0.54)	1	34,581	204.51	(0.43)	0	17,934
APPFSB	50	226.50	(0.48)	2	47,236	224.08	(0.48)	1	45,255
FSB	inf	871.27	(1.85)	8	26,597	513.99	(1.09)	2	13,273
PFSB	inf	438.86	(0.93)	5	32,782	335.79	(0.71)	1	13,988
PPFSB	inf	278.61	(0.59)	2	33,866	222.76	(0.47)	1	15,144
APPFSB	inf	211.23	(0.45)	2	41,826	212.43	(0.45)	3	41,228
RB						226.19	(0.48)	1	49,681

Table 2: Average results (geometric means) for different branching rules with at most k dual pivots for each LP reoptimization within the branching procedure.

The results confirm that PFSB leads to a considerable speedup with respect to the standard FSB implementation, whichever the `score` function and the upper limit on the number of dual pivots. Again, further reductions of the computing times can be obtained by also implementing the perseverant and asymmetric branching rules.

4 Conclusions and future work

We have proposed simple modifications of the well-known full strong branching policy and have shown through computational tests that they lead to a significant performance improvement.

Future research should investigate a tighter integration of the new policies with state-of-the-art strategies for branching using pseudocost information. As a first step in this direction, we next report the outcome of preliminary experiments with a simple `hybrid` branching strategy using pseudocost information. To be specific, we use a method based on full strong branching (FSB or PFSB or PPFSB) at the first 10 levels of the branching tree and for the first θ branching nodes, where θ is an input parameter, and then switch strong branching computation off and resort to a standard pseudocost selection rule. The rational of this scheme is that, when enough branching nodes have been explored, pseudocost information is assumed to be reliable enough to allow for an effective selection of the branching variable. Note that, in this new context, the APPFSB scheme is less attractive in that most of the computing time is expected to be spent after the first θ nodes, so the increased number of nodes expected as a consequence of the use of the asymmetric branching rule is not likely to be compensated by a substantial reduction of the average computing time per node.

Table 3 reports the average values (geometric means) of the computing time and number of nodes, along with the number of unsolved instances, for different different values of θ of the `hybrid` algorithm when strong branching is implemented, respectively: (i) in the standard

way, (ii) in the parametric version of Section 2.1, and (iii) in the perseverant parametrized of Section 2.2. All these algorithms were run with the dual pivot limit $k = 20$. The performance of our Reliability Branching implementation is also given. For benchmarking purposes, the table also reports the performance of `IBM ILOG Cplex` 12.2 in its default setting (without heuristics and any further preprocessing) when the optimal solution value is given on input. Observe however that this software has a native (highly-optimized) access to its own internal structures, and is allowed to fully exploit strong branching information for variable fixing, propagation, and early node fathoming. Hence a direct comparison of `IBM ILOG Cplex`'s performance with that of the other methods would be not completely fair. For the methods for which the value of θ is immaterial, only the first row of the table is used.

	FSB+pcosts			PFSB+pcosts			PPFSB+pcosts			RB			`IBM ILOG Cplex`		
θ	Time	#nodes	#uns.	Time	#nodes	#uns.	Time	#nodes	#uns.	Time	#nodes	#uns.	Time	#nodes	#uns.
0	223.24	95,727	1	214.14	96,055	0	143.03	75,855	0	226.19	49,681	1	48.00	32,825	0
500	212.90	64,134	1	194.98	64,152	0	133.68	55,419	0						
1,000	226.50	58,848	1	203.12	58,959	1	131.60	50,374	0						
5,000	258.38	41,092	0	214.74	41,067	0	138.32	39,205	0						
10,000	282.23	36,268	0	237.67	36,268	0	148.31	34,964	0						

Table 3: Average results (geometric means) for hybrid algorithms exploiting pseudocosts.

According to Table 3, for any value of θ our parametrized full strong-branching version (PFSB+pcosts) is about 10% faster than the standard one (FSB+pcosts), while the additional use of our perseverant rule (PPFSB+pcosts) adds a further 30% speedup. As long as computing time is concerned, the best results for both FSB+pcosts and PFSB+pcosts are obtained for $\theta = 500$, whereas increasing the value of θ to 1,000 allows PPFSB+pcosts to reduce by more than 30% the computing time, and by about 20% the number of nodes. Reliability Branching turns out to be competitive in terms of number of nodes, but it does not produce improved results in terms of average computing time—at least, in our implementation. As expected, `IBM ILOG Cplex` 12.2 turns out to be significantly faster than all other methods, as its proprietary *dynamic search* policy leads to a number of nodes comparable to case $\theta = 10,000$ but requires much shorter computing time as a result of its tight integration within the solver.

Future work should address the integration of the ideas presented in this paper into more sophisticated (open source or commercial) solvers; a first step in this direction has been already performed by the IBM ILOG Cplex's team [14] in the latest version.

5 Appendix

Table 4 gives more detailed results on the performance of FSB, FSB+pcosts, APPFSB, PPFSB+pcosts, each with its best-tuned parameter θ. For all algorithms, an upper bound of 20 dual pivots is imposed for each LP reoptimization within the branching procedure, and the `prod` function is used. In addition, results for both Reliability Branching and `IBM ILOG Cplex` 12.2 (the latter in its default setting, providing the optimal solution value on input and disabling preprocessing and heuristics) are given. The last two lines of the table report geometric means for each algorithm: the former considers all the 60 instances in our testbed, whereas the latter includes only those problems that were solved to proven optimality by all

the algorithms—all instances but problems `neos-1396125`, `neos13` and `ns1688347`.

Performance profile plots about computing time and number of nodes are given in Figure 1 for all the algorithms considered in Table 4. As already observed, `IBM ILOG Cplex` 12.2 is the clear winner in terms of speed, PPFSB+pcosts being its best alternative. As to branching nodes, the best method is FSB (as expected), followed by `IBM ILOG Cplex` and APPFSB.

Acknowledgments

This research was 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 also due to an anonymous referee for his/her helpful 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. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

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

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

[5] 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.

[6] B.N. Dilkina, C.P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In Willem Jan van Hoeve and John N. Hooker, editors, *CPAIOR*, volume 5547 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2009.

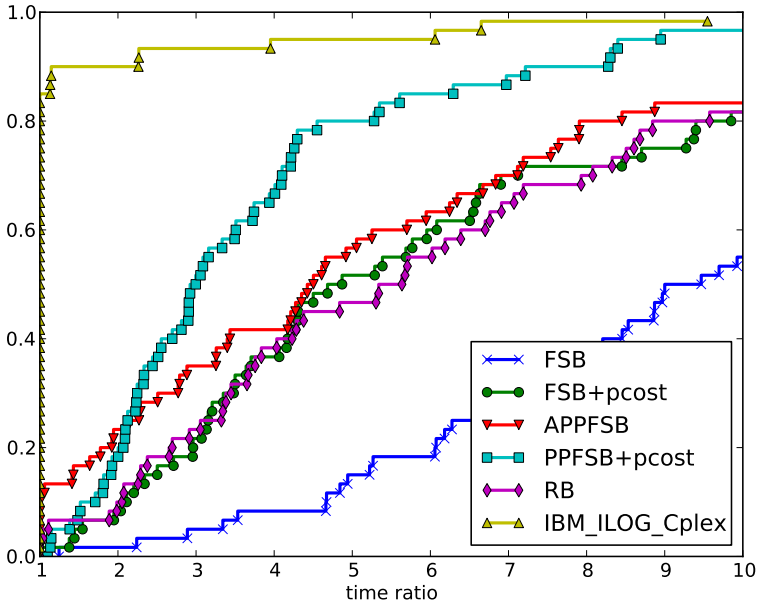
[7] M. Fischetti and M. Monaci. Backdoor branching. In O. Günlük and G.J. Woeginger, editors, *Integer Programming and Combinatorial Optimization (IPCO 2011)*, pages 183–191, Berlin Heidelberg, 2011. Springer Lecture Notes in Computer Science 6655.

[8] 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.

[9] 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.

	FSB		FSB+pcosts(1)		APPFSB		PPFSB+pcosts(2)		RB		IBM ILOG Cplex	
	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes	Time	#nodes
aflow30a	19.70	1,751	13.44	2,951	7.92	3,891	6.28	2,381	9.98	4,361	2.33	3,520
aflow40b	4,624.44	64,383	2,015.43	1,219,621	8,962.26	899,513	1,151.24	698,563	1,301.33	677,531	307.67	141,435
air04	128.09	69	125.41	69	129.28	3,245	56.11	101	128.47	37	18.17	913
ash608gpia-3col	1,842.05	49	1,647.31	49	287.88	47	537.16	47	8,697.42	39	175.74	1,493
bc1	155.30	2,075	100.05	3,419	412.20	16,051	133.40	8,779	464.62	11,049	69.33	8,799
biella1	5,226.06	485	4,970.47	485	5,168.81	28,565	1,521.67	686	4,311.37	595	379.45	11,785
bienst1	107.01	8,317	65.19	10,875	56.82	8,307	61.23	10,947	84.99	13,227	32.01	7,525
bienst2	3,661.54	169,285	1,264.50	220,573	1,296.70	133,893	1,548.85	342,837	1,384.96	286,209	377.64	76,033
cap6000	0.52	2,091	0.38	2,256	0.38	2,436	0.42	2,531	0.55	2,439	0.18	2,543
eil33-2	295.85	5,481	149.99	12,093	71.71	7,425	129.50	10,577	668.68	5,661	36.90	10,747
eilB101	2,134.55	3,785	857.13	25,789	388.37	7,693	587.67	7,545	2,167.19	6,137	84.29	10,380
k16x240	3,473.72	1,076,605	9,707.63	25,497,899	3,935.35	4,274,309	6,809.97	18,380,957	8,122.28	21,201,137	2,787.99	6,899,304
markshare_4_0	160.32	1,318,751	70.16	2,121,453	113.82	1,684,491	67.25	1,958,071	90.07	1,958,373	45.37	1,579,769
mas284	19.94	11,403	11.38	56,013	15.41	27,109	9.57	43,337	9.32	49,691	2.43	14,931
mas74	1,658.65	2,458,293	976.87	13,233,467	1,928.81	7,669,915	1,504.19	21,358,187	2,309.28	26,375,957	268.32	3,482,938
mas76	158.91	314,153	82.69	1,279,467	83.48	471,849	83.51	1,382,135	105.81	1,422,433	19.82	322,265
mine-166-5	275.27	2,089	299.77	21,615	397.53	15,081	110.19	7,569	439.00	3,617	52.72	14,095
misc07	164.83	24,715	60.92	137,995	17.41	15,491	19.95	39,385	36.11	48,909	19.95	47,543
neos-1109824	1,373.63	5,985	265.27	10,331	6,414.92	374,257	142.59	12,321	394.68	12,627	63.79	10,007
neos-1200887	7,160.53	71,089	1,148.97	652,047	715.35	57,766	773.53	700,709	866.04	767,645	107.20	69,794
neos-1211578	502.97	133,201	170.24	491,769	86.54	89,355	103.26	266,009	103.74	285,937	19.56	47,000
neos-1228986	312.91	96,783	741.41	2,909,265	164.24	187,035	291.97	968,813	212.41	659,949	35.28	79,615
neos-1396125	4,974.97	31,858	10,000.00	621,701	651.80	39,433	900.87	111,778	5,168.72	316,927	6,222.77	386,111
neos-1440447	224.99	35,835	130.86	162,282	48.26	23,801	121.37	154,275	110.50	150,724	54.58	77,220
neos-476283	904.43	676	847.13	795	154.89	405	426.05	740	457.87	1,162	67.71	525
neos-480878	1,582.92	40,307	280.55	102,847	1,373.16	267,803	275.53	132,871	551.03	94,335	24.30	14,645
neos-504674	2,656.27	47,347	550.73	402,409	332.49	40,851	361.47	303,805	384.99	241,489	43.53	44,945
neos-538867	1,779.52	114,385	417.23	676,531	83.75	40,929	124.45	169,985	120.21	151,941	58.59	78,747
neos-538916	1,002.72	50,553	189.84	212,627	60.56	17,993	121.28	154,765	163.25	180,441	137.01	176,225
neos-598183	1,517.60	146,948	369.29	225,764	1,062.14	281,719	286.68	186,089	308.25	192,469	82.06	54,919
neos-803220	540.93	101,929	345.07	495,005	798.91	348,657	212.90	300,631	374.16	502,265	52.04	72,497
neos-807639	177.88	10,859	58.48	15,411	39.15	11,861	45.27	15,295	45.14	15,489	13.58	11,944
neos-860244	458.08	7,643	122.72	22,495	115.59	7,133	85.72	11,209	1,464.66	67,547	41.44	13,175
neos-863472	1,282.09	354,815	880.16	2,615,739	274.52	299,947	867.62	2,553,195	285.67	750,681	622.35	1,777,647
neos13	42.94	25	38.08	25	10,000.00	17,185	30.10	25	60.14	25	7.07	99
neos2	264.83	30,911	36.00	38,525	59.34	41,123	29.67	38,233	1,302.24	213,769	26.15	38,452
neos22	8,327.22	176,935	582.83	254,899	387.26	79,649	554.88	265,399	896.31	291,275	62.01	25,748
neos23	2,480.39	846,309	255.03	777,665	217.44	206,807	102.38	296,189	120.37	310,025	27.50	94,990
neos3	3,487.84	266,887	372.65	474,621	1,429.89	837,805	354.45	452,707	1,468.85	236,231	169.17	236,268
neos5	3,175.48	2,453,537	3,104.57	28,172,115	2,821.20	7,431,395	2,230.36	20,404,671	2,138.79	17,938,181	109.16	932,657
ns1688347	5,499.38	6,491	3,206.77	8,455	1,044.36	2,577	1,897.09	5,871	10,000.00	834	1,182.93	3,162
ns1766074	1,276.08	1,042,239	217.31	1,243,865	404.83	1,103,621	231.52	1,331,353	234.11	1,272,179	210.64	1,244,037
pk1	243.60	231,363	146.09	1,501,235	110.49	342,931	131.23	1,391,093	138.30	1,315,035	24.54	219,590
pp08aCUTS	4.59	687	4.31	767	1.66	1,285	2.32	881	2.91	1,263	0.51	1,035
prod1	219.14	38,091	618.67	1,982,291	104.40	90,495	205.19	615,511	622.39	1,902,529	24.43	59,360
prod2	576.96	40,803	280.31	483,459	320.30	129,515	189.30	319,749	369.66	521,169	65.12	102,094
qiu	5,930.59	254,317	353.55	61,589	233.61	19,775	230.68	45,781	641.10	87,705	53.70	12,388
rail507	4,436.85	1,905	3,189.70	3,213	673.09	6,561	353.91	1,585	754.64	1,847	118.12	3,029
ran10x26	68.60	5,499	29.78	21,203	63.54	39,735	23.32	30,465	27.72	15,437	8.04	13,815
ran12x21	398.86	36,961	121.37	194,267	341.21	214,105	112.44	181,239	128.55	197,025	38.46	70,902
ran13x13	75.63	12,793	30.16	45,635	39.26	36,537	27.06	49,043	44.01	64,323	12.05	29,664
ran16x16	968.11	98,959	291.21	509,191	653.40	362,427	289.62	535,639	295.60	504,823	124.42	206,191
ran8x32	18.48	3,501	11.06	18,223	15.61	16,969	10.52	21,913	12.65	24,943	3.74	10,899
rmatr100-p10	284.18	959	279.44	989	601.37	8,681	124.91	1,049	173.25	1,169	42.96	1,037
rmatr100-p5	226.51	489	227.79	489	319.76	2,469	104.04	519	323.24	509	46.78	484
rocII-4-11	1,305.37	11,863	354.21	62,571	4,302.84	339,747	276.04	143,608	181.83	14,131	718.78	378,793
stein45	86.26	23,817	29.41	67,593	19.09	29,857	17.60	64,953	15.56	85,895	6.89	45,721
swath1	466.44	29,387	106.96	68,591	20.23	5,017	36.57	12,661	38.19	11,921	122.59	96,924
swath2	3,677.27	260,607	803.04	668,455	86.41	24,229	139.33	105,321	81.52	50,729	542.16	418,494
vpm2	2.56	971	2.25	1,143	1.87	3,171	1.56	1,817	1.79	1,601	0.37	1,679
Avg. (all)	471.00	20,917	212.90	64,134	199.41	43,957	131.60	50,374	226.19	49,681	54.19	33,238
Avg. (opt)	451.42	23,849	195.56	73,288	177.12	47,057	124.60	58,950	205.05	59,031	43.09	36,264

Table 4: Detailed results with at most $k = 20$ dual pivots for each LP reoptimization inside the branching procedure. $^{(1)}\theta = 500$, $^{(2)}\theta = 1,000$.



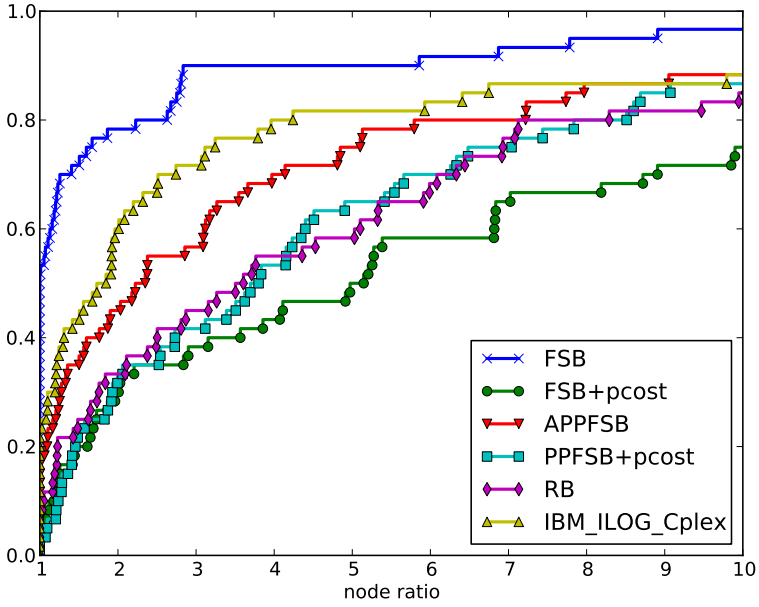


Figure 1: Performance profiles.

[10] 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.

[11] A. Mahajan. *On selecting disjunctions for solving mixed integer programming problems.* PhD thesis, Lehigh University, 2009.

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

[13] J. Pryor and J.W. Chinneck. Faster integer-feasibility in mixed-integer linear programs by branching to force change. *Computers & OR*, 38(8):1143–1152, 2011.

[14] R. Wunderling. Private communication, 2011.