Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

GERALD GAMRATH

Improving strong branching by domain propagation

ZIB-Report 13-47 (Rev. version December 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

Improving strong branching by domain propagation

Gerald Gamrath*

Abstract

One of the essential components of a branch-and-bound based mixed-integer linear programming (MIP) solver is the branching rule. *Strong branching* is a method used by many state-of-the-art branching rules to select the variable to branch on. It precomputes the dual bounds of potential child nodes by solving auxiliary linear programs (LPs) and thereby helps to take good branching decisions that lead to a small search tree. In this paper, we describe how these dual bound predictions can be improved by including *domain propagation* into strong branching. Domain propagation is a technique MIP solvers usually apply at every node of the branch-and-bound tree to tighten the local domains of variables. Computational experiments on standard MIP instances indicate that our improved strong branching method significantly improves the quality of the predictions and causes almost no additional effort. For a full strong branching rule, we are able to obtain substantial reductions of the branch-and-bound tree size as well as the solving time. Moreover, the state-of-the-art hybrid branching rule can be improved this way as well.

This paper extends previous work by the author published in the proceedings of the CPAIOR 2013 [18].

Keywords: mixed-integer programming, branch-and-bound, branching rule, variable selection, strong branching, domain propagation

Mathematics Subject Classification: 90C10, 90C11, 90B40, 9008, 90C57

1 Introduction

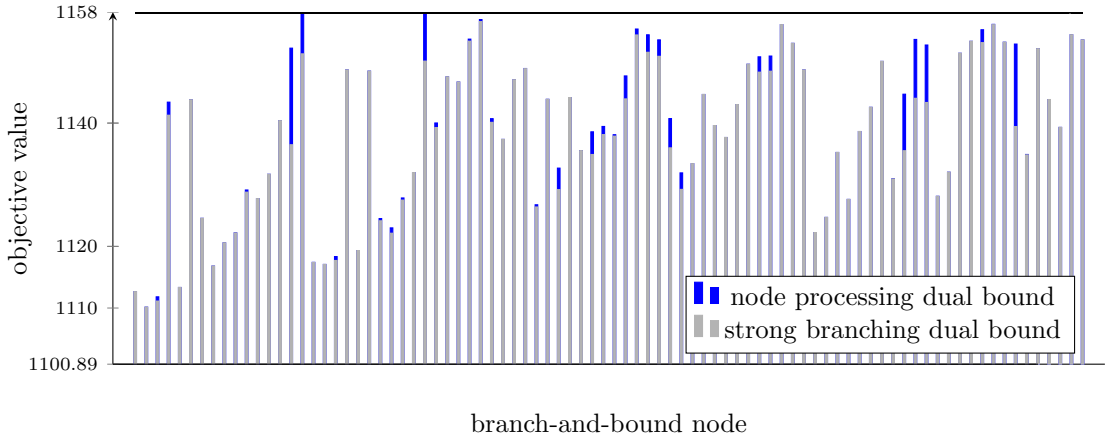
Since the invention of the linear programming (LP) based branch-and-bound method for solving mixed-integer linear programs (MIPs) in the 1960s [24, 14], branching rules have been an important field of research in that context, being one of the core parts of the method (for surveys, see [28, 25, 5]). Their task is to split the current problem into two or more disjoint subproblems if the solution to the LP relaxation of the current problem does not fulfill the integrality restrictions. Thereby, it should exclude the LP solution from all subproblems while keeping at least one optimal solution.

The most common way to split the problem is to branch on trivial inequalities, which split the domain of a single variable into two parts (called *variable branching*). Alternatively, branching can be performed on general linear constraints (see [31, 29, 27, 21, 13]) or can create more than two subproblems, cf. [12, 26]. In case of variable branching, the variable to actually branch on is typically chosen with the goal of improving the local dual bound of both created subproblems (also called *child nodes*). This helps to tighten the global dual bound and prune nodes early. For recent research on alternative criteria, see, e.g., [30, 22, 15, 20].

A very popular branching rule called *pseudo-cost branching* [9] uses history information about the change of the dual bound caused by previous branchings. More accurate, but also more

*Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, gamrath@zib.de

Figure 1: Comparison of the dual bounds computed during strong branching and those obtained later during node processing for instance aflow30a from MIPLIB 2003.



expensive, is *strong branching* [19, 7, 25], which explicitly computes dual bounds of potential child nodes by solving auxiliary LPs with the branching bound change temporarily added. The *full strong branching rule* does this at every node for each integer variable with fractional LP value which empirically leads to very small branch-and-bound trees [1, 5]. Modern branching rules typically combine these two approaches and use strong branching in case of uninitialized or unreliable pseudo-cost values (see [5, 3]). Often, they also impose an iteration limit to restrict the effort for solving the strong branching LPs, see, e.g., [19]. In case this limit is reached, the current (possibly suboptimal) LP value is used as a prediction for the child node's dual bound.[1] An important example for a modern branching rule is *reliability pseudo-cost branching* [5], which considers the history information for a variable unreliable if the number of pseudo-cost updates for this variable is below a given threshold. What this all amounts to is that strong branching is an important component of state-of-the-art branching rules, predicting the bounds of potential children when no other information is available yet.

In practice, however, one can often observe a difference between the dual bound that strong branching computes for a node and the actual dual bound obtained later when this particular node is processed. An example is shown in Figure 1, which depicts the differences for instance aflow30a from MIPLIB 2003. The y-axis ranges from the root node dual bound up to the optimum objective value which was provided at the beginning. The instance was solved with full strong branching after 87 nodes, each of which is represented by one bar. All strong branching LPs were solved to optimality, but still for almost half of the nodes, the bound obtained during node processing is better than the predicted bound, on average, 10% of the gap between predicted and primal bound is closed by the node LP. This restrains the effectiveness of strong branching, since with more accurate predictions, better branching variables could have been selected. There are various reasons for the difference, most prominently *domain propagation* and global domain changes found in the meantime. The task of domain propagation (or *node preprocessing*) is to tighten the local domains of variables by inspecting the constraints and current domains of

[1] Since strong branching LPs are typically solved with the dual simplex method, the LP value often provides a valid dual bound even if the iteration limit is reached. Note, however, that modern simplex implementations use pertubation to deal with degeneracy and need to do some final primal simplex pivots to undo this, in which case the LP value does not necessarily provide a valid dual bound after reaching the limit. Nevertheless, the value is often close to the optimal LP value so that it can be used as a prediction, anyway.

other variables at the local subproblem. It is the essential part of each constraint programming solver [8] and has also proven to improve MIP solvers significantly by tightening the LP relaxation, resulting in better dual bounds and detecting infeasibilities earlier [1, 32, 17].

While strong branching cannot do anything about the difference in the dual bounds caused by global domain changes, it should react upon the continuous improvement in domain propagation techniques. In this paper, we examine how strong branching can be improved by combining it with domain propagation in order to compute better dual bound predictions. This means that we perform the same domain propagation steps that are already performed at each node of the branch-and-bound tree also during strong branching, prior to solving the strong branching LP of a potential child node. The general idea and an evaluation of the direct effects on single strong branching calls are presented in the next section. We implemented the modified strong branching method within the MIP solver itself instead of using external methods of the LP solver. This allows us to apply additional tricks to further improve the strong branching performance which are discussed in Section 3. After that, in Section 4, we provide detailed computational results on a collection of MIPLIB [11, 6, 23] instances. In a setting which focuses on the branching performance of a full strong branching rule, we achieve significant reductions of both number of nodes and solving time, but also with default settings, both the full strong branching rule as well as a hybrid branching method combining strong branching and pseudo-costs can be improved.

2 Strong branching with domain propagation

In the following, we consider mixed-integer linear programs of the form:

$$\min\{c^T x \mid Ax \geq b, x \geq 0, x_i \in \mathbb{Z}\ \forall i \in I\}. \tag{1}$$

A basic implementation of strong branching works as follows: Given the current problem P of form (1) and an integer variable x_i, $i \in I$, with fractional LP solution value $\hat{x}_i$, it computes dual bounds of the two potential child nodes that would be created by branching on x_i. Therefore, it creates two temporary subproblems P_d (the *down child*) and P_u (the *up child*) by adding to P the bound changes $x_i \leq \lfloor \hat{x}_i \rfloor$ and $x_i \geq \lceil \hat{x}_i \rceil$, respectively, and solves their LP relaxations. Strong branching with domain propagation (*SBDP*) improves this by applying domain propagation to tighten the variable domains of P_d and P_u after adding the bound changes. The strong branching dual bound for each of the children is then either set to $+\infty$, if propagation detected infeasibility, or obtained by solving the (tightened) LP relaxation of the respective child node. Computing these two dual bound predictions for a given variable with fractional value will be referred to as a *strong branching call* in the following.

Since the additional domain propagation step can only tighten the LP relaxation, the dual bounds obtained by SBDP are always greater than or equal to the ones computed by standard strong branching as long as we solve the LPs to optimality. This is the case for the full strong branching rule we focus on in this section, later, in Section 4, we will also regard a hybrid branching rule which imposes an iteration limit for the strong branching LPs. The questions we want to consider in the remainder of this paper are the following:

a) How does the strong branching effort change?

b) By how much can SBDP improve the strong branching predictions?

c) What is the effect of SBDP on the overall performance of a MIP solver?

Question a) addresses the time spent for domain propagation as well as the change in the number of LP iterations (and therewith the LP solving time). The simplex warmstart normally allows to

Table 1: Impact of domain propagation on the strong branching calls.

		strong branching			strong branching with propagation				
category	calls	inf	iters	time	domchgs	inf	cl. gap	iters	time
better bound	376.0	–	44.0	19.5	38.9	–	20.7%	57.2	23.1
same bound	23802.0	–	82.3	39.7	23.7	–	–	79.0	40.6
cutoff	3342.6	0.92	56.8	27.3	35.7	1.11	8.5%	46.7	25.6
all	30469.4	0.14	81.0	40.2	26.3	0.17	2.7%	77.5	40.5

solve the strong branching LPs with just a few iterations as there is only one bound tightened, but additional changes performed by domain propagation might change this. On the other hand, infeasibility can already be detected by domain propagation so that we do not need to solve the strong branching LP afterwards. Question b) addresses both the detection of infeasibilities as well as the computed dual bounds of potential child nodes. While answering the first two questions will give us an indication concerning the profitability of SBDP, the final question c) should assess it by taking into account the changes of tree size and solving time caused by SBDP.

For answering these questions, we performed computational experiments using an implementation of SBDP based on the MIP solver SCIP 3.0 [1, 2] with underlying LP solver SoPlex 1.7 [35]. They were performed on a cluster of Intel Xeon E5420 2.5 GHz computers, with 6 MB cache and 16 GB RAM, running Linux (in 64 bit mode). A time limit of two hours per instance was imposed. We use full strong branching to measure the impact of our changes for each candidate variable at each node and concentrate on the branch-and-bound performance by providing the optimal objective value as objective cutoff and disabling primal heuristics and cutting plane separation as well as the components presolver[2] of SCIP. We will refer to this setting as the *sandbox* setting in the following. As test set, we used the MMM test set consisting of all instances from MIPLIB 3.0 [11], MIPLIB 2003 [6], and the benchmark set of MIPLIB 2010 [23]. We excluded all instances for which no significant amount of strong branching was performed—either because the instance was solved in presolving or at the root node prior to branching or because the time limit of two hours was hit before strong branching was performed on at least ten variables. Additionally, we excluded the three infeasible instances from MIPLIB 2010 in order to be able to compute the additional gap closed by SBDP, which left us with a total number of 147 instances. With respect to the other measures, however, the results on the infeasible instances were consistent with those presented here.

The experiments were conducted as follows: After each standard strong branching call, we additionally performed a call of SBDP on the same variable, running the default domain propagation techniques of SCIP (see [1]). We collected statistics about the differences, but did not use any of the information produced by SBDP within the branch-and-bound search. We chose this approach instead of running twice, one time with each variant, to isolate the impact of the new method on each single strong branching call and avoid noise introduced by differences in the branch-and-bound tree created by different branching choices.

For analyzing the impact of SBDP, we divide the strong branching calls into three categories: *Cutoff* if at least one of the two potential child nodes was detected to be infeasible by any of the methods, *better bound* if no infeasibility was detected and SBDP computed a better dual bound for at least one of the potential child nodes, and *same bound* if both strong branching variants computed the same (finite) bounds for both potential child nodes.

The results for each of these categories are presented in one line in Table 1, with an additional line that summarizes these results for all strong branching calls. Besides the number of strong

[2]The components presolver solves small independent subproblems in advance, excluding them from the main branch-and-bound search.

branching calls (column calls), we show for both strong branching variants the number of potential subproblems detected to be infeasible (column inf), the number of LP iterations for solving the LPs of the two subproblems (column iters), and the strong branching time in milliseconds (column time). Furthermore, we present the number of additional domain changes performed by SBDP (column domchgs) and the percentage of the gap between primal bound and strong branching dual bound closed by using SBDP instead of standard strong branching (column cl. gap). Except for the entries in column cl. gap, each of the numbers listed is computed by taking the arithmetic mean over all strong branching calls for a single instance and average over the instances by taking the *shifted geometric mean*.[3] We use a shift of 100 for the number of strong branching calls, 10 for time, iteration number and domain changes, and 1 for the number of child nodes declared infeasible per call. For the closed gap, having only values between 0 and 100, the average over the instances is determined by arithmetic mean.

As expected, the *better bound* case—which happens only rarely—is typically caused by a high number of domain changes during propagation and leads to an increase in both the average sum of LP iterations and time per strong branching call, thereby closing the gap by more than 20% on average. In the most common case, the *same bound* category, a smaller, but still relevant number of domains are changed by propagation. But instead of slowing down the simplex warm start, these bound changes even reduce the average number of LP iterations, e.g., by fixing variables that would otherwise need to be rendered feasible by some simplex pivots. Last, in the *cutoff* case, SBDP detects infeasibility of more potential child nodes—on average 1.11 of the two children regarded per call are declared infeasible compared to 0.92 otherwise. In about 15% of the cases, infeasibility is detected already during propagation, leading to a reduction of the average number of LP iterations and strong branching time. On average over all strong branching calls, SBDP can declare every twelfth instead of nearly every fourteenth strong branching child node infeasible and closes the gap by 2.66%. The average number of LP iterations is slightly decreased, while the time per strong branching call increases marginally. This demonstrates that the domain propagation time is relatively small compared to the total strong branching time; on average, it was less than 5%.

We also repeated our motivating example from Section 1 using SBDP and illustrated the results in Figure 2. The differences between the predicted bounds and the bounds obtained when processing the nodes are much smaller now: the prediction was inaccurate for only 17% rather than 47% of the nodes, the average gap closed by the node LP reduces from 10% to 2.6%. This also leads to solving the instance after 81 instead of 87 branch-and-bound nodes.

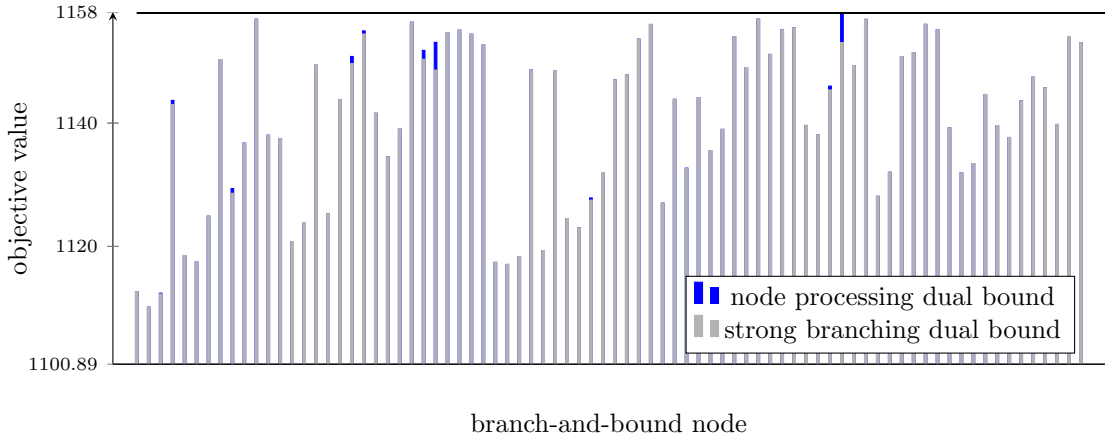
Summing up, the increase in the strong branching effort is negligible—which answers question a)—while also question b) can be answered positively: The strong branching predictions are improved both by detecting infeasibilities more often as well as computing more accurate LP bounds. For the aflow30a instance which served as our motivating example, this helps to reduce the difference between predicted and LP bound during node processing by 74%. In a branch-and-bound search, this should lead to more variable fixings and help us taking better branching decisions. This assumption will be examined further in Section 4, based on computational results showing the impact of SBDP on the performance of SCIP, thereby also answering the remaining question c).

3 Additional strong branching improvements

Our enhanced strong branching method uses the domain propagation methods of the MIP solver which also exploit integrality information. Thus, we implemented it within SCIP itself and do

[3] See Achterberg [1, Appendix A3] for a definition and discussion of the shifted geometric mean.

Figure 2: Comparison of the dual bounds computed by SBDP and those obtained later during node processing for instance aflow30a from MIPLIB 2003.



not use the highly optimized strong branching interface methods provided by many LP solvers such as CPLEX or XPRESS. On the one hand, this might slightly deteriorate the performance of the LP solver for the strong branching LPs; on the other hand, it allows us to tailor the strong branching method to our needs even more. In this section, we present three such improvements. Some of them may be considered common sense, but at least they have not been implemented in the state-of-the-art non-commercial MIP solver SCIP so far. The first two improvements were already discussed in the proceedings paper [18] extended by this article while the third one was newly added.

The first improvement treats the case of an *infeasible strong branching subproblem*, which traditionally leads to simply tightening the domain of the candidate variable at the current node (or cutting off the current node if both subproblems are infeasible). While, e.g., the strong branching interface method of CPLEX always regards both subproblems,[4] we interrupt a strong branching call when the first potential child is found infeasible, saving the effort we would spend for the second child node. As usual, the domain change of the second subproblem is then applied at the current node either directly or after a certain number of domain changes has been collected this way. This causes a reoptimization of the node LP, after which branching is started again, if needed. Any information we would get by solving the second strong branching child, e.g., an improved bound for the current node or a proof for its infeasibility, we get when reoptimizing the node LP, anyway. A simple trick helps to exploit this even further: In our computational experiments presented in Section 2, about 69% of the infeasible subproblems were up children. This is not surprising since problems are often modeled in a way such that changing a variable's lower bound—in particular, fixing a binary variable to one—has more impact than changing its upper bound (fixing a binary variable to zero). Thus, we investigate the potential *up child first* in order to profit from infeasible child nodes more often.

Secondly, we can often identify *valid local bounds* for some variables even if neither of the two potential child nodes is infeasible. Suppose that during the investigation of the two potential child nodes for a candidate variable, the domain of another variable x_i was tightened to $[lb_d, ub_d]$ and $[lb_u, ub_u]$, respectively. Then, without branching on this candidate variable, the domain of x_i at the current node can be tightened to $[\min\{lb_d, lb_u\}, \max\{ub_d, ub_u\}]$. Similar to the bound

[4] Also with other LP solvers that do not provide a strong branching interface method—in particular SoPlex, which we use in our computational experiments—the LP interface implemented in SCIP behaves the same way.

changes deduced from infeasible strong branching subproblems, these domain changes then cause a reoptimization of the node LP, followed by another branching call if needed. This means that we essentially perform *probing preprocessing* [32] locally as a side product of SBDP with negligible additional cost. For 94 of the 147 instances considered in Section 2, this technique was able to identify tighter bounds, identifying on average 3.15 bounds that could have been tightened per strong branching call with both subproblems feasible.

Finally, we exploit the fact that we get *LP solutions during strong branching* instead of just the LP objective value as would be the case when using the external strong branching methods of CPLEX or XPRESS.[5] Because the changed bound of the investigated variable typically forces this variable to an integer value in the strong branching LP solution, we are optimistic that this solution is "more integral" than the node LP solution. Therefore, we check this solution for integrality, hoping to find a new primal feasible solution. Moreover, we even run the fast *simple rounding heuristic* [4] implemented in SCIP on the optimal strong branching LP solution. In the next section, we will answer the question of how often solutions are found this way and how good they are.

4 Computational results

In this section, we present computational experiments illustrating the effect of SBDP on the overall performance of SCIP and will finally answer question c) posed in Section 2. The results significantly extend those presented in [18] and use new features implemented in the meantime, e.g., the third improvement discussed in Section 3, so that we used a new development version of SCIP 3.0.1 [1, 2] (git hash `9e5a0ff`) with underlying LP solver SoPlex 1.7.1 [35] (git hash `82b4e30`). The experiments were performed on a cluster of Intel Xeon E5672 3.2 GHz computers, with 12 MB cache and 48 GB RAM, running Linux (in 64 bit mode). As test set, we used the MMM set as in Section 2, this time without excluding any instances.

In order to increase the reliability of our computational results, we ran each problem in four variants: the original instance and three random permutations of variables and constraints. Although this does not change the instance, it often results in a significantly different solving behavior. One main reason for this is imperfect tie-breaking, leading to different decisions being taken because of small numerical differences caused by floating-point arithmetics. Moreover, computationally expensive methods are often stopped early after a certain number of unsuccessful tries, meaning that the order of variables and constraints—in particular which of them are considered before such a limit is reached—plays a major role. This effect is amplified in MIP solvers, where one choice made differently at the beginning can lead to a completely different branch-and-bound tree to be investigated. Such changes of performance due to seemingly performance-neutral changes in the environment or the input format are denoted *performance variability* [23] and affect all major MIP solvers. As proposed in [23], we use permutations of variables and constraints as a good random generator affecting most instances and almost all components of a MIP solver. For each instance, we take the arithmetic mean of the solving time and number of branch-and-bound nodes over the four regarded permutations (including the original one). An instance is counted as solved only if all four permutations were solved to optimality within the time limit of two hours.

We use a shifted geometric mean with shift 10 and 100 to average over the solving times and node numbers, respectively. Detailed instance-wise results of the computational experiments presented in this section are provided in Table 5 in Appendix A.

[5] Also with other LP solvers like SoPlex, SCIP did not use the strong branching LP solution before.

Table 2: Comparison of full strong branching with and without SBDP, focusing on the branching performance by using the sandbox setting.

		full strong branching			full strong with SBDP		
test set	size	solved	nodes	time	solved	nodes	time
all instances	168	98	1832.7	549.2	105	1376.7	479.4
solved by both	98	98	710.1	80.0	98	534.4	71.2
solved with tree	84	84	1046.1	108.9	84	761.7	95.5

For our first experiment, we used the sandbox setting from Section 2 again: We provided the optimum as cutoff bound and disabled primal heuristics, cutting plane separation, and the components presolver in order to focus on the branch-and-bound search and to decrease performance variability. Moreover, we used full strong branching as a branching rule which completely relies on strong branching and uses no history information. Within the full strong branching rule, we exchanged the strong branching calls for SBDP using all improvements discussed in Section 3. Note that the last improvement has no impact in this experiment because we provide the optimum right at the beginning and disable primal heuristics; its impact will be assessed later.

The results are summarized in Table 2. They are promising: With SBDP, SCIP is able to solve 105 out of the 168 instances of the MMM test set within the time limit of two hours, seven instances more than with standard strong branching. The shifted geometric mean of the solving time is reduced by 13%. For the subset of instances that both versions solved to optimality (row solved by both), the average number of nodes and the solution time are reduced by 25% and 11%, respectively. Since we are testing a branching rule, we are mainly interested in instances of this subset for which at least one of the variants built a branch-and-bound tree and did not solve the instance at the root node already. The results for these instances (row solved with tree), are slightly better: The number of branch-and-bound nodes and the solving time are reduced by 27% and 12% in the shifted geometric mean, respectively. The running times are also illustrated by the performance diagram shown in Figure 3. We see that in this experiment, full strong branching with domain propagation clearly dominated the version without propagation: For every possible

Figure 3: Performance diagram for full strong branching on the MMM test set. The graph indicates the number of instances solved within a certain time when using the sandbox setting to focus on the branching performance.

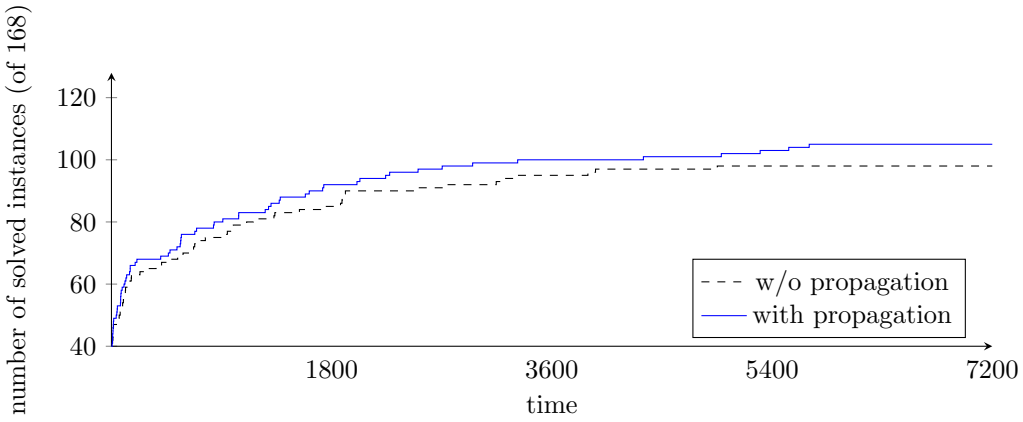


Table 3: Comparison of full strong branching with default settings, with and without SBDP.

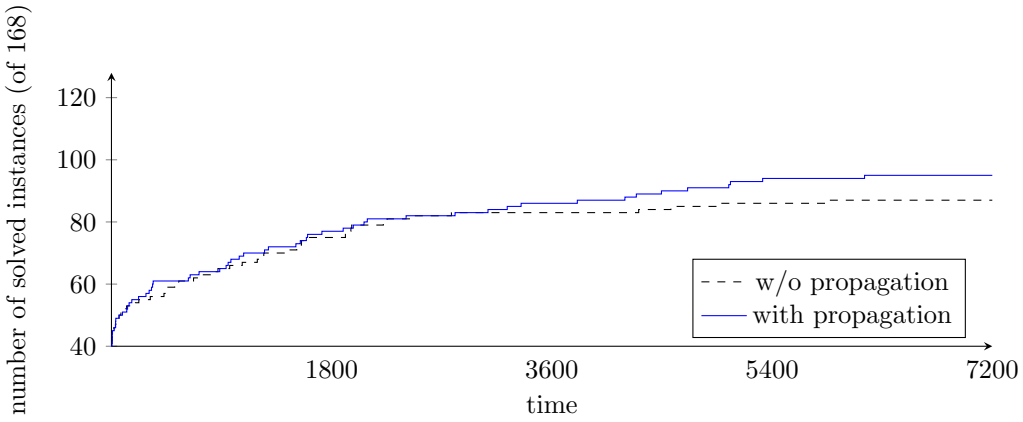
		full strong branching			full strong with SBDP		
test set	size	solved	nodes	time	solved	nodes	time
all instances	168	87	1025.5	710.5	95	909.1	673.7
solved by both	87	87	555.3	74.4	87	487.5	70.3
solved with tree	76	76	759.0	100.6	76	658.0	94.4

time limit between 10 seconds and two hours, it solved more instances to optimality within this time. Finally, let us mention that we also did an experiment with SBDP without the additional improvements to single out their effects. It turned out that they only have a small impact in this setting, causing a change of one percent in running time and number of nodes; the main improvement comes from applying domain propagation.

Our previous experiments give a first answer to question c) posed in Section 2: They indicate that SBDP is quite beneficial in our sandbox environment focusing on the branching performance. But does this also hold for default settings and a state-of-the-art branching rule?

In order to answer this question, we performed additional experiments. In the first one, we still used full strong branching, but—except for the branching rule—the default settings of SCIP, i.e., we did not disable cutting plane separation, heuristics, or the components presolver, neither did we provide the optimum objective value as the cutoff bound. Thus, branching is still based on strong branching only, but we do not focus on proving optimality of a given solution by branching anymore, as we did before; instead, we also investigate how the branching rule interacts with cutting plane separation and performs with respect to finding good solutions during the search. As was to be expected, this results in a smaller impact of SBDP on the overall SCIP performance as can be seen in Table 3. It helps to solve 95 instances to optimality within the time limit, compared to only 87 without SBDP, and leads to a reduction of the shifted geometric mean of the solving time by 5% for the complete MMM testset. For the instances solved to optimality by both variants (row solved by both), the average numer of nodes and the solution time are reduced by 12% and 5.5%, respectively. For the set of instances which were solved after some branching (row solved with tree), the results are slighty better: the number of nodes is reduced by 13%, the solving time by 6%. The performance diagram shown in Figure 4 shows

Figure 4: Performance diagram for full strong branching with SCIP default settings on the MMM test set. The graph indicates the number of instances solved within a certain time.



that full strong branching with domain propagation dominates the version without propagation also with default settings, the number of instances solved after a certain time is never smaller with SBDP than without. However, as the average numbers already indicated, the difference is smaller compared to full strong branching with sandbox settings, because more time is spent for other components like cutting plane separation or primal heuristics which are not improved by SBDP and the optimal solution needs to be found during the search, which is not necessarily promoted by a better branching rule focussing on the dual bound improvement. Nevertheless, we can conclude that also when using full strong branching with default settings, SBDP is able to improve the performance of SCIP.

Finally, let us take the next step and check how SBDP performs when used within *hybrid branching*, the default branching rule of SCIP. Hybrid branching is an extension of reliability pseudo-cost branching which takes into account not only the predicted dual bounds, but also other history information, e.g., the number of implied reductions of other variable domains and the number of (recent) conflicts a variable appears in. For more details on hybrid branching, we refer to [3].

The main criterion for selecting the branching variable, however, is still the predicted child node dual bound, which in the beginning of the search is computed by strong branching. Thus, we expect SBDP to have a positive impact also on hybrid branching. Moreover, also some of the other regarded history information is automatically initialized when running SBDP on a variable. On the other hand, hybrid branching uses a reliability mechanism, i.e., strong branching will be performed only a limited number of times on each variable; later on, only history information is used. This is why we expect the effect of SBDP to be smaller for this branching rule than for full strong branching.

Another important difference with respect to our previous experiments is that hybrid branching imposes a limit on the number of iterations per strong branching LP. Therefore, tightening the LP relaxation might not pay off if this results in reaching the iteration limit where the untightened LP would be solved to optimality. Thus, we did an experiment similar to the one presented in Section 2: Running hybrid branching on each instance of the MMM test set with a time limit of two hours, we additionally performed SBDP after each normal strong branching call and compared how often the LP could be solved to optimality (or infeasibility was detected) by the two variants. It turns out that for most of the instances, the iteration limit was reached more often by SBDP; only for four instances, SBDP performed slightly better than standard strong branching. On average over all the instances where branching was performed, standard strong branching reached the iteration limit for 4.9% of the children while SBDP reached it for 9.8% of them. Although the limit was reached twice as often with SBDP, the majority of strong branching LPs was still solved to optimality, so that we expect only a minor disadvantage for SBDP due to the LP iteration limit.[6]

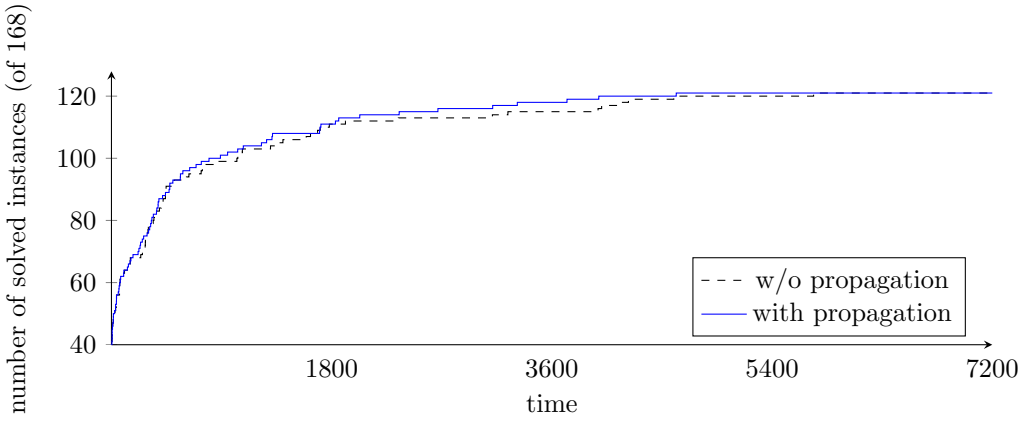
The overall effects of SBDP on the performance of the hybrid branching rule are the subject

[6]We also tried to increase the iteration limit for SBDP, hoping that the benefit caused by domain propagation would outweight the additional LP solving effort, but did not succeed in improving the performance of hybrid branching with SBDP this way.

Table 4: Comparison of hybrid branching with and without SBDP, default settings.

		hybrid branching			hybrid branching with SBDP		
test set	size	solved	nodes	time	solved	nodes	time
all instances	168	121	8626.5	326.1	121	7990.9	313.1
solved by both	118	118	2591.9	83.1	118	2409.5	80.6
solved with tree	107	107	3672.5	103.9	107	3391.6	100.6

Figure 5: Performance diagram for hybrid branching with SCIP default settings on the MMM test set. The graph indicates the number of instances solved within a certain time.



of our final computational experiment. Table 4 illustrates the same type of information as Tables 2 and 3 now for the experiment running hybrid branching with default SCIP settings. The performance improvement is only slightly smaller than for the full strong branching rule with default SCIP settings: The shifted geometric mean of the solving time over all instances is reduced by 4%. This shows that the early branchings, when strong branching is still used and SBDP can improve the branching decision, are the most important ones. The performance diagram illustrated in Figure 5 shows that the improvement is consistent for almost every possible time limit between ten seconds and two hours. The difference in the reduction of the number of nodes is almost halved compared to our previous experiment: The tree size is reduced by 7% on average for those instances solved to optimality and slightly more for those not already solved at the root node. Taking into account that most branching decisions are taken based on history information after some time—and the therefore less important role of strong branching within hybrid branching—these results are consistent to our previous results for full strong branching.

Let us come back to our third improvement and evaluate the impact of running the simple rounding heuristic on the strong branching LP solutions: In our run on the original instances of the MMM test set, we can improve the incumbent this way for 19 instances at least once during search; in total, 53 solutions are found. Moreover, the first primal solution is found this way for eight instances, and five times even the optimum is obtained during strong branching. Similar observations can be made for the permuted instances. As for the full strong branching rule, about 1% of the speedup is caused by the three improvements described in Section 3, so our general idea to include domain propagation into strong branching still has the highest impact.

Finally, we can conclusively answer question c) posed in Section 2: Our experiments indicate that SBDP is also able to slightly improve the performance of the default hybrid branching rule when used within the state-of-the-art non-commercial MIP solver SCIP.

5 Conclusions and outlook

This paper examined improvements of strong branching, one of the main components of most state-of-the-art branching rules for mixed-integer linear programming. The primary improvement is the incorporation of domain propagation into the strong branching method in order to compute more accurate dual bound predictions. Our computational experiments on general MIP instances

show that this comes with negligible cost and reduces the branch-and-bound tree size as well as the solving time. In addition, we presented three further enhancements which improve the strong branching performance.

The effect is most distinctive in a full strong branching rule, where our method might prove particulary useful when the branch-and-bound tree should be kept small, e.g., under tight memory restrictions or for massively parallel MIP solvers (see, e.g., [33, 34]), where reducing the tree size has the added advantage of reducing the message passing overhead. But also in a state-of-the-art branching rule like hybrid branching, the average tree size was reduced and the average solving time was decreased. Therefore, the new method will be included in the next SCIP release and used by default.

For "structured" or more general problem classes like MINLP or CIP [1] for which the LP typically misses more information that can be exploited by domain propagation, we expect an even larger improvement by SBDP. A field for future research is the combination with other recent strong branching improvements such as cloud branching [10] or nonchimerical branching [16].

6 Acknowledgements

The author would like to thank Tobias Achterberg and Michael Winkler for fruitful discussions, and Ambros Gleixner, Timo Berthold, and the anonymous reviewers for constructive criticism and helpful suggestions.

References

[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[2] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] T. Achterberg and T. Berthold. Hybrid branching. In W. J. van Hoeve and J. N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer, May 2009.

[4] T. Achterberg, T. Berthold, and G. Hendel. Rounding and propagation heuristics for mixed integer programming. In D. Klatte, H.-J. Lthi, and K. Schmedders, editors, *Operations Research Proceedings 2011*, Operations Research Proceedings, pages 71–76. Springer Berlin Heidelberg, 2012.

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

[6] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

[7] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, pages 645–656, 1998.

[8] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

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

[10] T. Berthold and D. Salvagnin. Cloud branching. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 28–43. Springer Berlin Heidelberg, 2013.

[11] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, (58):12–15, June 1998.

[12] R. Borndörfer, C. E. Ferreira, and A. Martin. Decomposing matrices into blocks. *SIAM J. Optim.*, 9(1):236 – 269, 1998.

[13] G. Cornuéjols, L. Liberti, and G. Nannicini. Improved strategies for branching on general disjunctions. *Mathematical Programming*, 130:225–247, 2011.

[14] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250–255, 1965.

[15] M. Fischetti and M. Monaci. Backdoor Branching. In O. Günlück and G. J. Woeginger, editors, *Integer Programming and Combinatorial Optimization*, volume 6655 of *Lecture Notes in Computer Science*, pages 183–191. Springer Berlin / Heidelberg, 2011.

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

[17] A. Fügenschuh and A. Martin. Computational integer programming and cutting planes. In K. Aardal, G. L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 69–122. Elsevier, 2005.

[18] G. Gamrath. Improving strong branching by propagation. In C. Gomes and M. Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 347–354. Springer Berlin Heidelberg, 2013.

[19] J.-M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12(1):26–47, 1977.

[20] A. Gilpin and T. Sandholm. Information-theoretic approaches to branching in search. *Discrete Optimization*, 8(2):147–159, 2011.

[21] M. Karamanov and G. Cornuéjols. Branching on general disjunctions. *Mathematical Programming*, 128:403–436, 2011.

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

[23] 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. *Mathematical Programming Computation*, 3(2):103–163, 2011.

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

[25] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed-integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[26] A. Lodi, T. Ralphs, F. Rossi, and S. Smriglio. Interdiction branching. Technical Report OR/09/10, DEIS, Università di Bologna, 2009.

[27] A. Mahajan and T. K. Ralphs. Experiments with branching using general disjunctions. In J. W. Chinneck, B. Kristjansson, and M. J. Saltzman, editors, *Operations Research and Cyber-Infrastructure*, volume 47 of *Operations Research/Computer Science Interfaces Series*, pages 101–118. Springer US, 2009.

[28] G. Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4:155–170, 1973.

[29] J. H. Owen and S. Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear programs. *Computational Optimization and Applications*, 20:159–170, 2001.

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

[31] D. M. Ryan and B. A.Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*, pages 269–280. North Holland, Amsterdam, 1981.

[32] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[33] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP – a parallel extension of SCIP. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 135 – 148, 2012.

[34] Y. Shinano, T. Berthold, S. Heinz, T. Koch, M. Winkler, and T. Achterberg. ParaSCIP – a parallel extension of SCIP. Technical Report ZR 11-10, Zuse Institute Berlin, 2011.

[35] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

A Detailed Computational Results

Table 5 lists detailed results for the computational experiments described in Section 4. For each of the instances of the MMM test set, we present the results for full strong branching with our sandbox settings focusing on the branching performance (column full strong branching – sandbox settings), full strong branching with default settings (column full strong branching – default settings), and hybrid branching with default settings (column hybrid branching – default settings). For each of the three experiments, we compare strong branching with and without propagation. We list the average number of processed branch-and-bound nodes for the four permutations of an instance and the average solving time. If at least one permutation of an

instance reached the time limit, we write the average gap in percent, followed by the number of permutations that reached the time limit (in brackets). According to [23], the gap is defined as

$$\text{gap} = \frac{pb - db}{\inf\{|z|, z \in [db, pb]\}},$$

where pb and db are primal bound and dual bound, respectively. Since the test set contains only minization problems, $pb \geq db$ holds for all instances. If the gap is infinite, we print "inf%". If the results with and without domain propagation differ by more than five percent, we print the dominating value in boldface. Large numbers are abbreviated, "k" and "M" stand for a missing factor of 1 000 and 1 000 000, respectively. At the end of the table, we present the number of solved instances, where an instance is only counted as solved if all four permutations were solved to optimality within the time limit. Furthermore, we print the shifted geometric mean of the number of nodes and the solving time. Additionally, we show these means over two subsets of the instances: The instances that were solved to optimality both with and without domain propagation and the part of the former subset where at least one of the two variants did not solve the problem in the root node already. These subsets are defined individually for each of the three experiments so that for each of them, the impact of SBDP can be evaluated separately.

Table 5. Detailed computational results for the comparision of strong branching with and without propagation, as described and summarized in Section 4.

instance	full strong branching – sandbox settings				full strong branching – default settings				hybrid branching – default settings			
	w/o propagation		with propagation		w/o propagation		with propagation		w/o propagation		with propagation	
	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time
10teams	2	6.5	**2**	**5.1**	283	517.0	**96**	**144.1**	**332**	15.2	402	15.3
30n20b8	164	100.0%(4)	**25**	**1256.8**	1	inf%(4)	10	inf%(4)	**208**	**392.3**	1.9k	1226.2
a1c1s1	19.0k	195.5%(4)	14.9k	192.1%(4)	2.8k	40.1%(4)	2.6k	38.9%(4)	508.5k	21.9%(4)	532.5k	22.9%(4)
acc-tight5	1	0.5	1	0.5	**53**	**2779.0**	75	3235.2	1.6k	239.5	**1.5k**	**224.5**
aflow30a	4.2k	71.4	**2.3k**	**48.4**	211	**19.9**	**174**	22.6	2.3k	10.7	**1.4k**	10.9
aflow40b	51.4k	5.8%(4)	49.4k	**0.5%(1)**	9.8k	2.2%(2)	7.7k	**1.1%(1)**	192.2k	1300.1	184.6k	1312.1
air03	**1**	**0.8**	2	0.9	1	1.4	1	1.4	1	1.4	1	**1.3**
air04	9	40.7	9	38.9	58	2446.8	**22**	**1283.1**	182	68.2	**157**	67.7
air05	9	115.3	**9**	118.2	116	1192.2	**39**	**628.7**	329	41.3	**281**	41.1
app1-2	39	2472.7	**13**	**1616.7**	60	2252.5	**16**	**1603.2**	**181**	**1070.2**	1.3k	2029.7
arki001	115.6k	0.0%(4)	127.8k	0.0%(4)	33.4k	0.0%(4)	26.8k	**0.0%(4)**	499.0k	**0.0%(4)**	634.5k	0.0%(4)
ash608gpia-3col	1	inf%(4)	1	inf%(4)	8	870.4	**3**	**308.1**	10	66.0	**7**	**55.5**
atlanta-ip	3	9.9%(4)	3	9.9%(4)	5	inf%(4)	5	inf%(4)	10.2k	**7.1%(4)**	7.2k	10.4%(4)
beasleyC3	27.9k	178.3%(4)	19.8k	**128.9%(4)**	4.2k	17.8%(4)	4.0k	17.8%(4)	1.3M	14.0%(4)	1.4M	13.6%(4)
bell3a	13.7k	3.2	14.4k	**3.0**	21.5k	4.9	21.6k	4.8	23.8k	5.2	23.3k	5.2
bell5	954.4k	281.1	**18.5k**	**9.9**	1.1k	0.7	**980**	**0.6**	1.2k	0.7	**1.1k**	0.7
bab5	215	7.8%(4)	258	7.6%(4)	161	10.4%(4)	141	10.2%(4)	26.4k	**1.4%(4)**	32.0k	1.8%(4)
biella1	10	409.7	11	402.5	34	8.6%(4)	31	8.6%(4)	**5.5k**	**1368.7**	7.0k	1857.6
bienst2	21.3k	947.4	20.9k	**839.4**	23.1k	1069.0	22.6k	**937.2**	**99.9k**	**341.5**	113.3k	370.9
binkar10_1	432.6k	0.4%(4)	317.7k	0.4%(4)	33.7k	1552.3	**26.3k**	1543.7	**204.3k**	**265.0**	215.8k	291.3
blend2	107	0.5	110	0.5	**130**	**0.8**	155	0.9	**205**	**0.7**	218	0.8
bley_xl1	14	2.5%(1)	1	2.5%(1)	5	2.6%(1)	3	2.6%(1)	6	324.4	**5**	326.0
bnatt350	1	0.5	1	0.5	531	inf%(4)	294	inf%(4)	11.2k	866.8	**5.1k**	**476.8**
cap6000	619	**0.8**	627	0.9	1.1k	**2.4**	1.1k	2.7	3.3k	2.5	**2.6k**	2.4
core2536-691	1	9.4	1	9.4	26	1.3%(4)	23	**0.9%(4)**	**418**	**422.4**	558	562.9
cov1075	5.9k	12.3%(4)	5.8k	12.3%(4)	7.2k	11.8%(4)	6.7k	11.8%(4)	1.6M	7.6%(4)	1.6M	7.6%(4)
csched010	25.8k	10.4%(4)	24.4k	10.6%(4)	6.6k	16.0%(4)	6.2k	**13.6%(4)**	907.4k	**0.6%(1)**	1.0M	2.5%(2)
dano3mip	30	19.3%(4)	29	19.3%(4)	9	22.0%(4)	9	22.0%(4)	2.3k	22.0%(4)	2.2k	22.0%(4)
danoint	31.8k	3.3%(4)	33.1k	3.3%(4)	25.0k	3.5%(4)	25.3k	3.6%(4)	**867.9k**	**4226.1**	934.5k	4616.6
dcmulti	293	**1.0**	290	1.2	145	3.3	150	3.4	**180**	**2.0**	201	2.3
dfn-gwin-UUM	71.2k	**671.2**	**67.6k**	834.1	7.0k	445.2	**4.4k**	**336.1**	67.3k	147.0	**47.9k**	**99.9**
disctom	1	2.9	1	2.9	1	6.0	1	6.1	1	6.0	1	6.0
ds	1	63.2%(4)	1	63.2%(4)	1	1.8k%(4)	1	1.8k%(4)	545	**453.8%(4)**	569	491.9%(4)
dsbmip	1	0.5	1	0.5	28	3.0	27	2.9	27	2.4	**21**	**2.2**
egout	35	0.5	**16**	0.5	1	0.5	1	0.5	1	0.5	1	0.5
eil33-2	**279**	163.4	314	**75.1**	454	550.0	437	**221.7**	10.5k	71.9	**1.2k**	**63.0**
eilB101	164	974.1	164	**681.4**	61	4309.6	**43**	**2808.8**	19.1k	443.0	**12.1k**	**387.7**
enigma	1	0.5	1	0.5	321	0.8	**185**	**0.7**	**945**	**0.6**	1.6k	0.8
enlight13	947.2k	42.0%(4)	**5.2k**	**73.5**	566.6k	98.6%(4)	199.6k	**82.1%(4)**	17.7M	44.2%(4)	**1.2M**	**477.2**
enlight14	909.3k	inf%(4)	**8.5k**	**153.3**	520.2k	inf%(4)	184.6k	inf%(4)	17.6M	inf%(4)	**5.1M**	**2351.8**
ex9	1	0.5	1	0.5	1	34.1	1	34.0	1	34.1	1	34.1
fast0507	26	**1912.4**	**25**	2241.1	92	1.4%(3)	**59**	**5045.8**	920	358.5	**830**	**252.1**
fiber	1.2k	**15.4**	**1.0k**	18.5	13	**1.3**	**12**	1.4	**17**	1.4	24	**1.3**
fixnet6	81	0.9	79	0.9	13	**2.3**	**11**	2.6	17	2.5	**15**	2.5

cont'd next page

instance	full strong branching – sandbox settings				full strong branching – default settings				hybrid branching – default settings			
	w/o propagation		with propagation		w/o propagation		with propagation		w/o propagation		with propagation	
	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time
flugpl	340	0.5	**61**	0.5	474	0.5	**51**	0.5	424	0.5	**113**	0.5
gen	**22**	0.5	26	0.5	1	0.5	1	0.5	1	0.5	1	0.5
gesa2-o	8.1k	151.5	**5.4k**	153.4	**5**	1.4	8	1.4	7	1.4	6	**1.3**
gesa2	7.8k	100.9	**5.0k**	102.7	**4**	1.2	5	1.2	5	1.2	**5**	1.2
gesa3	58	1.9	**36**	**1.8**	7	1.7	7	1.7	**8**	**1.7**	9	1.8
gesa3_o	56	2.6	**37**	2.6	9	**1.5**	9	1.7	9	**1.7**	10	1.8
glass4	77.2k	541.6	**564**	**13.2**	316.5k	100.0%(4)	168.8k	98.8%(4)	2.7M	65.9%(3)	5.7M	**18.2%(1)**
gmu-35-40	1.6M	0.0%(1)	**1.2M**	**5703.5**	1.3M	**0.0%(4)**	792.6k	0.0%(4)	12.7M	**0.0%(4)**	12.4M	0.0%(4)
gt2	2	0.5	2	0.5	1	0.5	1	0.5	1	0.5	1	0.5
harp2	158.1k	693.1	**95.1k**	**568.1**	285.5k	1233.5	**173.5k**	**1080.6**	15.8M	5739.5	**11.6M**	**3983.5**
iis-100-0-cov	2.9k	**23.7%(4)**	3.2k	25.8%(4)	3.2k	**22.6%(4)**	3.5k	25.3%(4)	100.2k	1780.9	103.2k	1830.9
iis-bupa-cov	737	22.1%(4)	752	21.6%(4)	832	26.0%(4)	842	**22.2%(4)**	184.9k	**4.4%(3)**	185.6k	6.7%(4)
iis-pima-cov	620	12.2%(4)	610	11.8%(4)	785	**12.0%(4)**	759	13.3%(4)	14.0k	1069.0	**11.2k**	**949.4**
khb05250	421	**2.0**	413	2.8	4	0.5	4	0.5	5	0.5	5	0.5
lectsched-4-obj	1	0.5	1	0.5	48	**2.2k%(4)**	42	2.5k%(4)	**62.2k**	**743.0**	82.9k	1268.1
liu	18.4k	102.1%(4)	12.7k	102.1%(4)	6.3k	272.9%(4)	5.2k	260.2%(4)	1.6M	141.0%(4)	1.8M	**128.6%(4)**
l152lav	16	1.4	16	1.4	51	5.1	**27**	**3.4**	43	2.7	**29**	2.7
lseu	765	**0.7**	**475**	0.8	145	0.6	**94**	0.6	659	0.6	**400**	**0.5**
m100n500k4r1	1	0.5	1	0.5	42.3k	4.2%(4)	41.2k	4.2%(4)	8.0M	4.2%(4)	7.7M	4.2%(4)
macrophage	2.7k	378.7%(4)	2.3k	390.5%(4)	1.3k	48.6%(4)	1.1k	48.2%(4)	746.9k	29.0%(4)	1.0M	29.2%(4)
manna81	3.8k	1.0%(4)	3.0k	1.0%(4)	1	0.8	1	0.8	1	0.9	1	**0.8**
map18	148	3318.2	143	3320.4	120	1912.3	119	1894.2	408	**446.7**	**362**	502.0
map20	148	2711.4	**136**	2703.5	179	1963.9	**169**	1976.7	421	413.4	**382**	**381.0**
markshare1	14.0M	inf%(4)	8.6M	inf%(4)	12.3M	inf%(4)	8.6M	inf%(4)	84.6M	inf%(4)	84.5M	inf%(4)
markshare2	10.6M	inf%(4)	6.6M	inf%(4)	10.6M	inf%(4)	7.0M	inf%(4)	72.0M	inf%(4)	69.5M	inf%(4)
mas74	544.8k	**996.1**	523.4k	1304.4	647.2k	**1515.1**	658.6k	2064.4	3.5M	633.1	**3.3M**	**583.6**
mas76	75.6k	**91.4**	72.3k	124.2	85.5k	**124.1**	81.9k	166.3	464.9k	67.2	**422.7k**	**60.6**
mcsched	357	7.4%(4)	453	7.0%(4)	329	7.6%(4)	551	**7.1%(4)**	20.8k	275.7	**12.7k**	**176.0**
mik-250-1-100-1	1.8M	**3.3%(4)**	1.2M	4.1%(4)	285.9k	**1957.3**	276.2k	3077.8	1.6M	407.8	**1.6M**	417.6
mine-166-5	251	16.7	**171**	16.1	1.4k	**114.5**	**1.3k**	125.4	**2.5k**	39.2	3.1k	41.2
mine-90-10	56.9k	1335.6	**21.0k**	**1040.3**	237.9k	0.2%(4)	**112.7k**	**4709.5**	**135.8k**	**513.0**	168.0k	797.4
misc03	103	**1.0**	104	1.1	109	**1.4**	**99**	1.5	208	1.2	**106**	1.2
misc06	12	0.5	12	0.5	6	0.7	6	0.7	6	0.8	6	**0.7**
misc07	2.1k	45.7	2.1k	46.8	2.8k	62.2	2.7k	62.1	26.0k	16.1	**23.9k**	15.7
mitre	1	5.5	1	5.7	1	5.9	1	5.8	1	5.8	1	5.8
mkc	295.3k	1.4%(4)	164.9k	1.4%(4)	17.7k	3.0%(4)	16.2k	**2.0%(4)**	3.0M	**1.5%(4)**	2.2M	1.6%(4)
mod008	363	**0.5**	347	0.6	84	0.9	86	**0.8**	336	0.9	**269**	0.9
mod010	5	0.5	5	0.5	2	0.8	**1**	**0.6**	2	0.8	**1**	**0.6**
mod011	2.9k	585.0	**2.7k**	**535.6**	179	731.2	179	716.1	979	153.6	**882**	**138.6**
modglob	1.1M	**3894.8**	1.1M	5302.6	138	**1.9**	139	2.3	602	1.2	577	1.2
momentum1	339	**16.6%(4)**	315	18.6%(4)	315	137.1%(4)	309	**98.4%(4)**	51.4k	**13.3%(4)**	28.0k	87.5%(4)
momentum2	190	13.9%(4)	**144**	**4985.0**	98	inf%(4)	84	inf%(4)	90.5k	0.4%(2)	62.5k	6.4%(1)
momentum3	1	150.8%(4)	1	150.8%(4)	1	258.3%(4)	1	258.3%(4)	118	258.2%(4)	152	258.3%(4)
msc98-ip	55	0.7%(4)	48	0.7%(4)	11	inf%(4)	10	inf%(4)	3.3k	37.8%(4)	4.1k	**25.4%(4)**
mspp16	29	3204.0	**27**	**2952.3**	37	4967.5	**33**	**4495.8**	62	3977.4	**52**	**3116.2**
mzzv11	38	4952.0	**36**	**4348.2**	62	**11.0%(4)**	159	13.2%(4)	3.3k	277.7	**2.3k**	**263.0**

cont'd next page

instance	full strong branching – sandbox settings				full strong branching – default settings				hybrid branching – default settings			
	w/o propagation		with propagation		w/o propagation		with propagation		w/o propagation		with propagation	
	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time
mzzv42z	5	1904.2	**3**	**1726.3**	128	**13.4%(4)**	95	14.1%(4)	1.3k	253.1	**1.1k**	**233.8**
n3div36	2.8k	13.6%(4)	2.2k	13.8%(4)	5.2k	17.0%(4)	4.1k	**16.0%(4)**	282.0k	9.5%(4)	263.0k	10.0%(4)
n3seq24	40	0.4%(4)	16	0.4%(4)	1	61.9%(4)	1	61.9%(4)	1.2k	**3.6%(4)**	1.9k	8.4%(4)
n4-3	54.6k	40.9%(4)	41.1k	43.1%(4)	5.0k	5861.9	4.8k	6155.9	48.3k	772.6	**45.1k**	734.5
neos-1109824	7.4k	**1160.2**	**6.9k**	1284.1	808	**317.4**	803	340.9	19.8k	154.2	19.1k	159.2
neos-1337307	9.8k	0.5%(4)	2.3k	**0.5%(4)**	74	inf%(4)	42	inf%(4)	339.8k	0.0%(4)	369.6k	**0.0%(4)**
neos-1396125	2.7k	3145.2	**1.3k**	**1584.9**	6.4k	6.6%(2)	**3.8k**	**4197.1**	**71.7k**	**1912.8**	77.7k	2668.5
neos13	6	**478.2**	6	571.7	48	62.0%(4)	81	**55.9%(4)**	24.3k	33.1%(4)	24.8k	**8.6%(2)**
neos-1601936	1	651.1	1	**565.9**	17	inf%(4)	19	inf%(4)	9.8k	**8.3%(1)**	21.4k	25.0%(2)
neos18	14.2k	1872.4	**2.0k**	**561.9**	24.0k	1.6%(1)	**12.4k**	**2408.5**	8.1k	41.2	**6.2k**	**33.6**
neos-476283	22	**97.5**	**20**	110.5	**321**	**671.0**	406	976.3	**483**	**285.5**	686	330.7
neos-686190	268	148.4	**242**	147.8	2.2k	834.8	**1.6k**	**643.2**	8.7k	106.8	**8.2k**	101.5
neos-849702	1	0.5	1	0.5	547	inf%(3)	507	inf%(2)	91.3k	1627.1	**27.1k**	**471.0**
neos-916792	81.7k	11.7%(4)	57.3k	11.9%(4)	75.8k	12.8%(4)	53.8k	13.0%(4)	110.5k	441.2	**95.9k**	**368.1**
neos-934278	2	0.2%(4)	1	0.2%(4)	2	328.2%(4)	2	328.2%(4)	6.5k	**1.6%(4)**	5.1k	3.3%(4)
net12	57	**109.0%(4)**	44	136.0%(4)	71	inf%(4)	34	inf%(4)	**6.7k**	**4006.8**	7.3k	7.7%(1)
netdiversion	1	4.9%(4)	1	4.9%(4)	1	2.1M%(4)	1	2.1M%(4)	115	**1.0M%(3)**	145	1.6M%(4)
newdano	173.4k	14.1%(4)	181.3k	**12.7%(4)**	118.7k	22.5%(4)	127.8k	**20.7%(4)**	2.7M	**1.0%(1)**	2.7M	1.2%(1)
noswot	431.2k	918.8	**99.3k**	**478.5**	560.5k	1927.4	**257.7k**	**1720.3**	1.4M	296.4	**1.2M**	**237.8**
ns1208400	1	233.1	1	**208.0**	117	inf%(4)	138	inf%(4)	11.5k	1592.1	**3.4k**	**564.0**
ns1688347	183	98.6	**76**	**43.8**	131	23.7%(3)	253	**3808.2**	5.6k	499.5	**3.7k**	**381.9**
ns1758913	1	2.2%(4)	1	**1.1%(2)**	1	514.3%(4)	1	516.3%(4)	44	374.8%(4)	78	381.6%(4)
ns1766074	248.3k	**364.9**	**218.3k**	910.3	242.2k	**455.2**	**219.1k**	953.9	930.5k	733.3	941.8k	**691.8**
ns1830653	1.7k	3956.8	**1.3k**	**2506.3**	3.0k	22.4%(3)	**2.8k**	**5059.8**	**46.8k**	612.9	56.3k	639.4
nsrand-ipx	39.1k	1.5%(4)	27.4k	1.5%(4)	24.2k	5.3%(4)	17.9k	5.2%(4)	1.6M	3.8%(4)	1.7M	3.7%(4)
nw04	12	12.5	12	12.3	5	35.2	5	**32.4**	5	28.9	5	**26.0**
opm2-z7-s2	62	1104.7	**58**	**1039.8**	2	232.8%(4)	2	232.8%(4)	4.8k	1025.4	**3.8k**	**890.7**
opt1217	1.5M	18.8%(4)	938.6k	18.8%(4)	1	0.7	1	**0.6**	1	0.7	1	0.7
p0033	31	0.5	**7**	0.5	1	0.5	1	0.5	1	0.5	1	0.5
p0201	61	**1.3**	**57**	1.7	50	2.0	**44**	2.1	59	1.5	**44**	1.5
p0282	14	0.5	**10**	0.5	3	0.5	3	0.5	3	0.6	**3**	**0.5**
p0548	138	0.5	**18**	0.5	**5**	0.5	10	0.5	5	0.5	5	0.5
p2756	592	**5.1**	**436**	5.8	79	**1.2**	**32**	1.3	81	1.3	**34**	1.3
pg5_34	56.9k	12.2%(4)	53.4k	12.3%(4)	39.5k	2222.9	39.2k	**2091.3**	317.5k	1401.6	**210.6k**	**1077.7**
pigeon-10	1.9M	11.1%(4)	1.2M	11.1%(4)	2.0M	11.1%(4)	1.8M	11.1%(4)	15.5M	11.1%(4)	15.9M	11.1%(4)
pk1	55.8k	**162.0**	**51.6k**	193.8	**57.9k**	**196.0**	69.6k	282.1	**341.4k**	**70.3**	359.4k	75.3
pp08a	2.1M	**24.5%(4)**	1.3M	27.5%(4)	199	**3.1**	192	3.6	541	1.7	525	**1.6**
pp08aCUTS	249.2k	**1709.6**	238.1k	2030.8	169	2.9	**136**	2.8	367	1.6	**326**	1.6
protfold	10	34.2%(4)	12	34.2%(4)	14	inf%(4)	22	inf%(4)	12.4k	inf%(4)	5.4k	**62.1%(4)**
pw-myciel4	2.3k	150.0%(4)	2.3k	**88.8%(4)**	1.8k	150.0%(4)	1.6k	**135.3%(4)**	**626.5k**	**4596.9**	1.1M	21.2%(2)
qiu	13.7k	1536.5	**11.9k**	**1379.7**	7.8k	1245.2	7.7k	1251.2	10.7k	**67.0**	11.2k	72.5
qnet1	7	**1.1**	7	1.2	**25**	**7.4**	27	8.6	33	4.9	32	4.9
qnet1_o	23	0.9	23	0.9	13	**4.9**	13	6.4	**13**	**3.0**	19	3.9
rail507	26	**1886.4**	**24**	2274.0	85	1.1%(4)	**63**	**5322.9**	**806**	**277.0**	1.0k	303.6
ran16x16	1.1M	5.6%(4)	900.4k	**5.2%(4)**	24.6k	966.6	**19.6k**	**885.1**	414.3k	351.9	405.5k	342.5
reblock67	20.9k	767.6	**8.0k**	**465.2**	27.8k	1597.7	**12.4k**	**1044.4**	130.8k	301.7	**94.1k**	**217.7**

cont'd next page

instance	full strong branching – sandbox settings				full strong branching – default settings				hybrid branching – default settings			
	w/o propagation		with propagation		w/o propagation		with propagation		w/o propagation		with propagation	
	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time	nodes	time
rd-rplusc-21	107	165.3k%(4)	151	165.3k%(4)	920	165.3k%(4)	1.1k	166.5k%(4)	55.6k	inf%(4)	41.2k	inf%(4)
rentacar	2	**1.3**	2	1.4	4	2.6	4	2.6	2	2.4	2	2.4
rgn	249	**0.7**	252	0.8	5	0.5	**1**	0.5	5	0.5	**1**	0.5
rmatr100-p10	93	680.5	**87**	695.8	244	1553.4	**174**	1593.2	907	**143.6**	870	152.2
rmatr100-p5	33	1328.0	33	1371.4	33	1453.3	33	1507.5	**381**	298.3	421	312.5
rmine6	38.7k	**1884.5**	37.6k	2006.8	74.9k	4572.9	**64.3k**	**4290.5**	**951.6k**	**3240.6**	1.5M	0.0%(2)
rocII-4-11	2.6k	24.3%(4)	**483**	**1735.4**	2.1k	50.8%(4)	**1.1k**	**3348.5**	27.2k	347.2	**20.0k**	**320.5**
rococoC10-001000	9.5k	7.2%(4)	11.0k	**6.2%(4)**	10.6k	6.5%(4)	8.5k	**5.3%(4)**	739.0k	4163.0	**662.7k**	**3723.7**
roll3000	10.2k	8.8%(4)	8.9k	8.9%(4)	9.3k	1.9%(4)	9.9k	**1.4%(4)**	1.9M	0.3%(2)	1.8M	0.3%(1)
rout	3.0k	113.9	**1.7k**	**79.8**	5.1k	223.8	**2.5k**	**131.9**	32.4k	41.5	31.3k	40.3
satellites1-25	2	300.0%(4)	38	**5536.2**	1	inf%(4)	1	inf%(4)	26.0k	119.9%(2)	**6.9k**	**1704.5**
set1ch	178.5k	33.8%(4)	129.4k	32.2%(4)	11	**0.9**	11	1.0	13	**1.0**	13	1.1
seymour	197	**4.0%(4)**	209	4.3%(4)	192	6.8%(4)	196	6.9%(4)	129.5k	2.3%(4)	120.8k	2.3%(4)
sp97ar	5.2k	1.2%(4)	5.4k	1.2%(4)	1.4k	9.0%(4)	3.2k	9.3%(4)	10.0k	**7.1%(4)**	5.4k	8.5%(4)
sp98ic	2.3k	0.1%(1)	2.8k	0.1%(1)	3.8k	3.1%(4)	2.7k	**2.5%(4)**	179.2k	2.0%(4)	151.0k	1.9%(4)
sp98ir	110	75.8	**97**	77.3	689	429.3	**418**	**327.5**	**6.2k**	**94.9**	9.1k	130.5
stein27	872	**1.6**	871	2.2	811	**2.0**	803	2.6	4.3k	1.1	**4.0k**	1.1
stein45	7.8k	**63.2**	7.8k	75.4	8.0k	**77.4**	7.9k	90.0	52.7k	14.9	51.3k	14.5
stp3d	1	2.5%(4)	1	2.5%(4)	1	inf%(4)	1	inf%(4)	1	inf%(4)	1	inf%(4)
swath	44.9k	34.5%(4)	43.9k	34.1%(4)	41.5k	22.5%(4)	39.6k	22.0%(4)	1.2M	19.4%(4)	1.0M	18.5%(4)
t1717	7	25.3%(4)	8	25.3%(4)	6	180.5%(4)	6	180.5%(4)	2.7k	41.3%(4)	2.7k	41.5%(4)
tanglegram1	2	77.5%(4)	2	77.5%(4)	2	167.8%(4)	2	167.8%(4)	31	1033.2	**29**	1032.0
tanglegram2	1	2.0	1	2.0	2	35.1	2	35.4	3	8.8	3	8.7
timtab1	744.1k	37.9%(4)	470.3k	**35.1%(4)**	267.3k	6.3%(2)	141.4k	**5.6%(2)**	1.1M	445.6	1.0M	441.5
timtab2	163.0k	122.5%(4)	121.1k	**111.8%(4)**	49.5k	121.1%(4)	63.5k	**93.1%(4)**	10.0M	49.9%(4)	10.1M	47.9%(4)
tr12-30	45.1k	333.3%(4)	31.4k	332.6%(4)	81.0k	**0.0%(1)**	73.0k	0.0%(2)	1.3M	1684.9	1.3M	1700.9
triptim1	1	87.9	1	87.6	1	inf%(4)	1	inf%(4)	65	1706.5	**32**	**1316.5**
unitcal_7	360	0.6%(4)	279	**0.4%(4)**	329	0.9%(4)	272	**0.6%(4)**	36.4k	2344.9	**25.5k**	**1711.9**
vpm1	10.9k	11.8	**4.1k**	**8.0**	1	0.5	1	0.5	1	0.5	1	0.5
vpm2	7.6k	15.4	**4.4k**	15.0	222	2.3	**167**	**2.1**	449	1.2	**400**	1.2
vpphard	8	inf%(4)	10	inf%(4)	6	inf%(4)	7	inf%(4)	18.9k	inf%(4)	19.5k	inf%(4)
zib54-UUE	32.2k	39.2%(4)	38.1k	**31.0%(4)**	10.3k	18.3%(4)	10.6k	**15.0%(4)**	**384.3k**	**3113.9**	413.8k	3317.6
solved (of 168)		98		105		87		95		121		121
sh. geom. mean	1832.7	549.2	1376.7	479.4	1025.5	710.5	909.1	673.7	8626.5	326.1	7990.9	313.1
solved by both		98		98		87		87		118		118
sh. geom. mean	710.1	80.0	534.4	71.2	555.3	74.4	487.5	70.3	2591.9	83.1	2409.5	80.6
solved with tree		84		84		76		76		107		107
sh. geom. mean	1046.1	108.9	761.7	95.5	759.0	100.6	658.0	94.4	3672.5	103.9	3391.6	100.6