Timo Berthold[1], Gerald Gamrath[2], and
Domenico Salvagnin[3][4]

Exploiting Dual Degeneracy in Branching

[1]Fair Isaac Germany GmbH, c/o ZIB, Germany
[2] 0000-0001-6141-5937
[3] 0000-0002-0232-2244
[4]DEI, Via Gradenigo, 6/B, 35131 Padova, Italy

ZIB Report 19-17 (April 2019)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

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

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

Exploiting Dual Degeneracy in Branching

Timo Berthold[*], Gerald Gamrath[†], and Domenico Salvagnin[‡]

Abstract

Branch-and-bound methods for mixed-integer programming (MIP) are traditionally based on solving a linear programming (LP) relaxation and branching on a variable which takes a fractional value in the (single) computed relaxation optimum. In this paper, we study branching strategies for mixed-integer programs that exploit the knowledge of *multiple* alternative optimal solutions (a *cloud*) of the current LP relaxation. These strategies naturally extend common methods like most infeasible branching, strong branching, pseudocost branching, and their hybrids, but we also propose a novel branching rule called cloud diameter branching.

We show that dual degeneracy, a requirement for alternative LP optima, is present for many instances from common MIP test sets. Computational experiments show significant improvements in the quality of branching decisions as well as reduced branching effort when using our modifications of existing branching rules. We discuss different ways to generate a cloud of solutions and present extensive computational results showing that through a careful implementation, cloud modifications can speed up full strong branching by more than 10 % on standard test sets. Additionally, by exploiting degeneracy, we are also able to improve the state-of-the-art hybrid branching rule and reduce the solving time on affected instances by almost 20 % on average.

Keywords: mixed integer programming, branching rule, search strategy, dual degeneracy

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

1 Introduction

In this paper, we address branching strategies for the exact solution of a generic mixed-integer program (MIP) of the form

$$\min\{cx : Ax = b \quad \ell \leq x \leq u \quad x_j \in \mathbb{Z} \quad \forall j \in J\}$$

where $x \in \mathbb{R}^n$, $\ell \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{\infty\})^n$ and $J \subseteq N = \{1, \ldots, n\}$. Every MIP can be written down in this form; in particular, inequalities can be transferred into this form by introducing a slack variable. In the following, x may consist of both structural variables as well as artificial slack variables introduced to obtain this form.

Good branching strategies are crucial for any branch-and-bound based MIP solver. Unsurprisingly, the topic has been the subject of constant and active research since the very beginning of computational mixed-integer programming, see, e.g., [1]. We refer to [2, 3, 4] for some comprehensive studies on branching strategies.

[*]Fair Isaac Germany GmbH, c/o ZIB, Takustr. 7, 14195 Berlin, Germany, timoberthold@fico.com
[†]Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, gamrath@zib.de
[‡]DEI, Via Gradenigo, 6/B, 35131 Padova, Italy, salvagni@dei.unipd.it

In mixed-integer programming, the most common methodology for branching is to split the domain of a single variable into two disjoint intervals. In this paper, we will address the key problem of how to select such a variable. Let $x^\star$ be an optimal solution of the linear programming (LP) relaxation at the current node of the branch-and-bound tree and let $F(x^\star) = \{j \in J : x^\star_j \notin \mathbb{Z}\}$ denote the set of fractional integer variables. A general scheme for branching strategies consists in computing a score s_j for each candidate variable $j \in F(x^\star)$ and then picking the variable with maximum score. Different branching rules then correspond to different ways of computing this score.

Several branching criteria have been studied in the literature. The simplest one, *most infeasible branching*, is to branch on the variable whose fractional part is as close as possible to 0.5; in practice, however, this performs almost as bad as selecting the branching variable randomly, see [3]. A more sophisticated branching strategy is *pseudocost branching* [1], which consists in keeping a history of how much the *dual bound* (the LP relaxation) improved when branching on a given variable in previous nodes, and then using these statistics to estimate how the dual bound will improve when branching on that variable at the current node. Pseudocost branching is computationally cheap since no additional LPs need to be solved and performs reasonably well in practice. However, at the very beginning, when the most crucial branching decisions are taken, there is no reliable historical information to build upon.

Another effective branching rule is *strong branching* [5, 6]. The basic idea consists in simulating branching on the variables in F and then choosing the actual branching variable as the one that gives the best progress in the dual bound. This greedy local method performs very well with respect to the number of nodes of the resulting branch-and-bound tree, but introduces quite a large overhead in terms of computation time, since $2 \cdot |F(x^\star)|$ auxiliary LPs need to be solved at every node. Many techniques have been studied to speed up the computational burden of strong branching, in particular by heuristically restricting the list of branching candidates and imposing simplex iteration limits on the strong branching LPs [2] or by ruling out inferior candidates during the strong branching process [7]. However, according to computational studies, a pure strong branching rule is still too slow for practical purposes. Branching rules such as *reliability branching* [3] or *hybrid branching* [8], that combine ideas from pseudocost branching and strong branching, are considered today's state of the art.

Other approaches to branching include the *active constraint* method [9], which is based on the impact of variables on the set of active constraints; branching on general disjunctions [10]; *inference branching* and *VSIDS* [11, 12, 4] based on SAT-like domain reductions and conflict learning techniques. Finally, information collected through restarts is at the heart of the methods in [13, 14].

All branching strategies described so far are naturally designed to deal with only one optimal fractional solution. History-based rules use the statistics collected in the process to compute the score of a variable starting from the current fractional solution. Even with strong branching, the list of branching candidates is defined according to the current fractional solution $x^\star$.

However, LP relaxations of MIP instances are well-known for often being massively degenerate; multiple equivalent optimal solutions are the rule rather than the exception. Therefore, branching rules that consider only one optimal solution risk taking arbitrary branching decisions (thus contributing to performance variability, see [15, 16]), or being unnecessarily inefficient. In the present paper, we study the extension of some branching strategies to exploit degeneracy and the knowledge of multiple optimal solutions of the current LP relaxations (*a cloud of solutions*). In this course, the following questions arise:

(i) How common is dual degeneracy in MIP instances from standard test sets? Should it be a standard consideration in branching rules?

(ii) How could the knowledge of multiple optimal solutions to the current LP relaxation be exploited in branching rules?

(iii) How can multiple optimal solutions be generated efficiently?

The contribution of the present paper lies in discussing and answering all three questions. First, we address question (i) by discussing the presence of dual degeneracy in typical MIP models and analyzing how it changes during the branch-and-bound search (Section 2). Question (ii) is answered in Section 3, where we introduce a new branching rule based on the knowledge of a cloud of LP optima and show how existing branching rules can naturally exploit cloud information. The potential savings in tree size and strong branching calls are demonstrated by computational experiments. Section 4 treats question (iii), proposing more efficient ways to sample a cloud of solutions. All of this contributes to Section 5, which illustrates that the cloud variant of full strong branching leads to significant savings in computation time on standard MIP test sets. On the same sets, we demonstrate that also hybrid branching can be improved considerably by incorporating degeneracy information. Finally, some conclusions are drawn in Section 6.

Previous work on this topic was already discussed in the conference proceedings version [17] of this paper. It presented the idea of generating a cloud of solutions, modifications of pseudocost branching and full strong branching to use the cloud, and computational results for the latter. The present paper extends this work by providing answers to questions (i) and (iii) and a more thorough discussion of question (ii), suggesting one new branching rule based on cloud information and modifications of three more rules as well as a computational evaluation of all modified branching rules.

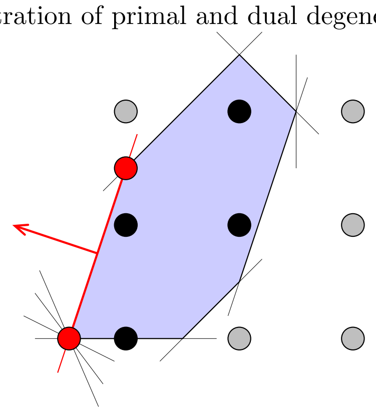
2 Dual degeneracy in MIP

Degeneracy describes the case that multiple simplex bases correspond to the same primal or dual solution of an LP. It has been a topic of interest since the invention of the simplex method, see, e.g., [18, 19]. A solution to a linear program is called *primal degenerate* if there are basic variables set to one of their bounds in the solution. Those variables can be pivoted out of the basis to obtain a different basis that still represents the same primal solution. Primal degeneracy can be a consequence of redundant constraints, but may as well be inherent in the specific problem and unavoidable.

In this paper, we are focusing on *dual degeneracy*. An LP solution is called *dual degenerate* if the corresponding dual solution is degenerate. The solution is degenerate if one of the primal non-basic variables has reduced costs zero. These variables will be called *dual degenerate* in the following. A dual degenerate optimal LP solution implies that there is a potential for alternative optimal solutions to this LP. More specifically, each dual degenerate variable can be pivoted into the basis without changing the objective value. Thus, multiple optimal bases are guaranteed to exist. If the variable that is pivoted out of the basis is not subject to primal degeneracy, i.e., was not at its bound before, this corresponds to a distinct primal optimal solution represented by the new basis.

An illustration of the two types of degeneracy is given in Figure 1. The upper red point lies at the intersection of exactly two hyperplanes defined by constraints. With only two structural variables (plus slack variables), this solution is primal non-degenerate. The dark red point lies at the intersection of five hyperplanes. Even if both structural variables are basic, three of the slack variables of those inequalities have to be basic as well although the inequalities are fulfilled with equality, meaning that the slack variables have value zero and are degenerate. Therefore, this solution is primal degenerate. Both red points and all points on the line in between are optimal

Figure 1: Illustration of primal and dual degenerate solutions.



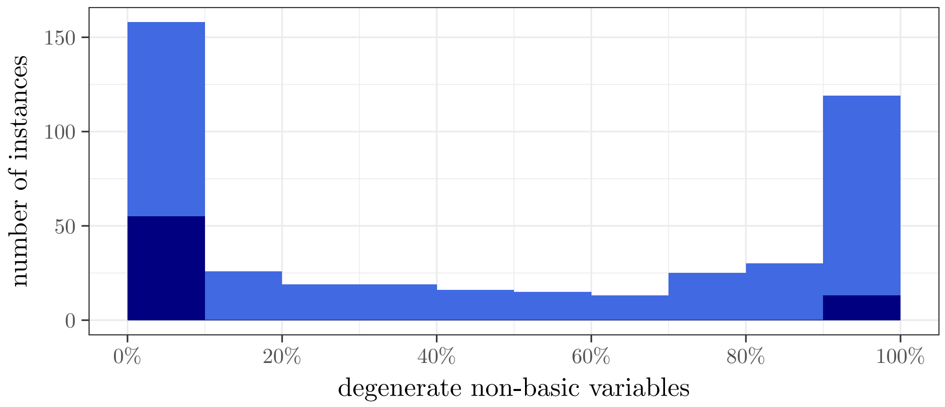
solutions; thus, the two red points are both dual degenerate basic solutions. For the upper point, for example, the slack variable of the constraint drawn as a black line is non-basic with reduced cost zero, since this constraint has non-zero slack in the lower optimal solution.

In case of dual degeneracy, which of the alternative optimal solutions is returned by the LP solver is arbitrary and cannot be predicted. While this may be acceptable when solving a pure LP, it quickly becomes an issue for LP relaxations being solved during the solving process of a MIP solver. A MIP solver takes many decisions based on the current LP solution, and if this solution is just arbitrary among many possible ones, a slight change in the algorithm can lead to a very different LP solution being returned. As a conseuqence, the decisions that are taken based on the LP solution may change, and therewith the subsequent solving process can vary significantly. Such sensitivity to small, seemingly performance neutral, changes in the algorithm is called *performance variability* [15, 16] and should typically be avoided. Thus, one motivation for taking into account multiple optimal LP solutions is to reduce performance variability and make the solver more stable.

On the other hand, decisions that are based on the single computed LP solution use very limited data. In many cases, it may be preferable to take into account multiple optimal LP solutions or to select a particular one among the set of optimal solutions. The pump-reduce procedure [20] and primal solution polishing, as implemented in SoPlex 3.1 [21], are examples for the latter case which mainly aim at increasing integrality of the LP solution to aid primal heuristics and branching rules. The former is the more interesting case. Pump-reduce and concurrent root cut loops [22] exploit a small set of optimal solutions for the separation process, dual variable picking [23] tries to increase the number of reduced cost fixings by the investigation of multiple dual solutions for primal degenerate LPs. Multiple ways to exploit degeneracy in different algorithmic components of the MIP solver Gurobi are discussed in a talk by Achterberg [24]. In this paper, we are focusing on branching improvements obtained by exploiting dual degeneracy and the knowledge of multiple optimal LP solutions. To the best of our knowledge, this is so far the only research that focuses on exploiting degeneracy within the branching component of MIP solvers.

First, let us investigate how common dual degeneracy is in real-world MIPs. Our test set for the remainder of this paper is the MMMC test set which contains all instances from MIPLIB 3 [25], MIPLIB 2003 [26], and the MIPLIB 2010 benchmark set [15] as well as the Cor@l test set [27], which mainly contains instances that users worldwide submitted to the NEOS server [28]. Duplicate instances were removed, leaving us with a total of 496 instances. We use the academic MIP solver SCIP 5.0.1 [4, 21] with SoPlex 3.1.1 [29, 21] as underlying LP solver.

Figure 2: Share of non-basic variables that are dual degenerate (final root LP). The darker parts represent instances with the extreme degeneracy rates of 0 % and 100 %.



The observations we describe in the following naturally depend on the branching rule used for the experiments. We used reliability branching and provided the optimum as a cutoff bound to focus on the branching performance. We chose that branching rule because it is a state-of-the-art rule used (in slight variations) by many solvers. Still, we are optimistic that the results should transfer to other common branching rules.

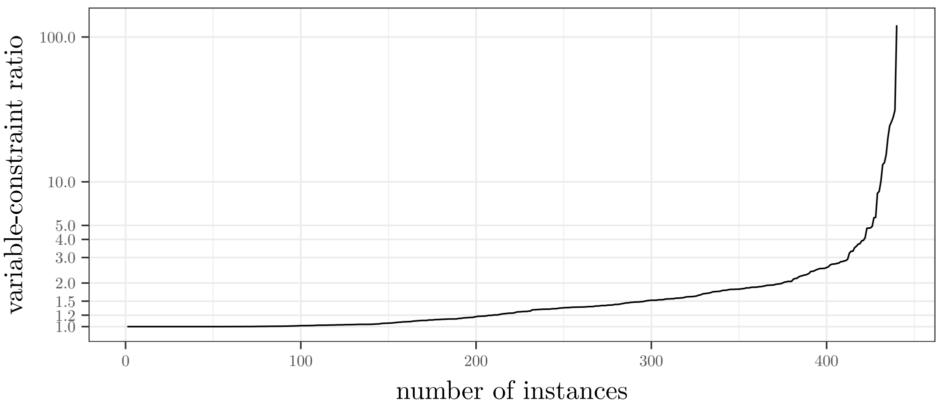
We are interested in the amount of degeneracy inherent in the LP solution used for branching. To get a first impression, we collected statistics about the degeneracy in the LP solution that SCIP uses for the final branching call at the root node, i.e., the LP solution after presolving, node preprocessing, the cutting plane separation loop, and potential restarts.

Figure 2 illustrates the relative number of dual degenerate non-basic variables in these LP solutions. Thereby, we disregard non-basic variables that have been fixed at the root node, independently of their reduced costs. The degeneracy rate is between 0 % (if all non-basic unfixed variables have non-zero reduced cost) and 100 % (if all non-basic unfixed variables have reduced cost zero). Each bar shows the number of instances with degeneracy rate in a certain 10 % range (except for the first bar, all bars are left-open). Instances with the extreme degeneracy rates of 0 % and 100 % are highlighted in the respective bars by a darker color.

Out of the 496 instances, 56 instances are solved at the root node before degeneracy information is computed. Of the remaining 440 instances, only 55 instances show no dual degeneracy in the solution of the final root LP. On the other hand, 13 instances have only degenerate non-basic variables, i.e., have a degeneracy rate of 100 %. These instances are pure feasibility problems with a zero objective function in the original model (6 instances), after presolving (4 instances), or after root processing (3 instances). Overall, it looks like degeneracy tends to cover only a few or almost all variables: 158 instances have a degeneracy rate no larger than 10 % while the rate is larger than 90 % for 119 instances. The remaining instances distribute among the degeneracy rates from 10 % to 90 % with slightly fewer instances at medium ranges.

In order to exploit dual degeneracy, we will investigate the optimal face of the current LP relaxation and generate additional points lying on this face. We can restrict the LP to the optimal face by fixing all variables to their current value whose reduced costs are non-zero, using the reduced costs associated with the starting optimal basis. By additionally changing the objective function and optimizing over the restricted LP, we can move to different optimal bases that potentially represent alternative LP optima. Different possibilities of doing this are

Figure 3: Variable-constraint ratios of the optimal face for instances in the MMMC test set (final root LP).



discussed in the second part of this section and Section 4.

Next, we investigate another measure for degeneracy that depends on the size of the restricted problem. We denote the number of unfixed variables in the restricted problem by $\bar{n}$ and call $\beta = \frac{\bar{n}}{m}$ the variable-constraint ratio of the optimal face. This ratio is 1 if no non-basic variable is dual degenerate, as only the basic variables remain unfixed. It increases as the number of dual degenerate non-basic variables increases.

The share of non-basic dual degenerate variables as illustrated in Figure 2 expresses how many non-basic variables may be pivoted into the basis, which potentially assigns a fractional solution value to these variables that are initially at their (integral) bound. As we will see later, the more interesting information is if a basic variable with a fractional value may be pivoted out of the basis, making it integral. We expect the probability for this to be higher the larger the variable-constraint ratio is. While this is true for the share of non-basic dual degenerate variables as well when looking at a particular instance, it does not allow to easily compare instances with different problem sizes. As an example, the share of non-basic dual degenerate variables would be the same (50 %) if one of only two non-basic variables is dual degenerate as it would be if 500 000 out of a million non-basic variables are dual degenerate. For a fixed basis size, however, the latter will probably allow pivoting more of the current basic variables out of the basis.

The variable-constraint ratios for the final root LP of the 440 instances from the MMMC test set not solved without any branching are presented in Figure 3. There are 55 instances with a ratio of 1.0, which are the instances showing no degeneracy at all. 111 and 129 instances have ratios larger than 1 but no larger than 1.1, and larger than 1.1 but no larger than 1.5, respectively. Overall, 374 instances have a variable-constraint ratio of the optimal face no larger than 2.0 and 412 instances have a still reasonable ratio of 3.0 or smaller. On the other hand, there are 10 instances with a ratio larger than 10; one of them even has a ratio of 120.

We observed degeneracy in the final root LPs of most of the regarded instances, but how does the degeneracy change during the tree search? In order to investigate this, we ran each instance with a node limit of 1 million and a time limit of 2 days. We computed the average degeneracy per depth level in the branch-and-bound tree by first averaging over all nodes of one depth for each instance and then over the instances for each depth level. Note that we disregard variables fixed by branching or domain propagation for the degeneracy computation and only compute the degeneracy share of the unfixed variables. The observed average degeneracy rate is

Figure 4: Difference between average degeneracy and root node degeneracy per depth level. Each point represents one instance, their distribution is visualized by the underlying box plot and the average is represented by ◈.

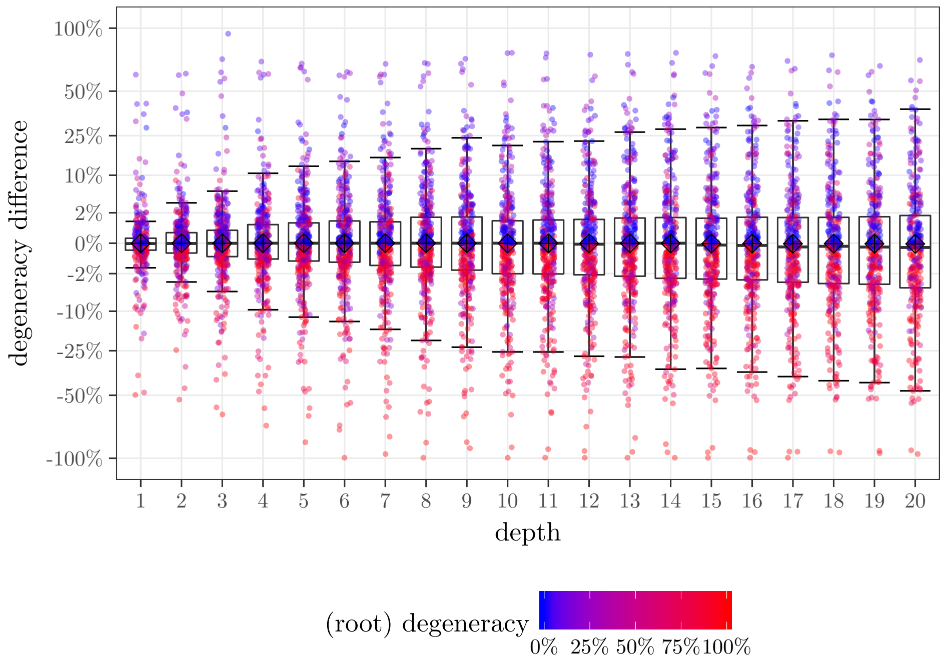
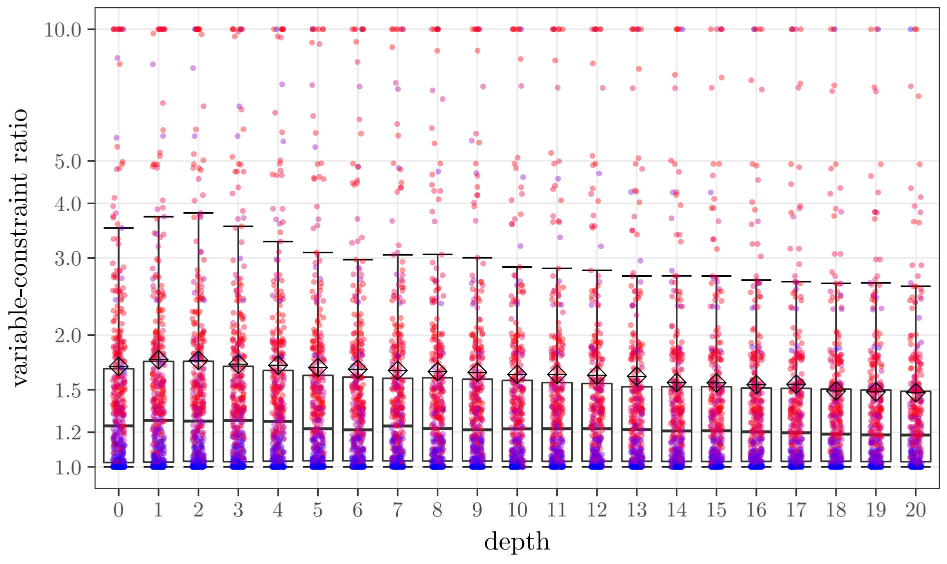


Figure 5: Average variable-constraint ratio of the optimal face per depth level in the branch-and-bound tree. Each point represents one instance, their distribution is visualized by the underlying box plot, and the average is presented by ◈.



almost constant during the tree search. At the root node, the average degeneracy rate is 45.8 %, at depth 100, it is 45.7 %. In between, it varies slightly between 44.4 % and 49.4 %.

However, is the average degeneracy also constant for individual instances? This question is answered by Figure 4, which focuses on the change in the average degeneracy per depth level, compared to the degeneracy of the final LP of the root node (after potential restarts). It shows the first 20 levels of the tree. We chose this limit because the number of instances reaching this level is still reasonable: out of the 440 instances for which we computed the degeneracy at the root node, 327 reached a tree depth of 20. Each instance is represented by one point for each depth level; the color of the point represents the root degeneracy of the instance. The x-axis represents the depth level; coordinates are jittered, and points are drawn translucent to better display the density of instances at common degeneracy differences. The y-coordinate of a point represents the average difference between the degeneracy of that instance at the corresponding depth level to its degeneracy rate at the root node. Note that we use a square root scaling of the y-axis, i.e., we take the square root of the absolute difference, but keep the sign. This scaling provides a higher resolution for smaller values while still allowing for values with an absolute value smaller than 1, in particular, zero values, as opposed to a log scaling. We observe that many instances change their degeneracy rate during the tree search and that there are roughly the same number of instances with increased degeneracy rate at deeper levels as with reduced degeneracy rate. The variance increases with a higher depth in the tree, which is reasonable as the deeper a subproblem is in the tree, the more different it is to the global problem. In order to better evaluate the distribution of the points, Figure 4 also shows a box plot. It shows that the median is around 0 for most of the depth levels; it reaches its highest absolute value at level 20, where it is at $-0.03\,\%$. Half of the instances do not change their degeneracy rate a lot, as illustrated by the box. Up to depth level 5, 50 % of the instances change their degeneracy rate by less than 1 %, compared to the root node; at level 10, by no more than 2 %. Even at level 20, the average change in degeneracy rate of 50 % of the instances is between $-4.3\,\%$ and 1.7 %. On the other hand, there are also 25.7 % of the instances that change their average degeneracy rate by more than 10 % in either direction at level 20, and almost 12 % that change by more than 25 %. Unsurprisingly, instances that reduce their degeneracy rate typically started with a high degeneracy rate at the root node, while those increasing the degeneracy rate often had small degeneracy rates at the root LP.

Figure 5 shows the development of the variable-constraint ratio in the tree. As in the previous figure, each instance is represented by one point for each depth level, jittered to better show the number of instances in common regions. The y-coordinate corresponds to the variable-constraint ratio, using a $\log_{10}$ scale this time as all values are no smaller than 1.0. The color of each point displays the average degeneracy rate of this instance at the corresponding depth level. Note that we had 134 instance/depth combinations (of 27 different instances) with ratios between 11 and 162, which are displayed as a ratio of 10 in the figure to allow a more detailed view of the most interesting region.

We observe a tendency for slightly decreasing variable-constraint ratios with higher depths. An exception is the higher ratio in depth one compared to the root node, which is consistently identified by average, median, and the quartiles in the box plot. An implementation detail of SCIP causes this increase: at the end of the root node, all cutting planes that are not tight are removed from the LP. Thus, the denominator of the variable constraint ratio is decreased for the following nodes so that the ratio increases. The box plot shows that 75 % of the instances have an average variable constraint ratio of less than 1.75 at depth 1 and 2. This number decreases by increasing depth level and is below 1.5 for depth levels 19 and 20. We can also see that no more than 25 % of the instances have a ratio smaller than 1.02, which means that for more than half of the instances, the ratio is in a reasonable range. We also observe that instances with a high variable constraint ratio typically have a high degeneracy as well. There are exceptions, though, one of them being instance neos-495307, which has an average degeneracy rate of less than 4 % at all depth levels. However, since its matrix dimensions are very unbalanced (it has more than 9000 variables and only 3 constraints), the variable constraint ratio is very high and slowly decreases from 26 at the root node to 18 at depth 20.

This study of degeneracy allows us to give a first answer to question (i). For 87.5 % of the instances, the root LP solution is subject to dual degeneracy. Throughout the tree, almost half of the non-basic variables are dual degenerate on average. On the other hand, the variable-constraint ratio is larger than 1.2 for more than half of the instances up to depth level 17. Together, these numbers indicate that both many non-basic variables can become basic in alternative optimal LP solutions, possibly moving away from the bound they are set to, as well as many of the basic variables are non-basic for alternative optima, thus being set to one of their (integer-valued) bounds. The latter will be evaluated in more detail in the remainder of this section, but we can already conclude that dual degeneracy is prevalent in standard MIP models which provides a strong motivation to consider it in branching rules.

2.1 A cloud of optimal LP solutions

We saw that dual degeneracy is very common in real-world problems. This observation raises the question: how can the knowledge of multiple optimal solutions be exploited?

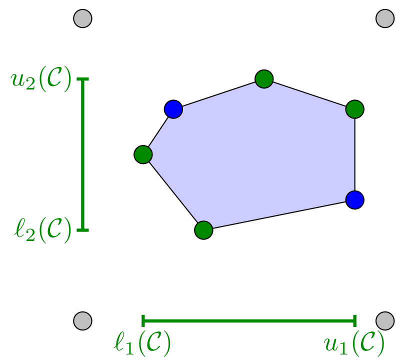
As discussed before, we can restrict the search to the optimal face $\mathcal{F}$ of the LP relaxation polyhedron by fixing all variables whose reduced costs are non-zero for the starting optimal basis. The validity of this approach follows directly from the fact that the objective value of any LP solution can be expressed as the sum of the current LP objective and the product of the current reduced cost with the change in the value of every non-basic variable, see [19]. Then, auxiliary objective functions can be used to move to different bases and generate alternative LP optima. We call the set $\mathcal{C} = \{x^1, \dots, x^k\}$ of generated optimal LP solutions a *cloud of solutions*. Storing a large number of such solutions, however, is impractical and also not needed to improve branching. What we are mainly interested in is the range of values each variable takes in the set of cloud solutions. Therefore, we define the *cloud interval* $\mathcal{I}_j(\mathcal{C})$ of variable x_j as $\mathcal{I}_j(\mathcal{C}) := [\ell_j(\mathcal{C}), u_j(\mathcal{C})]$, where

$$\ell_j(\mathcal{C}) = \min\{x_j \mid x \in \mathcal{C}\} \quad \text{and} \quad u_j(\mathcal{C}) = \max\{x_j \mid x \in \mathcal{C}\}.$$

The cloud interval captures the most important information in just two numbers per variable. An example of a simple optimal face, cloud solutions, and cloud intervals is given in Figure 6.

The set of cloud solutions and the associated cloud intervals depends on the choice of auxiliary objective functions. One option is to try to minimize and maximize each variable which is not yet fixed: this is what optimization-based bound tightening techniques do (see, e.g., [30, 31]), but applied to the optimal face. That is why we will refer to this option as the *obbt-style sampling* of cloud solutions in the following. In the worst case, this requires to solve $2m + n$ auxiliary LPs: two LPs for each of the m basic variables and one LP for each non-basic variable as they are initially set to one of their bounds and can only move in one direction. Using filtering techniques, see [32], the number of LPs to be solved can often be reduced, but typically stays too large to use this approach in a practical application at every node of the branch-and-bound tree. From a theoretical point of view, however, this method is interesting since the cloud points $\mathcal{C}$ it generates are guaranteed to generate the exact cloud intervals of the optimal face $\mathcal{F}$, i.e., $\ell_j(\mathcal{C}) = \min\{\ell_j(\mathcal{C}') \mid \mathcal{C}' \subseteq \mathcal{F}\}$ and $u_j(\mathcal{C}) = \max\{u_j(\mathcal{C}') \mid \mathcal{C}' \subseteq \mathcal{F}\}$. Therefore, we will use this method for a deeper analysis of degeneracy in our models in the remainder of this section as well as for evaluating the potential for improvements in branching rules by using cloud information as

Figure 6: Illustration of optimal face, set of cloud solutions, and cloud intervals. Out of the six extreme points of the optimal face, the four points marked green are sufficient to obtain the exact cloud intervals.



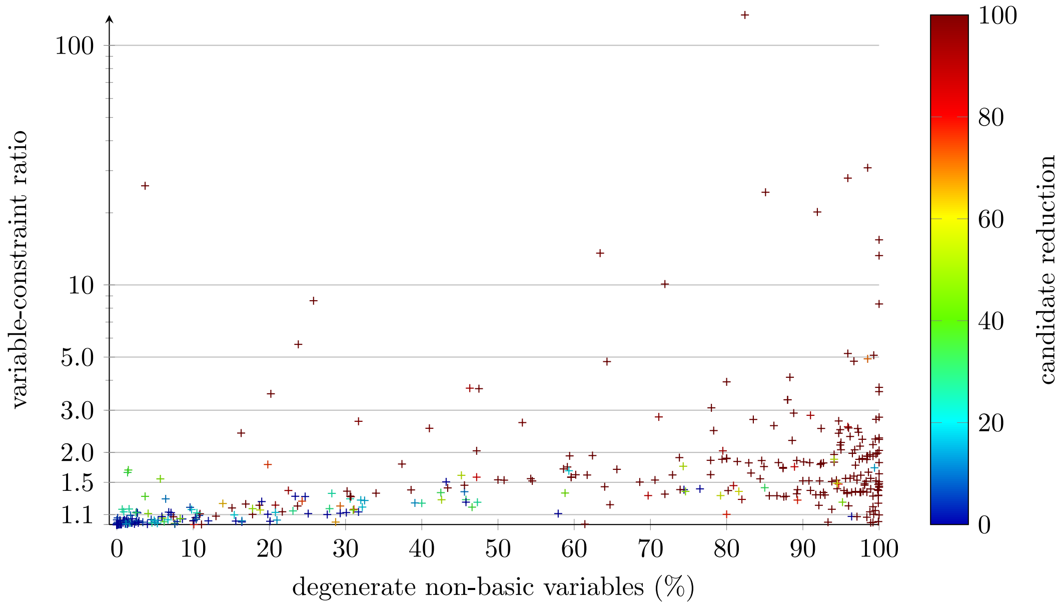
discussed in Section 3. Other methods to generate a partial set of cloud solutions and approximate cloud intervals are discussed in Section 4.

Figure 7 shows how many of the fractional branching candidates of the LP solution used for branching at the root node are found to have an integer point in their cloud interval. We call the share of such candidates the branching candidate reduction in the following since we would normally prefer branching candidates that are not already guaranteed to not improve the dual bound for one child node. Each instance is represented by one cross that is positioned according to the share of non-basic variables that are degenerate (x-coordinate) and the variable-constraint ratio of the optimal face (y-coordinate, log scale). The color encodes the relative reduction in the number of branching candidates, i.e., candidates with an integer point in the cloud interval. First, we can see that a small degeneracy rate typically leads to a small variable-constraint ratio and a small reduction in the number of branching candidates. That is the expected behavior, but there are outliers as well, typically with larger variable-constraint ratios, that results in high branching candidate reductions. In general, a variable-constraint ratio of 2.0 or larger leads to very high candidate reductions. Out of the 63 instances that fall into this category, 79.4 % have integer points in the cloud intervals of all branching candidates at the final root LP. On one instance, we can filter out 71.4 % of the candidates this way, while the remaining ones allow filtering out around 90 % and more. For high degeneracy rates, the variable-constraint ratios increase as does the reduction in the number of branching candidates. Out of the 152 instances with a degeneracy rate of 80 % or higher, 127 instances have integer points in the cloud intervals of all branching candidates.

The one outlier in the upper left area is again instance neos-495307 with a degeneracy rate of 3.7 % and a variable-constraint ratio of 25.9. Consequently, all 5 branching candidates at the root node have integer points in their cloud intervals. We would not have expected this based on the degeneracy rate alone, which illustrates the usefulness of the variable-constraint ratio as a degeneracy measure. As discussed before, the average variable-constraint ratio per depth level for this instance stays extremely high for the first 20 depth levels. As a consequence, all branching candidates have integer points in their cloud intervals for more than 95 % of the nodes up to depth 20.

Figure 8 shows the average branching candidate reduction per depth level of the branch-and-bound tree for all instances reaching depth 20. The color scale is similar to the one in Figure 7 but split into 10 % buckets with extra buckets for candidate reductions of 0 % and 100 %. At the root node, almost 25 % of the instances do not have a single fractional variable that has an

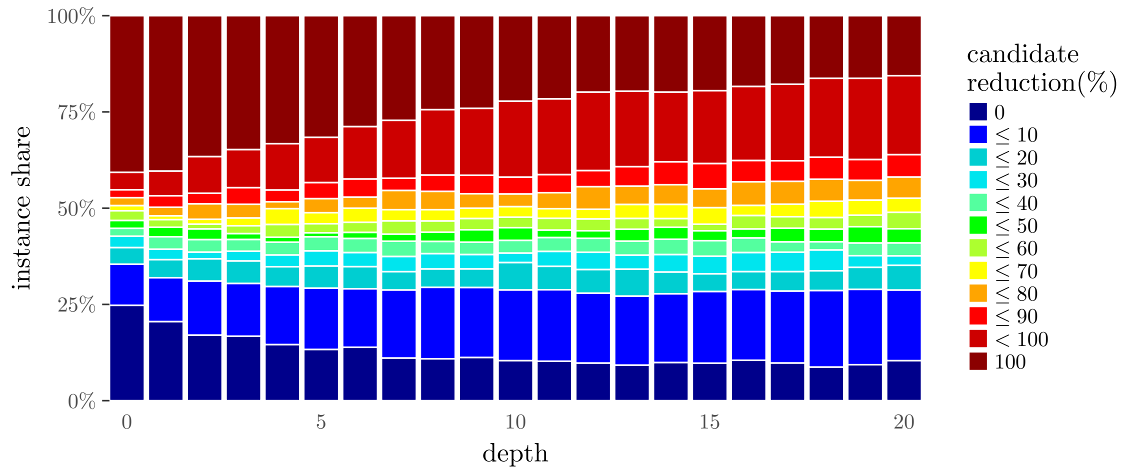
Figure 7: Share of branching candidates with an integer point in their cloud interval (color), plotted by dual degeneracy share (x-axis) and variable-constraint ratio of the optimal face (y-axis).



integer point in its cloud interval. The other extreme—an integer point in the cloud interval of each fractional variable—can be observed for almost 40 % of the instances. The remaining instances are split among the other buckets with more instances going into the more extreme buckets. For the deeper levels of the tree, we again averaged over all nodes in that level for each instance before averaging over the instances. We can see that the two extreme cases happen less frequently in deeper levels of the tree. At depth 20, only 11.3 % of the instances have no candidate with an integer value in its cloud interval at any node in this depth. This decrease, however, comes with an increase in the number of instances with more than 0 % and less than 10 % candidate reduction. The sum of these two cases stays almost the same over the different depth levels in the tree. For high candidate reductions, the picture looks similar. At depth level 20, only 16.5 % of the instances have an integer value in each fractional variable's could interval, while the share of instances with a candidate reduction between 90 % and 100 % increases from 6 % to 18.7 %. Nevertheless, their sum decreases by more than 10 % so that there are more instances with integer values in the cloud intervals of some, but not (close to) all or none of the candidates.

The improvements presented in the following section profit a lot from high candidate reductions, as this allows to focus on a small number of most promising candidates. However, this is only the case if the reduction rate is less than 100 %; otherwise, cloud information only proves that none of the candidates will improve the dual bound for both child nodes, but still one has to be selected for branching. In this light, Figure 8 provides some indication that the cloud improvements can be useful throughout the tree and might even get more important as the tree search progresses. Coming back to question (i) posed in the introduction, we conclude that dual degeneracy is very common and that we see a significant potential to improve branching rules by exploiting it.

Figure 8: Branching candidate reduction by depth level. Each bar is split into multiple pieces according to the share of instances with branching candidate reduction in a particular range (color).



3 Exploiting a cloud of solutions in branching

This section discusses how the knowledge of a cloud of solutions can be exploited in branching. We first present a generic filtering algorithm based on cloud information before describing modifications of common branching rules as well as a new branching rule based on cloud information.

3.1 A cloud-based branching candidate set

The most straightforward way to exploit the cloud in a branching rule is to use it to define the candidate set F. Instead of defining the candidate set $F(x^\star) = \{j \in J : x^\star_j \notin \mathbb{Z}\}$ based on a random optimal solution $x^\star$ to the current LP relaxation, we use the complete cloud $\mathcal{C} = \{x^1, \dots, x^k\}$ of alternative LP-optimal solutions. We can then define an extended candidate set $F(\mathcal{C})$ as

$$F(\mathcal{C}) = \{j \in J \mid \mathcal{I}_j(\mathcal{C}) \setminus \mathbb{Z} \neq \emptyset\}$$

i.e., $F(\mathcal{C})$ contains all the variables that are fractional in at least one optimal LP solution that can be expressed as a convex combination of the given cloud solutions.

Given the cloud interval for each branching candidate, we partition the set $F(\mathcal{C})$ into three subsets, depending on the number of integer values contained in each interval $\mathcal{I}_j(\mathcal{C})$. We define:

$$F_0(\mathcal{C}) = \{j \in F(\mathcal{C}) \mid \mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z} = \emptyset\},$$

$$F_1(\mathcal{C}) = \{j \in F(\mathcal{C}) \mid |\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}| = 1\}, \text{ and}$$

$$F_2(\mathcal{C}) = \{j \in F(\mathcal{C}) \mid |\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}| \geq 2\}.$$

For binary variables, $F_0(\mathcal{C})$ contains exactly those variables which are fractional for all $x^i \in \mathcal{C}$, or differently spoken: $F_0(\mathcal{C})$ is the intersection (taken over $\mathcal{C}$) of all branching candidates. If $\mathcal{C}$ contained all vertices of the optimal face, then $F_0(\mathcal{C})$ would be exactly the set of variables that are guaranteed to improve the dual bound in both child nodes. The hope is that also with a limited set of sample points in $\mathcal{C}$, $F_0(\mathcal{C})$ is still a good approximation to that set.

A variable x_j being contained in the set $F_1(\mathcal{C})$ is a certificate that branching on that variable will not improve the dual bound for at least one of the child nodes since there is an alternative optimum with $x_j = |\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}|$. Even worse, for each variable x_j contained in the set $F_2(\mathcal{C})$, there is at least one split, e.g., $x_j \leq \min\{|\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}|\} \vee x_j \geq \min\{|\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}|\} + 1$, that will not improve the dual bound on either side since alternative optima exist which respect the bounds after branching.

Note that in most cases, the cloud of solutions $\mathcal{C}$ that F_0, F_1, and F_2 are based upon is given by the context. Therefore, we simplify the notation by omitting the reference to the cloud $\mathcal{C}$ unless we need to distinguish different sets of cloud solutions.

Each common branching rule can now easily be improved with this knowledge. Instead of choosing the branching variable from the candidate set $F(x^\star)$, we use the restricted candidate set F_0. Note that $F_0 \subseteq F(x^\star)$ since $x^\star \in \mathcal{C}$. Other than that, the branching rule is not changed, i.e., it still scores the remaining candidates based on a single LP optimum $x^\star$.

For very degenerate problems, it might happen that $F_0 = \emptyset$, see Figure 8. In that case, the branching rule could switch to F_1 as the candidate set or even F_2 if also $F_1 = \emptyset$. However, $F_1 \subseteq F(x^\star)$ does not necessarily hold anymore, and so $x^\star_j \in \mathbb{Z}$ may hold for some $j \in F_1$. In that case, the branching disjunction for that variable is not implied by $x^\star_j$ and might also be non-unique for integer variables. Treating this case within a branching rule requires a modification of the rule itself and not only of the candidate set. However, we can still filter the candidate set if $F_0 = \emptyset$ by selecting the branching variable from the reduced set $F_1 \cap F(x^\star)$ and only fall back to $F(x^\star)$ if the latter is also empty (i.e., $F(x^\star) \subseteq F_2$).

The motivation for this filtering mechanism is that we want to prefer branching candidates which improve the dual bound for both child nodes over those that improve the dual bound for only one child node over those that will not improve the dual bound of any child node. While the latter is clear, it is not directly obvious that a small improvement in both directions is a better choice than a variable that improves the dual bound a lot in one direction, but leads to no improvement in the other. However, the latter case implies that this branching will never change the global dual bound, while a small improvement in both directions at least guarantees an increase in the global dual bound if the current node is the single dual-bound-defining node. Another argument for this filtering rule is that state-of-the-art solvers such as Cplex or SCIP compute the score of a variable as the *product* of the dual bound improvements in both directions (maybe using a minimum value of some epsilon close to zero for each factor). By this, the score of all variables in F_1 will be (nearly) zero and variables in F_0 will most likely be preferred.

Note that this filtering mechanism is similar in spirit to the strategy called *nonchimerical branching* proposed in [7], where the optimal solutions of the strong branching LPs (which might have a different objective function value) were used for this purpose. The two strategies have complementary strengths: nonchimerical branching does not need to solve any additional LP compared to strong branching but needs the strong branching LPs to be solved to optimality, because of the usage of the dual simplex. Cloud branching, on the other hand, needs additional LPs, but these are in principle simpler (we are fixing to the optimal face), need not be solved to optimality (primal simplex is used), and do not impose any requirements to the solution of the final strong branching LPs. As such, the two techniques can be easily combined and might synergize. Moreover, cloud branching can be used independent of strong branching to improve most of the common branching rules, as will be described in the next subsections.

In order to assess the potential of the simple filtering, we applied it to a set of common branching rules. Thereby, we computed exact cloud intervals by applying obbt-style sampling of cloud points. Although this is computationally too expensive to expect a speedup, we use this to evaluate the potential of cloud filtering with respect to the size of the branch-and-bound tree generated to solve instances to optimality.

	Δ Nodes		Δ Fair Nodes	
branching rule	> 1 node	≥ 100 nodes	> 1 node	≥ 100 nodes
random	-22.8%	-27.7%	-22.8%	-27.7%
mostinf	-21.5%	-26.8%	-21.5%	-26.8%
pscost	-21.1%	-25.5%	-21.1%	-25.5%
fullstrong	13.0%	23.8%	-3.3%	1.1%
reliability	-8.5%	-11.6%	-8.7%	-11.3%

Table 1: Change in the shifted geometric mean of the tree size for common branching rules when filtering candidates based on cloud information.

All computational experiments in the remainder of this section use the MMMC test set and run SCIP with 5 different random seeds on each instance (one of them being the default seed). In a slight abuse of notation, we are referring to each of the 2480 instance/seed combinations as an individual instance in the following. All computations were performed with a time limit of 5 hours and a memory limit of 13 GB on one of two computing clusters. Cluster 1 provides two Intel Xeon Gold 5122 CPUs at 3.60GHz with 96 GB main memory at each compute node. It was used for all experiments on random and most infeasible branching presented in this section. Cluster 2 uses two Intel Xeon E5-2660 v3 CPUs at 2.60GHz with 126 GB main memory per compute node and was used for all experiments on pseudocost, full strong, and reliability branching discussed in the remainder of this section.

Additionally, we provide the optimal value as a cutoff bound and disable primal heuristics to focus on the branch-and-bound process and reduce performance variability. Since we are focusing on tree size, we do not rely on accurate time measurements and allow multiple processes to run in parallel on a cluster node to reduce the computational burden of the experiments. To allow for a fair comparison of branch-and-bound nodes, instances that one variant did not solve to optimality within the time limit are excluded from the comparison. Computing the cloud intervals requires to solve auxiliary LPs, after which the original LP solution is restored, which may lead to a different state in the LP solver that might cause side-effects. In order to obtain an unbiased comparison, we compute cloud intervals by obbt-style sampling not only for the branching rule variants using cloud information but also for the default variants. The log files of our computational experiments were parsed and all tables in this paper were created with the aid of the *Interactive Performance Evaluation Tools* (IPET) [33].

The impact of cloud-based candidate filtering on different branching rules is evaluated in Table 1. Each row lists the effect on one particular branching rule on two sets of instances. Set "$>$ 1 node" contains all instances where some branching was performed, i.e., instances that were not solved at the root node by both variants. Set "$\geq$ 100 nodes" restricts this set to instances that needed at least 100 nodes to solve by one of the variants. Columns two and three present the relative change in the shifted geometric mean of the number of nodes (with a shift of 10) over the instances in the two sets when using cloud filtering as compared to the branching rule without cloud filtering.

We see that the effect of cloud filtering is comparable for the three branching rules that do not rely on strong branching. In the case of random branching, the node reduction obtained by cloud filtering is almost 23 % for instances that need more than one node and 28 % for instances needing at least 100 nodes. The node reductions for most infeasible branching and pseudocost branching are about 1 % and 2 % smaller, respectively. This difference is reasonable as those branching rules themselves already apply more intelligence for selecting a good branching candidate. On the other hand, it shows that cloud information can still improve these branching rules considerably.

For full strong branching, we see a negative effect: the node number is increased when performing cloud filtering. The reason for this is that cloud filtering does not provide additional information, as strong branching computes the dual bound changes in both potential child nodes for each candidate. Therefore, for each variable filtered out, strong branching would find out itself that this branching will not lead to an increased dual bound for at least one child node and the score of such candidates will be worse than that of candidates that provide an improvement in both directions due to the product scoring method. On the other hand, strong branching may detect infeasibility of one child node even if this candidate could be filtered out because the other child does not increase the dual bound. In that case, the domain of this candidate can be tightened to the one of the feasible child node and SCIP will re-process the current node. This side-effect of strong branching exceeds the main aim of branching, which is to select a good branching candidate. It is not covered in the node number, although it is information that is only available through the solving of two auxiliary strong branching LPs and implies additional effort that is spent for re-optimizing the current node. A new measure to gain a fair comparison of the quality of the branching decisions without side-effects was introduced in [34]. The *fair node number* defined there is based on an auxiliary branch-and-bound tree that encodes the same information as the tree built by the solver. However, instead of applying bound changes found through strong branching at the current node, branching is performed on the respective variable, see [34] for more details. The last two columns show the effect of cloud filtering on the average fair node numbers. When disregarding the side-effects of strong branching, the node number stays almost the same when applying cloud filtering to full strong branching; small changes are probably due to performance variability. All previously discussed branching rules produce no side-effects, so the fair node number does not differ from the default node number.

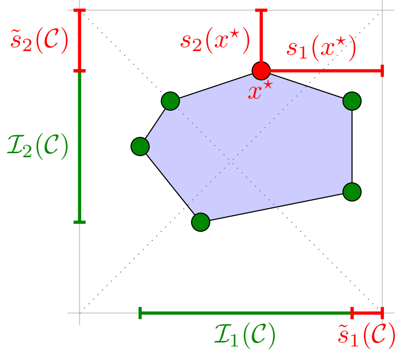
For reliability branching, being a combination of pseudocost branching and full strong branching, the node reduction lies somewhere in between the reductions obtained for its two components. For instances that need at least 100 nodes, the tree size is reduced by about 13 %. Note that the reduction is slightly smaller in the fair node number. This difference, however, is a consequence of generally higher node numbers obtained when using this measure. The absolute reduction in the node number is even larger in this measure (about 950 compared to 750 for the default node number).

Summing up, this experiment shows that for all branching rules we investigated that use imperfect knowledge of the dual bounds of potential child nodes, cloud filtering can improve the branching decision considerably. The improvement is slightly smaller the more intelligence the branching rule applies already without the filtering. The extreme case is full strong branching, which—using perfect knowledge of child node dual bounds—cannot profit significantly from cloud filtering in terms of the tree size. However, in this case, cloud filtering can be used to reduce the number of strong branching calls needed and thus the effort per node. More fundamental adjustments of branching rules based on cloud information are discussed in the following.

3.2 Most infeasible cloud branching

Most infeasible branching is a typical textbook method for branching, see, e.g., [35]. It selects a variable $x_j, j \in F$ for branching with fractional value $x^\star_j - \lfloor x^\star_j \rfloor$ closest to 0.5. This approach was reported to be as good as selecting a branching variable randomly [3]. In our experiments with the rules as they are implemented in SCIP, however, we observed a better performance in terms of tree size as compared to random branching. More precisely, most infeasible branching needed about 27 % fewer nodes in the shifted geometric mean on instances that were solved with a tree; the difference is slightly smaller with cloud filtering as this has slightly more effect on

Figure 9: Illustration of most infeasible branching and its variant taking into account cloud information. Most infeasible branching uses $x^\star$ to select a branching candidate and chooses x_1, while cloud information suggests that x_2 is to be preferred.


$\tilde{s}_2(\mathcal{C})$
$s_2(x^\star)$
$s_1(x^\star)$
$x^\star$
$\mathcal{I}_2(\mathcal{C})$
$\mathcal{I}_1(\mathcal{C})$
$\tilde{s}_1(\mathcal{C})$

random branching. However, a high degree of dual degeneracy, as we have observed it for many problems, means that variables can have different values in alternative optimal solutions, with different fractionalities. We see this as a burden for most infeasible branching, which very likely suffers from the random nature of which of the optimal solutions is returned in case of dual degeneracy.

In order to overcome this issue, we introduce *most infeasible cloud branching*. Instead of selecting a variable which maximizes $s_j(x^\star) := \min\{x_j^\star - \lfloor x_j^\star \rfloor, \lceil x_j^\star \rceil - x_j^\star\}$, it selects a variable $x_j \in F_0$ maximizing

$$\tilde{s}_j(\mathcal{C}) := \min\{\ell_j(\mathcal{C}) - \lfloor \ell_j(\mathcal{C}) \rfloor, \lceil u_j(\mathcal{C}) \rceil - u_j(\mathcal{C})\}. \tag{1}$$

If $F_0 = \emptyset$, i.e., all candidates have at least one integer value in their cloud interval I_j, we select the variable $x_j \in F_1 \cap F(x^\star)$ with the highest distance between the cloud interval and its upper bound rounded up as well as its lower bound rounded down. More precisely, we maximize $\max\{\ell_j(\mathcal{C}) - \lfloor \ell_j(\mathcal{C}) \rfloor, \lceil u_j(\mathcal{C}) \rceil - u_j(\mathcal{C})\}$. If the cloud interval ends at an integer value at one side, the maximum is always obtained at the other side, preferring a cloud interval $[0, 0.1]$ over a cloud interval $[0, 0.3]$. This behavior is in line with the original most infeasible branching idea: in both cases, the down-branch will not move away from the optimal face, but the value of the variable in the up-branch will be further away from the optimal face in the former case. If the cloud interval has the integer value in its interior, we also need to decide which branching disjunction is used for this variable, $x_j \leq \lfloor \ell_j(\mathcal{C}) \rfloor \vee x_j \geq \lceil \ell_j(\mathcal{C}) \rceil$ or $x_j \leq \lfloor u_j(\mathcal{C}) \rfloor \vee x_j \geq \lceil u_j(\mathcal{C}) \rceil$. In that case, we prefer the disjunction with the higher score, i.e., the first one if $\ell_j(\mathcal{C}) - \lfloor \ell_j(\mathcal{C}) \rfloor > \lceil u_j(\mathcal{C}) \rceil - u_j(\mathcal{C})$ and the second one otherwise. Note that we are not using F_1 as the candidate set, but $F_1 \cap F(x^\star)$. In preliminary experiments, this turned out to be more effective. We believe that this is due to the fact that selecting a candidate from $F_1 \setminus F(x^\star)$ means that the LP solution will stay the same for one of the created child nodes. On the other hand, a candidate from $F_1 \cap F(x^\star)$ leads to different LP solutions in both child nodes, even though one of them will have the same objective value. This changed solution, however, could still be integral or lead to reductions, e.g., based on reduced costs. As a last resort, if we did not find a suitable candidate so far, we use default most infeasible branching.

Figure 9 illustrates the difference between standard most infeasible branching and most infeasible cloud branching. The LP solution returned by the LP solver is extreme on the optimal face $\mathcal{F}$ in the direction of x_2, while being rather centric with respect to x_1. Therefore, most infeasible

setting	> 1 node (712)	≥ 100 nodes (547)
mostinf	2300.0	9672.7
+ filter	1806.6	7091.0
+ score	1612.0	6082.8

Table 2: Most infeasible branching w/o cloud information, with cloud filtering, and with cloud scoring on the MMMC test set.

branching prefers x_1. On the other hand, the LP face $\mathcal{F}$ spreads wider into x_1-direction than x_2-direction, and there is only one other extreme point of $\mathcal{F}$ for which x_2 is more fractional than x_1, while the reverse holds for three of the six extreme points. More generally, the area of $\mathcal{F}$ for which the fractionality of x_1 is larger than that of x_2 is larger than that of the area where this is not true. The dotted diagonal lines split these areas, the former ones are left and right, while the latter ones are at top and bottom. All of this indicates that the probability that most infeasible branching prefers x_1 over x_2 is smaller than for the reverse if the LP solver would return one of the extreme points uniformly at random, or even any point $\tilde{x} \in \mathcal{F}$. In this light, most infeasible cloud branching "de-randomizes" most infeasible branching.

Note, however, that the observations we made based on the example hold in many cases, but it is neither guaranteed that a larger cloud interval corresponds to a higher number of less fractional extreme points nor to a larger area that is less fractional in this direction. Still, the original motivation for branching on the most fractional variable is that this forces a substantial change to the value of this variable on both sides. Most infeasible cloud branching builds on this argument but uses the distance between cloud interval and the two next integer values. By maximizing the minimum distance, it also aims at performing a balanced branching.

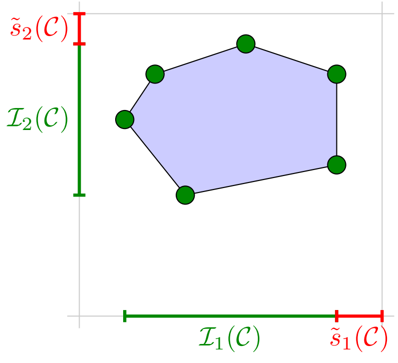
Computational experiments show an improvement in the quality of the branching decisions taken when integrating cloud information into most infeasible branching, see Table 2. For reference, we list results for default most infeasible branching, most infeasible branching with cloud filtering, and most infeasible cloud branching. The subsets of instances are defined as in Section 3.1 but with respect to all three variants; thus, the actual sets differ slightly. Nevertheless, the improvement by cloud filtering is similar: the tree size is reduced by 21.5 % and even 26.7 % for instances where at least one variant needed 100 nodes. The additional improvement by using cloud information in the branching score is considerably smaller but still relevant. For all instances that were solved, but not at the root node, we see an additional node reduction by 10.8 %, while for instances with larger trees of at least 100 nodes, this increases to 14.2 %. Compared to default most infeasible branching, the tree size is even reduced by 37.1 % for such larger trees.

3.3 Cloud diameter branching

In this section, we propose a new branching rule based only on cloud information. It is similar to most infeasible branching in the sense that it does not predict objective value changes as pseudocost or strong branching do, but solely relies on the solution values of the variables—in this case, the cloud intervals.

Cloud diameter branching compares the branching candidates based on the diameter of the optimal face in the direction of the variable. More precisely, for $j \in N$ and an optimal face $\mathcal{F} \subseteq \mathbb{R}^n$, we call $\varnothing_j(\mathcal{F}) = \sup\{d_j(x,y) \mid x, y \in \mathcal{F}\}$ with distance function $d_j(x,y) = |x_j - y_j|$ the diameter of $\mathcal{F}$ in direction j. Now, the larger the diameter of $\mathcal{F}$ is in the direction of a branching candidate, the further this variable can be moved without changing the optimal LP value, and

Figure 10: Comparison of diameter branching and most infeasible cloud branching. Most infeasible cloud branching would prefer x_1 since it maximizes the fractionality score $\tilde{s}$; diameter branching prefers x_2 because it has a smaller cloud interval $\mathcal{I}(\mathcal{C})$.


$\tilde{s}_2(\mathcal{C})$
$\mathcal{I}_2(\mathcal{C})$
$\mathcal{I}_1(\mathcal{C})$
$\tilde{s}_1(\mathcal{C})$

setting	> 1 node (688)	≥ 100 nodes (540)
mostinf	1992.7	7160.1
+ cloud	1373.9	4473.9
diameter	1034.5	3072.4

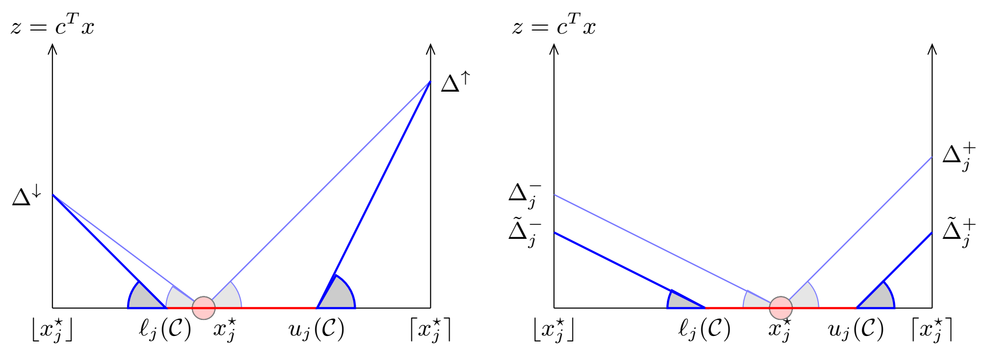
Table 3: Comparison of most infeasible branching (without and with cloud scoring) and diameter branching on the MMMC test set.

thus, the smaller are the improvements we expect when branching on this variable. Therefore, we aim at branching on a variable x_j that minimizes $\varnothing_j(\mathcal{F})$.

With complete cloud information, $u_j(\mathcal{C}) - \ell_j(\mathcal{C})$, i.e., the length of the cloud interval of x_j, equals the diameter of the optimal face $\mathcal{F}$ in this direction. Moreover, also with incomplete cloud information, we can use the cloud interval length as an approximation of the diameter of $\mathcal{F}$. We use the same tiebreaker as most infeasible branching and prefer variables with higher absolute objective function coefficient among variables with minimum cloud interval length.

The idea of diameter branching is visualized in Figure 10 and compared to most infeasible cloud branching. If $x_j \in F_0$, i.e., the cloud interval of x_j contains no integer point, minimizing the cloud interval length $u_j(\mathcal{C}) - \ell_j(\mathcal{C})$ corresponds to maximizing the sum of distances from the cloud interval to the two nearest integers. In other words, we maximize the sum of the definite changes in this variable's value in the two child nodes created by branching on it. This goal is different to most infeasible cloud branching, which also uses the cloud interval information, but maximizes the distance to the closest integer point. If there is no variable without integer point in the cloud interval, cloud diameter branching falls back to standard most infeasible branching.

Table 3 presents computational results comparing cloud diameter branching with most infeasible branching and most infeasible cloud branching. The computations were all performed on cluster 1 using the experimental setup described above. We observe an additional tree size reduction of 24.7 % for all instances not solved at the root node and 31.3 % for those that needed at least 100 nodes with one of the branching rules. These results indicate that the cloud diameter is a better criterion for selecting the branching candidate than the distance to the next integer point, as used by most infeasible cloud branching.



(a) (Cloud) pseudocost update (b) Bound prediction based on (cloud) pseudocosts

Figure 11: Graphical representation of pseudocosts update and usage.

3.4 Pseudocost cloud branching

Pseudocost branching [1] predicts the dual bounds of potential child nodes based on historical information about the average increase of the dual bound after previous branchings on this variable. This historical information is called *pseudocost.* Thus, pseudocost branching consists mainly in two operations: (i) updating the pseudocosts after an actual branching has been performed and the LP relaxations of the child nodes have been solved and (ii) computing the score of a variable using the current pseudocosts when deciding for a branching candidate. When updating the pseudocosts, the objective gains ς_j^- and ς_j^+ per unit change in variable x_j are computed; that is:

$$\varsigma_j^- = \frac{\Delta^{\downarrow}}{x_j^\star - \lfloor x_j^\star \rfloor} \quad \text{and} \quad \varsigma_j^+ = \frac{\Delta^{\uparrow}}{\lceil x_j^\star \rceil - x_j^\star}, \tag{2}$$

where $\Delta^{\uparrow}$ and $\Delta^{\downarrow}$ are the differences between the optimal LP objectives of the corresponding child nodes and the current LP value for up and down branching, respectively. These gains are then used to update the current pseudocosts Ψ_j^- and Ψ_j^+ which are the averages of the objective gains ς_j^- and ς_j^+ that have been observed for that particular variable so far. The shaded lines in Figure 11(a) illustrate the operation. These estimation formulas are based on the assumption that the objective value increases linearly in both directions. This assumption, however, may be a too crude approximation of the real shape of the projection on the split domain of x_j. In the case of dual degeneracy, there might be many optimal LP solutions with different values for x_j. Which of these values $x_j^\star$ takes is more or less arbitrary, but crucial for the pseudocost update—and by that for future branching decisions.

Using the cloud interval $\mathcal{I}_j(\mathcal{C})$ on the other hand, it is possible to replace this approximation with a more precise model (printed as solid lines in Figure 11(a)). The new model changes the gain computation as follows:

$$\tilde{\varsigma}_j^- = \frac{\Delta^{\downarrow}}{\ell_j(\mathcal{C}) - \lfloor x_j^\star \rfloor} \quad \text{and} \quad \tilde{\varsigma}_j^+ = \frac{\Delta^{\uparrow}}{\lceil x_j^\star \rceil - u_j(\mathcal{C})}, \tag{3}$$

with $\tilde{\varsigma}_j^- = 0$ and $\tilde{\varsigma}_j^+ = 0$ if $\ell_j(\mathcal{C}) \leq \lfloor x_j^\star \rfloor$ and $u_j(\mathcal{C}) \geq \lceil x_j^\star \rceil$, respectively. Where the values for ς^- and ς^+ may vary when a different LP optimum is computed by chance, the cloud gains $\tilde{\varsigma}^-$ and $\tilde{\varsigma}^+$ only depend on the cloud interval and the new bounds $\lfloor x_j^\star \rfloor$ and $\lceil x_j^\star \rceil$. Thus, they are less prone to variability when sufficiently many corners of the optimal face are used as a cloud.

setting	> 1 node (872)	≥ 100 nodes (691)
pscost	2106.3	6971.9
+ filter	1661.4	5199.6
+ score	1617.5	5023.9

Table 4: Comparison of standard pseudocost branching, pseudocost branching with cloud filtering and with cloud scoring on the MMMC test set.

In order to compute the branching score s_j for branching on variable x_j, the objective gains for both child nodes are predicted using the pseudocosts Ψ_j^- and Ψ_j^+. The standard formulas for this are

$$\Delta_j^- = \Psi_j^-(x_j^\star - \lfloor x_j^\star \rfloor) \quad \text{and} \quad \Delta_j^+ = \Psi_j^+(\lceil x_j^\star \rceil - x_j^\star). \tag{4}$$

Again, this assumes a linear increase in the objective function from the current LP value of the branching variable to its new bounds, see Figure 11(b) (shaded lines). A more accurate estimate exploiting interval $I_j(\mathcal{C})$ can be obtained as:

$$\tilde{\Delta}_j^- = \begin{cases} \Psi_j^-(\ell_j(\mathcal{C}) - \lfloor x_j^\star \rfloor) & \text{if } \ell_j(\mathcal{C}) \geq \lfloor x_j^\star \rfloor \\ 0 & \text{else} \end{cases} \tag{5}$$

and

$$\tilde{\Delta}_j^+ = \begin{cases} \Psi_j^+(\lceil x_j^\star \rceil - u_j(\mathcal{C})) & \text{if } u_j(\mathcal{C}) \leq \lceil x_j^\star \rceil \\ 0 & \text{else.} \end{cases} \tag{6}$$

A graphical representation is depicted in Figure 11(b); the cloud pseudocost prediction is printed bold. The following relations between the unit gains and predictions with and without cloud information holds:

Lemma 3.1. *Let $x^\star$ be an optimal solution of the LP relaxation at a given branch-and-bound node and $\mathcal{C}$ a set of cloud points for the LP relaxation. Then*

1. *for fixed $\Delta^\uparrow$ and $\Delta^\downarrow$, it holds that $\tilde{\varsigma}_j^- \geq \varsigma_j^-$ and $\tilde{\varsigma}_j^+ \geq \varsigma_j^+$, respectively;*

2. *for fixed Ψ_j^- and Ψ_j^+, it holds that $\tilde{\Delta}_j^- \leq \Delta_j^-$ and $\tilde{\Delta}_j^+ \leq \Delta_j^+$, respectively.*

Proof. Follows directly from Equations (2)–(6). □

Thus, under the same preconditions, the standard pseudocosts will be an underestimation of the more precise pseudocosts based on the cloud intervals, whereas the objective gain, based on which the branching decision is taken, will be an overestimation. Of course, these quantities interact directly which each other and the preconditions change. Ideally, the bound prediction will be made using cloud information if and only if the previous pseudocost updates used it as well. Then, the cloud pseudocosts will be higher, but they are multiplied with a smaller value to predict the objective gain. Therefore, the average predicted dual bound increase will be similar, but individual predictions will hopefully be more accurate.

In the following, we want to examine how these more accurate predictions impact the performance of pseudocost branching. For this, we performed computational experiments in a setting similar to the previous experiments in this section. Table 4 compares standard pseudocost branching, pseudocost branching with cloud filtering, and pseudocost cloud branching, which filters candidates and uses cloud pseudocost updates and predictions. We hoped that cloud branching helps to make better, more reliable predictions and thereby leads to better branching decisions. Such an effect can be observed in the computational results, but it is smaller than expected. Compared to the filtering variant, the tree size is reduced by only 2.6 %. Both methods suffer from the main weakness of pseudocost branching: the most important branching decisions taken at higher levels of the branch-and-bound tree cannot rely on any useful history information yet. Consequently, the improvement for instances that are solved in less than 100 nodes is marginal, while it increases to 3.4 % for instances which need at least 100 nodes with one variant. These numbers are somewhat underwhelming. It seems that the effects of the randomness of the actual LP solution from which standard pseudocosts suffer averages out when enough pseudocost samples are available. On the other hand, with a small number of samples, both methods are not reliable enough yet. The range in between, where cloud pseudocosts are considerably more reliable than standard pseudocosts seems to be small, and therefore, the effect is limited. Nevertheless, the overall tree size reduction that can be obtained using cloud information for filtering and scoring is still considerable and reaches 23.2 %.

3.5 Full strong cloud branching

Full strong branching [5, 6] predicts the dual bound of potential child nodes by solving auxiliary LPs with the potential branching bound changes added. The score for a branching candidate does not depend on its LP solution value; as long as the value is fractional, strong branching will solve the LPs of the two potential child nodes which gives very accurate predictions of the dual bounds. As a consequence, the knowledge of multiple LP optima does not influence the score of a branching candidate.

Therefore, other than for the previous rules, the enhancements we suggest for full strong branching do not improve the quality of the branching decision being taken. Nevertheless, cloud information can be used to save some strong branching LPs and reduce the effort of the branching rule, while maintaining the high quality of branching decisions. In this sense, the modifications we suggest for strong branching are similar to the cloud filtering, but extend it to single strong branching LPs.

For the full strong branching improvement, the only thing we need to know about a variable is whether there is an integer point in its cloud interval, and which one, if there is. The knowledge of an integer point in the interval proves that branching on this variable will imply no dual bound increase for at least one child. The product score used by SCIP will prefer candidates with a strictly improved dual bound in both child nodes over ones with a zero improvement on at least one side. Thus, we first perform full strong branching on all variables in the set F_0 of candidates with no integer point in their cloud interval. If we can find even one variable in this set with a strictly improved dual bound in both child nodes, we stop and pick the best variable within this set. In this case, we saved the strong branching calls for all variables in $F_1 \cap F(x^\star)$, i.e., $2|F_1 \cap F(x^\star)|$ LPs solves. Up to this point, this is similar to standard cloud filtering.

However, if no variable is found to have a non-zero improvement on both sides, cloud filtering would just select the best one in F_0. It is not clear though that those variables are better than the ones in set F_1. Therefore, we also check the variables in $F_1 \cap F(x^\star)$. For all these variables, we already know that the improvement will be zero on one side, so we only need to solve the auxiliary strong branching LP for the other side. Thus, we save $|F_1 \cap F(x^\star)|$ LPs solves. We compare these candidates to the ones of set F_0 and branch on the one with the highest non-zero improvement in one direction.

If all branching candidates investigated so far cause no increase in the dual bound of any of the child nodes, no candidate from $F(\mathcal{C})$ is able to do so. Therefore, strong branching is not

	> 1 node (881)				≥ 100 nodes (397)			
setting	nodes	fair nodes	SB LPs	SB iters	nodes	fair nodes	SB LPs	SB iters
full strong	181.4	1182.7	4569.9	365191	2524.3	17797.5	59679.9	3228143
+ filter	201.8	1132.0	3743.0	295362	3024.9	17587.4	48014.2	2647576
+ cloud	192.7	1086.4	3016.0	241964	2845.2	16827.6	35153.1	2024276

Table 5: Comparison of standard full strong branching, full strong branching with cloud filtering and full strong cloud branching.

able to effectively differentiate the candidates, and thus, another criterion should be used. In our SCIP implementation, we select a random unfixed variable.

Table 5 shows the results of our computational experiments comparing standard full strong branching (row 1), full strong branching with cloud filtering (row 2), and full strong cloud branching as described above (row 3). In contrast to previous tables, we are not only listing the shifted geometric means of node numbers but also the fair node number [34], see Section 3.1. Additionally, since the effect of our cloud modification is not an improved branching decision, but a reduction of the strong branching effort, we list the shifted geometric means of the strong branching LP solves (column 'SB LPs', shift 100) and the LP iterations spent in strong branching (column 'SB iters', shift 1000).

As described in Section 3.1, cloud filtering increases the node number because of the side-effects of the filtered-out strong branching calls. The proposed full strong cloud branching, on the other side, skips some of these strong branching calls as well, but may also do more than the filter variant, if there is no variable in F_0 with strictly positive improvement (in tolerances) on both sides. Therefore, the cloud variant needs fewer nodes on average than the filter variant, but still 6.3 % more than standard strong branching (12.7 % for instances with larger trees). In the fair node number, that takes into account side-effects of strong branching, the tree size reduction of full strong cloud branching is about twice as large as for the filter variant. For instance that need 100 nodes or more, the reduction is even four times as large. Cloud filtering is able to reduce the number of strong branching LPs being solved by 18.1 % (19.5 % for larger trees). However, the filter alone will sometimes still solve strong branching LPs for which the cloud information already proves that there will be no dual bound improvement. For example, it will perform default full strong branching on all candidates in $F_1 \cap F(x^\star)$ if F_0 is empty. Full strong cloud branching is able to save some additional LPs in such cases: 19.4 % (26.8 % for larger trees) compared to the filter variant. Finally, the strong branching LP iterations are reduced by similar ratios as the calls for cloud filtering. Additionally skipping LPs that will not improve the dual bound is slightly less beneficial in this regard: the improvement over cloud filtering is 18.1 % (23.5 % for larger trees). This result seems reasonable as we expect fewer changes in the LP solution—and thus fewer simplex iterations—if the optimal value of the LP does not change.

Summing up, full strong cloud branching reduces the average number of strong branching LPs and LP iterations by 34 % and 33.7 %, respectively, compared to standard full strong branching.

3.6 Reliability cloud branching

As reliability branching [3] is a combination of pseudocost branching and strong branching (for candidates with not yet reliable pseudocosts), reliability cloud branching applies the individual cloud improvements for these two techniques to the respective components of reliability branching.

	> 1 node (907)				≥ 100 nodes (617)			
setting	nodes	fair nodes	SB LPs	SB iters	nodes	fair nodes	SB LPs	SB iters
reliability	948.8	1583.6	712.7	52802	5856.3	7785.5	1393.4	100867
+ filter	855.0	1425.3	656.9	50173	5059.1	6771.8	1276.0	96576
+ cloud	817.3	1438.1	622.2	47275	4818.1	6775.7	1238.8	95749

Table 6: Comparison of standard reliability branching, reliability branching with cloud filtering and reliability cloud branching.

Since reliability branching is based mainly on pseudocosts and the number of candidates investigated via strong branching is limited, we do the filtering as for pseudocost branching: we prefer candidates from set F_0; only if this is empty, we investigate candidates from $F_1 \cap F(x^\star)$. Finally, if this is empty as well, we use the standard candidate set $F(x^\star)$. In this case, we know already that the cloud interval of each candidate covers the two potential branching values, so strong branching will never identify a dual bound improvement and can be skipped. Additionally, cloud pseudocosts will also not be able to differentiate the candidates. Therefore, we use standard pseudocosts based on the single LP optimum $x^\star$ to select a branching candidate. In principle, we could have decided to use a random selection here as well, but we decided for the standard pseudocosts to stay similar to what standard reliability branching would do and to reduce the effect of performance variability on the comparison. Note that this implies that we have to collect both types of pseudo costs individually during the search process. We avoid mixing them up, as we observed that cloud pseudocosts are typically larger than standard pseudo costs, see Section 3.4

In case the branching variable is selected from F_0 or $F_1 \cap F(x^\star)$, cloud pseudocosts are used to sort the variables by their expected branching score. If we did not collect enough cloud pseudocost samples for a variable yet, we perform strong branching, skipping one of the two strong branching LPs for candidates in F_1. We update both pseudocosts and cloud pseudocosts as well as the branching score of the candidate after strong branching. Note that the reliability branching rule in SCIP will perform strong branching on a variable at most once per node. If the branching rule applies some domain changes afterwards, re-optimizes the node, and enters the branching method again, SCIP assumes that the gains on both sides will stay similar for this variable and save another strong branching LP solve. In this case, we set the expected gain for one potential child to 0 if the cloud interval of the branching candidate proves that there will be no gain for this child.

Again, we performed computational experiments to compare standard reliability branching, a version that used cloud filtering, and a full-scale cloud version of reliability branching as described above. The results are presented in Table 6, which shows the same types of statistics as Table 5. We observe that the additional improvement obtained by reliability cloud branching is smaller than the improvement obtained by cloud filtering alone. Cloud filtering is able to reduce the node and fair node number by about 10 % for instances that are not solved at the root node and 13.6 % for instances that need at least 100 nodes in one variant. The number of strong branching LPs being solved is reduced by about 8 % and the strong branching LP iterations by 5 %, 4.3 % for instances with larger trees. Reliability cloud branching achieves an additional tree size reduction of 4.4 %. The fair node number stays almost the same compared to cloud filtering. The number of strong branching LPs is reduced by another 5.3 % on average over all regarded instances; for the ones with larger trees, this reduction is smaller and amounts to 2.9 %. The smaller effect on larger trees is a result of the reliability scheme: for large-enough trees, reliability branching will perform strong branching on variables as long as their pseudocosts are not reliable. When saving some strong branching calls, the pseudocosts are not updated and thus, strong branching will be performed later again to obtain reliability when standard reliability branching would already have generated enough pseudocost samples for this variable. For the same reason, reliability cloud branching only reduces the strong branching LP iteration count by 0.9 % for instances with larger trees, compared to cloud filtering. For instances solved in a few nodes, strong branching effort saved early is not compensated by additional calls later in the tree, so the LP iteration count is reduced by 5.8 % for the complete set of instances solved by all variants with some branching. Note that the strong branching effort is not necessarily reduced for large trees, but the effort is spent more wisely. The branching rule applies strong branching also later in the tree, where useful, which primarily causes additional side-effects that help to reduce the tree size but slightly increase the fair node number. Overall, reliability cloud branching reduces the tree size by 13.9 % and the strong branching LP iterations by 10.5 %, respectively, compared to standard reliability branching.

3.7 Conclusions

In this section, we considered question (ii) and evaluated the potential of exploiting cloud information within branching rules. We looked at both the tree size and the fair node number as a measure of the quality of the decision as well as a possibly reduced strong branching effort. Computational experiments demonstrated significant improvements with respect to both measures. On the one hand, about one-third of the strong branching LPs could be saved when using cloud information within full strong branching to skip strong branching LPs. The node number increased because of side-effects of the saved strong branching LPs, but the quality of the branching decisions was not worsened as shown by a slight reduction in the fair node number. On the other hand, rules that do not use strong branching to get accurate dual bound predictions could improve their branching decisions through cloud information. The simple filtering of branching candidates reduced the average tree size by more than 20 % for random branching, most infeasible branching, and pseudocost branching. Most infeasible branching could be improved further by maximizing the distance between cloud interval and integer point or minimizing the cloud interval length (cloud diameter branching). For pseudocost branching, we achieved only a slight additional improvement through more exact cloud pseudocosts. When combining the improvements for strong branching and pseudocost branching in reliability cloud branching, a decrease in tree size as well as strong branching effort was obtained. The improvement was not as large as for full strong branching and pseudocost branching individually, which, however, each improve only one of the measures. Still, a reduction of more than 10 % in both measures shows the potential of cloud reliability branching. Which of these improvements can be translated into a solving time reduction, however, strongly depends on the effort needed to generate cloud intervals. This will be discussed in the following.

4 Effective cloud sampling

So far, we used an obbt-style sampling of cloud points which generates the exact cloud intervals but is computationally very expensive. In this section, we describe alternative approaches for generating a cloud of solutions as well as additional improvements to speed up the sampling or extract additional information.

4.1 Focusing on relevant variables

Each optimal LP solution that extends one of the cloud intervals provides additional information, but not all this information is equally useful. First, continuous variables are not used as branching candidates in mixed-integer programming, so we do not need to spend any effort to compute their cloud intervals. Thus, these variables are always assigned an objective coefficient of 0, and we solve no auxiliary LPs that minimize or maximize the variables in the obbt-style sampling.

For integer variables, we distinguish three levels of detail with respect to cloud information. The highest level uses obbt cloud sampling to compute exact cloud intervals for all integer variables. It solves two auxiliary LPs per variable that minimize and maximize the value of that variable. Only if the cloud interval already touches a bound of the variable for the current set of cloud solutions $\mathcal{C}$, i.e., $\ell_j(\mathcal{C}) = \ell_j$ or $u_j(\mathcal{C}) = u_j$, can one or both of these LPs be omitted.

The fast level disregards variables that already have an integer value in their cloud interval, i.e., $|\mathcal{I}_j(\mathcal{C}) \cap \mathbb{Z}| \neq \emptyset$. For obbt cloud sampling, this means that no auxiliary LPs are solved for such variables. This level determines F_0, but cannot correctly distinguish between the sets F_1 and F_2, as it generates imprecise cloud intervals for the variables in those sets.

The medium level disregards an integer variable only if its cloud interval already contains two integer points. Thus, it correctly determines the partition of branching candidates into F_0, F_1, and F_2 and computes exact cloud intervals for all integer variables except those contained in F_2. Again, a variable that is already in F_2 for the current set of cloud solutions is always assigned an objective coefficient of 0, and obbt-style sampling will not solve auxiliary LPs for that variable. Additionally, we can choose to skip all variables that were already integer in the original LP solution $x^\star$. Since we disregard those variables in our branching methods, in any case, there is no need to correctly determine whether they belong to F_1 or F_2 and we can save some effort. Therefore, we skipped those variables in all our experiments.

For the experiments in Sections 2 and 3, the high level was not needed since our analysis of the cloud intervals as well as the described branching methods does not rely on exact cloud intervals for candidates in F_2. On the other hand, several of the proposed branching rules make use of the cloud intervals for candidates in F_1 and distinguish candidates in F_1 from those in F_2. Therefore, the fast level was not sufficient for the detailed analysis and the experiments were performed with the medium level of obbt sampling which provides all information needed to assess the full potential of cloud branching.

In the following, we present two alternatives and how the levels of detail translate to them.

4.2 Random cloud sampling

The first variant that limits the number of cloud sampling LPs uses random objective functions. We implemented this as follows: the variable set is partitioned by randomly assigning each variable to one of k subsets, where k can be controlled by a parameter. For each subset, the objective coefficients of all variables not contained in the subset are set to 0, while all variables in the set are randomly assigned an objective coefficient of -1 or $+1$ with equal probability. Then, we solve two auxiliary LPs, minimizing and maximizing the random objective function. This procedure is repeated for all subsets in the partition.

In line with our argumentation in the previous subsection, we partition the set of fractional variables in the original LP solution $x^\star$ rather than the set of all variables. After that, we set the objective coefficients of all variables in F_2 or $F_1 \cup F_2$ to 0 for the medium or fast level of detail, respectively.

4.3 Pump cloud sampling

A fast variant that aims at driving variables towards integrality is motivated by the feasibility pump [36] primal heuristic. It uses an objective function in which the current LP point is rounded and a Hamming distance function is generated to move to a different point. More precisely, given a fractional solution $x^\star$, the objective function coefficient c_j of an integer variable x_j is defined as

$$c_j = \begin{cases} 1 & \text{if} \quad 0 < f_j < 0.5 \\ -1 & \text{if} \quad 0.5 \leq f_j < 1 \\ 0 & \text{otherwise} \end{cases}$$

where $f_j = x^\star_j - \lfloor x^\star_j \rfloor$ is the fractional part of $x^\star_j$. The restricted LP is reoptimized with this new objective function, cloud intervals are updated, and the process is iterated, using the new LP optimum as $x^\star$. If at a given iteration, the update did not change any cloud interval, we stop. Alternatively, we can already stop if no cloud interval contains an additional integer point after the update. In the fast or medium level of cloud sampling, the objective coefficients of variables that are already contained in $F_1 \cup F_2$ or F_2, respectively, are always set to 0 and pump cloud sampling stops if only cloud intervals of such variables were extended with the last cloud solution.

This cloud sampling approach is related to the pump-reduce procedure that CPLEX performs to achieve more integral LP optima [20]. As such, it typically creates similar cloud solutions that are more and more integral but misses to construct diverse solutions that generate (close to) exact cloud intervals. However, it is successful in generating cloud intervals that contain one integer point, and this is all that is needed for the filtering of branching candidates based on cloud information. Since the results presented in the last section indicate that this already provides a large portion of the potential improvement that we can obtain by exploiting cloud information, pump cloud sampling seems to be a good compromise between effort and benefit.

4.4 Cloud LP preprocessing

When restricting the LP to its optimal face, all non-degenerate variables are fixed to one of their bounds. These changes may imply bound changes or even fixings of other variables. We implemented a preprocessing routine for the cloud LP that applies a simple domain propagation step. Given a constraint $A_i.x = b$, we compute the minimum and maximum activity $\underline{\alpha}$ and $\overline{\alpha}$ as

$$\underline{\alpha} = min\{A_i.x \mid \ell \leq x \leq u\} = \sum_{j \in N : a_{ij} > 0} a_{ij}\ell_j + \sum_{j \in N : a_{ij} < 0} a_{ij}u_j \in \mathbb{R} \cup \{-\infty\} \quad (7)$$

$$\overline{\alpha} = max\{A_i.x \mid \ell \leq x \leq u\} = \sum_{j \in N : a_{ij} > 0} a_{ij}u_j + \sum_{j \in N : a_{ij} < 0} a_{ij}\ell_j \in \mathbb{R} \cup \{\infty\}. \quad (8)$$

Note that $\underline{\alpha} \leq b \leq \overline{\alpha}$ holds since the original LP solution lies on the optimal face and thus, the restricted LP is feasible. Now, if $\underline{\alpha} = b$, all variables with non-zero coefficient in this constraint can be fixed to the value with which they appear in the minimum activity computation (7), i.e., $x_j = \ell_j$ for $j \in N$ with $a_{ij} > 0$ and $x_j = u_j$ for $j \in N$ with $a_{ij} < 0$. Analogously, if $\overline{\alpha} = b$, for $j \in N$ and $a_{ij} \neq 0$, x_j can be fixed to u_j if $a_{ij} > 0$ and to ℓ_j if $a_{ij} < 0$. This procedure is applied to each constraint, and the whole process is iterated until no more fixings were identified.

Each variable fixed that way does not need to be regarded in the cloud sampling process and reduces the size of the auxiliary LPs that are solved to obtain cloud information. In some cases, this even allows to fix all degenerate non-basic variables which proves that no alternative optimal LP solutions exist. This is the most beneficial case since it allows to save some setup effort for the cloud sampling LPs and at least one solve of an auxiliary LP. For obbt-style sampling, it may even save two LPs per basic variable plus one for each of the fixed variables.

Note that many more bound changes could be derived from the reduced LP by standard domain propagation techniques like activity-based bound tightening [37]. We restrict the preprocessing to this simple approach because it is fast and all reductions help directly to reduce the sampling effort. Our experiments with full activity-based bound tightening showed that this is often more time-consuming since it does many more iterations in which some bounds are tightened, but no variables fixed. Although this may lead to fixings at some point, a reduced domain alone provides no benefit to the cloud sampling procedure. As a result, the additional effort for full activity-based bound tightening did not pay off, so we stayed with the simple approach.

4.5 Rounding of cloud solutions

During cloud solution sampling, multiple optimal solutions to the current LP are examined. We only store the cloud intervals to avoid a large memory overhead, which is sufficient for the branching rules based on cloud information that we presented. Since the solutions are not stored, they are currently not used for primal heuristics, e.g., as a starting point for diving or large neighborhood search heuristics. We see this as an interesting direction for further research, but this is not in the scope of this paper. Right now, we only apply a simple rounding heuristic [4] to each cloud solution directly after solving the auxiliary LP, which is in line with what SCIP applies to each strong branching LP solution, see [38]. If this solution has a value no worse than the LP optimum, we stop cloud sampling and prune the node.

4.6 Computational evaluation

We performed a computational evaluation of the cloud sampling variants as well as the further improvements. For this, we ran all instances of the MMMC test set with standard reliability branching and a time limit of five hours. Different cloud sampling variants were performed one after the other before each branching call. The results of the sampling calls were discarded so that the next sampling variant started from scratch. However, feasible solutions obtained by rounding the sampling LP solutions were buffered, and the best one over all variants was added after sampling. Before starting the sampling process, we applied cloud LP preprocessing once.

The results are summarized in Table 7. The first six rows show statistics for the three sampling variants obbt, random (with $k = 8, 4, 2, 1$), and pump with the medium level of detail, the second set of rows shows the same information for the fast sampling level. We omit the high level since it does not provide any improvement over the medium one in our branching methods.

The first column specifies the sampling variant, followed by the column 'effort', which shows the relative effort for the sampling process. For each instance, we divide the cloud sampling time (including preprocessing) by the LP time on these instances, where the latter is given by the sum of node and strong branching LP time. The next column shows the average number of 'LPs' solved per sampling call, followed by the number of initial branching candidates with a non-trivial cloud interval ('succ'). Column 'size' shows the average size of the cloud intervals computed by the respective sampling variant, compared to the exact size computed by obbt with the medium level of detail. For integer variables, we limit the size of the cloud intervals by 1 in these computations, as larger cloud intervals are not of interest and may not be computed correctly by obbt sampling. The next two columns list the average number of branching candidates that have at least one or at least two integer points in their cloud interval. Column 'candred' shows the candidate reduction in percent, i.e., the relative number of LP branching candidates with an integer point in their cloud interval. Finally, column 'sols' lists the share of instances where

level of detail		Sample variant	effort	LPs	succ	size	1int	2int	candred	sols
	medium	obbt	3.52	125.0	40.2	100.0%	30.4	11.6	26.2%	19.3%
		random (k=8)	1.24	14.2	40.0	93.6%	29.9	9.1	25.7%	17.9%
		random (k=4)	0.92	7.6	39.7	88.6%	29.3	7.7	25.2%	18.6%
		random (k=2)	0.68	3.9	39.2	80.2%	28.3	5.8	24.2%	15.5%
		random (k=1)	0.58	2.0	38.4	68.7%	26.4	4.0	22.5%	14.8%
		pump	0.76	4.3	32.5	36.8%	24.3	2.4	20.2%	18.6%
	fast	obbt	1.64	78.0	40.2	72.9%	30.4	1.5	26.2%	16.7%
		random (k=8)	0.72	11.5	39.9	70.6%	29.9	1.6	25.7%	17.7%
		random (k=4)	0.60	6.7	39.8	68.9%	29.5	1.4	25.4%	17.9%
		random (k=2)	0.52	3.8	39.3	66.1%	28.6	1.2	24.4%	17.2%
		random (k=1)	0.48	2.0	38.3	60.1%	26.6	0.9	22.7%	15.0%
		pump	0.43	2.5	32.5	32.8%	24.3	0.9	20.3%	14.8%

Table 7: Cloud sampling statistics.

rounding a cloud sampling LP solution provided an improving solution at least once. All other numbers are averaged over the multiple calls for one instance by arithmetic mean and over the 419 instances where sampling was performed at least once by a shifted geometric mean. Since the relative 'size' of cloud intervals only makes sense if obbt sampling was successful at least once on an instance, this column is aggregated over those 378 instances only. We use a shift of 0.1 for the effort and LP calls, and 1 (or 1 % where this applies) for the remaining columns.

As expected, obbt cloud branching solves a high number of sampling LPs and even though these LPs have only one non-zero objective coefficient and are therefore easier to solve than the LPs of the other variants, the effort for obbt sampling is too high for practical uses. We observe that in terms of sampling success and branching candidate filtering, pump sampling performs worst of all variants and obbt sampling performs best. Random sampling is more successful as the number k of subsets is increased, solving more sampling LPs with less dense objective functions. For the same level of detail, spending more effort for sampling typically leads to higher success rates. The only exception is pump sampling in the medium level of detail. It spends more time than random sampling with 1 or 2 subsets because it solves more LPs, but typically searches in only one direction which leads to smaller success rates than the other variants. On the other hand, it is is quite successful in terms of constructing feasible solutions, finding solutions for 18.6 % of the instances, which is only topped by obbt sampling which finds solutions for 19.3 % of the instances. A closer look into this, however, shows that on those instances where solutions are found, pump sampling only finds 1.6 solutions per instance, while all other variants find 2.1 to 2.2 solutions per instance on average.

Success rates, the share of variables with an integer value in their cloud interval, as well as the branching candidate reduction are not considerably influenced by the level of detail applied in sampling. This result proves that the fast level of detail works as intended and is able to generate this information with reduced effort. Note that these numbers are even slightly higher with the fast level for some variants of random and pump sampling. This increase results from sparser objective functions in later sampling LP calls of these variants as they set the objective coefficients of variables that are already in F_1 to 0.

The relative size of the cloud intervals compared to the exact size that obbt computes in the medium level, however, suffers from the fast sampling version and is reduced by 11 % to 27.1 %

(with higher reductions the more successful a variant is compared to the others). As a result, also the number of candidates with two integer values in their cloud interval is reduced significantly. In the fast level, obbt sampling and random sampling with 8 subsets are most successful with 1.5 and 1.6 such candidates being identified, respectively. In the medium level, even the least successful variant pump identifies 2.4 such candidates and random and obbt sampling increase this number up to 11.6.

The variant to choose depends on the particular branching rule that is used, in particular, how much effort the rule spends per node and whether this can be reduced by cloud information. If the rule profits significantly from the knowledge of variables with two integer values in their cloud interval, the medium level of detail should be used. Out of the variants there, random sampling with 2 subsets seems to be a reasonable choice. On average, it constructs nontrivial cloud intervals for 97.6 % of the variables with such an interval and generates cloud intervals that have on average 80.2 % of the exact size. The number of branching candidates can be reduced by 24.2 %, and 5.8 variables contain two integer values on average. Compared to random sampling with only one subset, the effort is increased by only 15.2 %, while increasing the average number of candidates with one or two integer points in their cloud interval by 1.9 and 1.8, respectively. Going further up to four subsets leads to an increase of these numbers by only 1 and 1.9, while the effort goes up by 35.5 %.

With the fast sampling level, random sampling with two or four subsets both improve significantly over pump sampling and random sampling with one subset while the effort increases moderately. Depending on the application, both might be good choices.

Finally, cloud LP preprocessing fixed at least one column for 371 instances. On those instances, the preprocessing time amounts for 4.8 % of the LP time on average (shifted geometric mean, shift 1 %). On the instances where cloud sampling was performed, but propagation was never successful, the average preprocessing effort is only 0.9 % of the LP time. On the former instances, 3.2 preprocessing rounds are performed on average per sampling call, and 14.3 variables are fixed on average in each of the calls. The number of unfixed variables in the cloud LP is reduced by 5.4 % on average; 1.3 sampling calls could even be skipped on average because all dual degenerate variables were fixed. There are six additional instances where preprocessing was performed although no single sampling LP was solved, showing that preprocessing fixed all dual degenerate variables in all potential sampling calls.

4.7 Conclusions

In this section, we discussed different cloud sampling variants and two improvements of the sampling process in order to provide an answer to question (iii) posed in the introduction. The improvements comprise a preprocessing step and the rounding of cloud LP solutions, which both proved to be beneficial in computational experiments. The results verified that the obbt-style sampling we used in the previous section to demonstrate the potential of using cloud information within branching rules is too time-consuming for practical purposes. On the other hand, the pump cloud sampling used in the previous publication on cloud branching [17] was outperformed in computational experiments by a random sampling of cloud points. As an answer to question (iii), we thus suggest using the random sampling procedure. Depending on the branching rule and how it can be improved by cloud information, either the fast or medium level and two or four random objective functions being both minimized and maximized seem to be a good compromise between effort and sampling accuracy.

5 Computational experiments

In this section, we investigate the impact of exploiting degeneracy in branching rules on the overall solving process. As in the previous sections, we use SCIP 5.0.1 [4, 21] with SoPlex 3.1.1 [29, 21] as underlying LP solver. Except for the changes to the investigated branching rules, we use default settings, in particular, we do not provide the optimum as cutoff bound and let SCIP both find the optimal solution and prove its optimality. Since one of the most important performance criterions is the running time, we need to find a compromise between the effort spent on the sampling of cloud points and the accuracy of cloud intervals. This compromise may look different for different branching rules.

All computational experiments in this section are performed on the MMMC test set with five different random seeds (one of them being the default seed). In a slight abuse of notation, we are referring to each of the 2480 instance/seed combinations as an individual instance in the following. Instance/seed combinations on which one of the variants failed, e.g., due to constraints being slightly violated by the optimal solution, or inconsistent optimal values, were excluded from the evaluation.

For each experiment, the instance set is then divided into subsets as follows. The set oneopt contains all instances that were solved to optimality by at least one of the variants. By excluding instances that no variant solved to optimality within the time limit, we get a better picture of the relative performance of the variants than for the whole set, where such instances dampen the difference. The hard subset restricts the oneopt set to only those instances for which one variant needed at least 100 seconds. By excluding easy instances that all variants solve fast, this subset focuses on the most interesting instances on which improvement is most important.

The allopt set contains instances that both variants solved to optimality. It allows comparing the tree size needed to prove optimality. A subset of this is the affected set which contains only instances that have a different solving path with both variants. As a proxy to determine these instances, we compare the number of dual LP iterations for node LPs (not counting strong branching or cloud sampling LPs).

Finally, we regard two subsets MMM and Cor@l. Both sets restrict the allopt set of instances solved by both variants to the instances contained in the set given by the subset name, where MMM is the union of MIPLIB 3 [25], MIPLIB 2003 [26], and the MIPLIB 2010 benchmark set [15]. Note that these two sets are not disjoint.

The tables comparing two variants are then split into two sets of columns, one set for each variant. For each variant, column 'opt' lists the number of instances solved to optimality within the time limit of two hours. Column 'time' gives the shifted geometric mean of the running time with a shift of 1 second and 'nodes' the number of nodes in shifted geometric mean with a shift of 10. Finally, 'SBs' presents the shifted geometric mean of the number of strong branching LPs being solved, using a shift of 100. Each row presents these numbers for one of the different subsets of instances, as denoted in the first column. The number put in parentheses behind the subset name is the number of instances in the subset.

Note that we cannot provide the fair node number for the following experiments, as it requires a fair parameter setting that disables some features and provides the optimum as cutoff bound. While the experiments presented in Section 3 used such a setting to focus on the branching performance, the aim of this section is different. It regards the performance impact of the branching rules in a realistic environment that uses default settings and needs to find an optimal solution first before proving its optimality. Thus, we explicitly want to make use of potentially beneficial side-effects of strong branching to achieve the best performance possible.

	full strong branching				full strong cloud branching			
Subset	opt	time	nodes	SBs	opt	time	nodes	SBs
oneopt (1382)	1320	72.1	125.5	2941.3	1355	62.8	120.0	2029.2
hard (691)	629	829.9	516.4	21616.4	664	672.3	487.6	13290.9
allopt (1293)	1293	53.2	118.9	2484.0	1293	47.7	112.9	1800.7
affected (941)	941	132.9	297.9	8401.9	941	114.4	278.4	5494.4
MMM (520)	520	52.5	184.4	3287.2	520	51.7	185.5	2972.7
Cor@l (813)	813	60.9	103.2	2368.6	813	51.4	94.3	1484.3

Table 8: Comparison of standard and cloud full strong branching on different subsets of the MMMC test set: number of solved instances, shifted geometric means of solving time, node number, and strong branching LPs.

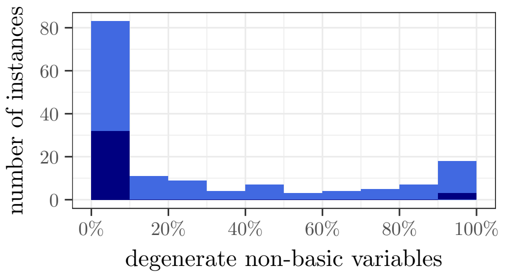
5.1 Full strong branching

Full strong branching solves two auxiliary LPs for each branching candidate at each node. Since full strong cloud branching profits both from an accurate approximation of both F_1 and F_2 and the effort that full strong branching spends per node is already relatively large, we chose random sampling with $k = 2$ subsets and the medium sampling level. This method solves at most four cloud LPs at each node to generate cloud information which can often prove that many more variables with fractional value in the original LP solution have one or even two integer values in their cloud interval, see Section 4.6. Even though the random cloud sampling LPs are typically harder to solve than strong branching LPs, this shows that there is potential for an improvement of the overall performance of full strong branching.

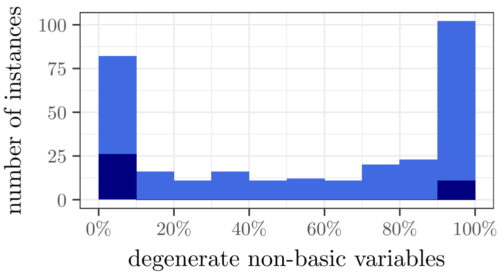
We compare this variant of full strong cloud branching with standard full strong branching, as implemented in SCIP. Note that we apply standard strong branching without domain propagation in both variants. In the cloud variant, we perform cloud LP preprocessing and try to round cloud solutions. The computational experiments were performed on cluster 2, see Section 3.1 for the specifications. At each compute node, only one job was executed at a time to ensure reliable time measures.

The results are summarized in Table 8. With cloud information, 35 additional instances are solved to optimality. On the oneopt subset, we observe a 13 % time reduction that results from a reduction in the number of strong branching LPs by 31 %. For hard instances, these reductions increase to 19 % and 38.5 %. We can also observe reductions in the node number.

The allopt subset allows to measure this effect more precisely: on this set, the shifted geometric mean of the tree size is reduced by 5 %. This node reduction is partly caused by primal solutions found through rounding of cloud solutions, but also due to an implementation detail of full strong branching in SCIP. Whenever it is called another time at the same node, it re-uses strong branching results of the variables that it already performed strong branching on in one of the previous calls at that node. If full strong branching finds infeasible sub-problems that result in a bound change being added at the current node, the next call of the branching rule assumes that the problem did not change too much and the dual bound improvements observed before are still reasonable predictions. From a performance viewpoint, the effort for recomputing these predictions would probably outweigh the potential improvement in prediction accuracy. While this improves the performance of the full strong branching rule in practice, we disabled it in previous full strong branching experiments to focus on the best branching decision full strong



(a) MMM test set



(b) COR@L test set

Figure 12: Dual degeneracy rates at the root node for MMM and COR@L test set.

branching could take. Now, since we are interested in performance, we kept this enabled for our runs. As a consequence, cloud information does even improve the branching decision now, as it may rule out candidates that showed good dual bound improvements in previous calls at that node. Overall, using cloud information in full strong branching affects most of the instances solved by both variants, 72.8 % of them change their solving path. On those instances, the solving time is reduced by 14 %.

When looking at the origins of our test instances, we observe that the cloud improvement is more substantial on instances originating from the COR@L test set than on MIPLIB instances. On the former, the shifted geometric mean of the solving time is reduced by 15.5 %, while on the latter, the solving time stays almost the same. The reason for this is that degeneracy is much more common in instances of the COR@L test set than for MIPLIB instances. Figure 12 illustrates this. It shows histograms of degeneracy rates similar to the one presented in Figure 2 but restricted to MIPLIB instances (left side) and COR@L instances (right side). The MMM test set, although less than half as large as the COR@L set, contains more instances with a degeneracy rate of no more than 10 % at the root node than the COR@L set. Only 18 MIPLIB instances have a degeneracy rate of more than 90 % at the root node, while this applies to 102 COR@L instances. Consequently, the potential improvement that can be obtained by cloud information is much smaller for the MMM test set and rarely makes up for the effort spent on cloud computation. Thus, we observe only a very small improvement in solving time of 1.7 % on the MMM test set.

5.2 Hybrid branching

Finally, let us consider hybrid methods like reliability branching or hybrid branching [8], the default branching rule of SCIP. These branching rules limit the strong branching effort considerably in order to achieve better performance. Hybrid branching, for example, needs on average more than twice as many nodes as full strong branching to solve an instance to optimality while reducing the average number of strong branching calls by more than a factor of six (on the allopt subset of the MMMC test set). Most of the time, trading tree size for reduced strong branching effort pays off and thus, hybrid branching is able to reduce the average time to optimality by more than a factor of two.

For reliability cloud branching, this means that cloud sampling is called more often and its effort per node weights higher than for full strong branching. In Section 4.6, we saw that the fastest sampling methods increase the effort for LP solving by about a half for reliability branching. On the other hand, the improvement we observed for reliability branching in Section 3.6 is much smaller than the one for full strong branching (see Section 3.5) and depends more on exact cloud intervals, which the fast sampling methods do not deliver. As a result, the potential savings obtained by cloud information for hybrid branching methods are currently outweighed by the effort for our sampling methods.

Nevertheless, our investigations of degeneracy inspired a modification of hybrid branching that uses degeneracy information. Hybrid branching uses pseudocosts and strong branching if the pseudocosts are still unreliable, same as reliability branching. It combines this, however, with inference, cutoff, and conflict information; see the original paper [8] and [4] for more details. A score is computed based on each of these criterions, and for each criterion, all scores are normalized by dividing by the average score over all variables. What is important to notice is that the different scores for one variable are combined with a weighted sum, where dual bound improvement (as predicted by strong branching or pseudocosts) is weighted most (with a factor of 1). Conflict information is added with a weight of 0.01, and both cutoff and inference get a weight of 0.0001. As a result, inference, cutoff, and conflict information mainly work as a tie-breaker in case of very similar pseudocosts.

Now, if the exact dual bound improvements—as strong branching would compute them—of all candidates at the current node were all very similar, they would provide little benefit for the selection of a branching variable. Since pseudocosts are based on historical information and, at other nodes, there might have been differences in the dual bound improvements among the variables, using pseudocosts could rather misguide the branching rule at the current node. Of course, if we would know all dual bound improvements, we would use those, rather than the pseudocosts, but computing them is very time-consuming, as full strong branching proves.

However, we observed before that a high dual degeneracy implies that with a high probability, there exist alternative optimal LP solutions such that each branching candidate has an integer value in at least one of these solutions, see Section 2.1, in particular, Figure 7. In such a case, the dual bound improvement of at least one of the two potential child nodes is zero for each of the candidates. There might be some dual bound improvements for the second child nodes, but it seems reasonable to look at other criterions in this situation. By this, we can aim at selecting a branching variable that provides an improvement for the first child as well, if not in dual bound, then in variables being fixed by domain propagation or moving closer to an infeasible problem. If the dual bound improvements for all candidates would be known, the product score that is used to combine the improvements in both potential child nodes would lead to a similar behavior: the dual bound scores would be close to zero for all variables and thus, inference, cutoff, and conflict scores would become important as tie-breakers.

We propose a variant of hybrid branching that follows this argument. At each call of hybrid branching, we first check the dual degeneracy of the current LP. We get the two measures for degeneracy we discussed in Section 2, the share $\alpha \in [0,1]$ of degenerate non-basic variables and the variable-constraint ratio $\beta \geq 1.0$ of the optimal face. Motivated by our observations in Section 2.1, we compute factors ψ and ω based on these measures as follows:

$$\psi = \begin{cases} 10^{10(\alpha-0.7)} & \text{if } \alpha \geq 0.8 \\ 1 & \text{else} \end{cases} \tag{9}$$

and

$$\omega = \begin{cases} 10\,\beta & \text{if } \beta \geq 2.0 \\ 1 & \text{else.} \end{cases} \tag{10}$$

After that, the product of these two factors is used to change the weights for dual bound improvement (divided by the product) and inference, cutoff and conflict scores (multiplied by the product). A similar adjustment of the weights is already implemented in SCIP based on the ratio between nodes pruned due to infeasibility and those pruned based on the cutoff bound. As a result, as soon as the degeneracy share α is at least 80 % or the variable-constraint ratio β is 2.0 or higher, conflict information is weighted at least as much as dual bound improvement. If those degeneracy measures increase, the impact of the dual bound gets smaller and may even fall behind cutoff and inference score.

Additionally, we even disable strong branching completely at the current node if $\psi \cdot \omega$ is 10 or larger, i.e., if any of the two conditions mentioned before is fulfilled. The motivation for this is that other measures are more important now than the dual bound improvement so that the additional effort for strong branching can be saved. On the other hand, the strong branchings saved now will potentially be performed at a later node where the LP solution is less degenerate if the pseudocosts of the variables did not become reliable until then. Thus, the benefit of this modification is not necessarily the time saved in strong branching, but also that the time for strong branching is spent where it pays off more because the strong branching information is more useful and also provides a better initialization of the pseudocosts.

We included this logic into the default hybrid branching rule of SCIP and performed computational experiments to evaluate the impact on performance. They were performed on cluster 1 (see Section 3.1 for the hardware specifications), running only one job per cluster node at a time. Aggregated results over different subsets of the MMMC test set and five random seeds are presented in Table 9.

We observe that this adjustment of hybrid branching improves performance considerably. Overall, it helps to solve 18 more instances, while reducing the shifted geometric mean of the solving time by 8.6 % on the oneopt subset. The number of strong branchings is reduced by 39.6 % while the node number does not change much. On the hard instances, we can even see a time reduction by 13.3 % and the shifted geometric mean of the number of strong branching calls is more than halved.

Subset allopt allows us to compare nodes accurately since no time outs are contained in this set. Here, we observe a slight node increase by 4.4 % and a time reduction of 6.3 %. The node increase is remarkably small, in particular, when taking into account that the number of strong branching calls is reduced by 34.7 %, resulting in fewer side effects that "hide" nodes from the evaluation.

Since the change in the branching rule becomes relevant only if the degeneracy of at least some node LPs is very high for an instance, it has an impact on only 30.8 % of the instances that both variants solved to optimality. On those instances, however, it improves performance significantly: the number of strong branching calls is reduced by 73.6 % while the node number increases only by 14.9 % in return. Overall, this leads to a reduction of the shifted geometric mean of the solving time by 19.4 %.

When considering the split into MMM and COR@L test set, the higher degeneracy rates lead to a higher impact on the latter set. On the MMM test set, we observe a slight speedup of 0.9 %, while the speedup on the COR@L test amounts to 10 %.

6 Conclusions and outlook

In this paper, we discussed dual degeneracy in mixed-integer programs and how branching rules can exploit it. We identified three essential questions in this context and devoted Sections 2–4 to providing answers for those questions. The first question was answered by showing that dual degeneracy is very common in MIP instances and the average degree of degeneracy stays almost constant during the branch-and-bound search. Degeneracy information is captured by sampling

	hybrid branching				degeneracy-aware hybrid branching			
Subset	opt	time	nodes	SBs	opt	time	nodes	SBs
oneopt (1747)	1711	50.1	587.8	684.4	1729	45.8	602.5	413.5
hard (717)	681	644.8	5175.6	2298.4	699	558.9	5169.9	1070.2
allopt (1693)	1693	43.3	511.4	620.8	1693	40.6	534.0	405.4
affected (521)	521	113.0	1268.8	1313.8	521	91.1	1457.9	346.5
MMM (682)	682	48.0	909.7	731.2	682	47.6	933.4	591.7
Cor@l (1071)	1071	44.8	422.0	604.8	1071	40.8	447.0	346.8

Table 9: Comparison of hybrid branching and degeneracy-aware hybrid branching on different sub-sets of the MMMC test set: number of solved instances, shifted geometric means of solving time, node number, and strong branching LPs.

a cloud of alternative LP optima and storing for each variable the cloud interval, i.e., the smallest interval containing all values this variable takes in the cloud of solutions.

The second question was concerned with ways to exploit degeneracy in branching rules by using cloud information. We introduced a new branching rule called cloud diameter branching that is based solely on the cloud interval of the branching candidates. Thus, it can best be compared with most infeasible branching, compared to which it more than halves the average tree size on instances that need no less than 100 nodes. On the other hand, also most of the common branching rules can be improved using cloud interval information. On models that need at least 100 nodes with the respective branching rule, we observed a 37.1 % tree size reduction for most infeasible branching, 27.9 % for pseudocost branching, and 17.7 % for reliability branching. On the other hand, the average number of strong branching calls is reduced by 41.1 % for full strong branching and 11.1 % for reliability branching.

While these results prove the large potential of cloud-based branching rules, they were performed using exact cloud intervals generated by an expensive sampling procedure. Thus, we considered the third essential question: how can a cloud of LP optima be generated efficiently? We discussed alternative sampling variants and general improvements and concluded that a sampling procedure that minimizes and maximizes multiple random objective functions, applies preprocessing to the cloud LP, and tries to round cloud solutions shows very promising results. In case of full strong branching, we were able to reduce the shifted geometric mean of the overall solving time by 19 % on hard instances by minimizing and maximizing two random objective functions during sampling.

For the default branching rule of SCIP, hybrid branching, generating a cloud of solutions is too time-consuming with our current methods. However, we developed a variant of hybrid branching that adjusts the algorithmic behavior based on the degeneracy present in the current LP solution. For measuring degeneracy, we used a combination of the variable-constraint ratio, which we introduced in this paper, with the share of degenerate non-basic variables. If the LP solution is too degenerate at a node, strong branching is disabled, and focus is switched more to other effects of branching like changed variable domains and infeasibilities. On affected instances, this results in a speedup of 24.1 % for hybrid branching.

The presented results are encouraging for further research on degeneracy and cloud branching. Faster cloud sampling procedures could make cloud variants of reliability or hybrid branching viable; we showed that there is significant potential to improve the branching decisions. Exploiting the knowledge of cloud points generated for branching in other components of the solver could also help to justify the effort for the sampling procedure. An obvious choice would be primal heuristics, which are often called after branching and use the same LP solution as branching does. We believe that both diving and large neighborhood search heuristics could profit from the cloud. The methods described in this paper use multiple optima from a single relaxation as cloud set. In particular in the context of MINLP, employing optima from *multiple, alternative relaxations* seems promising. Finally, our hybrid branching results prove that proper measures for the degeneracy of the current solution alone can be used to adjust algorithmic behavior and improve performance. The combination of two degeneracy measures that we used to improve hybrid branching has already been adapted for recent work on local rapid learning [**?**] and will certainly prove useful in other applications as well.

Acknowledgments

The work for this article has been conducted within the *Research Campus Modal* funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM).

References

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

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

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

[4] Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universität Berlin (2007) http://vs24.kobv.de/opus4-zib/frontdoor/index/index/docId/1018.

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

[6] Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, USA (2007)

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

[8] Achterberg, T., Berthold, T.: Hybrid branching. In van Hoeve, W.J., Hooker, J.N., eds.: 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., Springer (2009) 309–311

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

[10] Karamanov, M., Cornuéjols, G.: Branching on general disjunctions. Mathematical Programming **128**(1-2) (2011) 403–436

[11] Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Proc. of CP, Autriche, Springer (1997) 342–356

[12] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference (DAC '01). (2001) 530–535 doi:10.1145/378239.379017.

[13] Kılınç Karzan, F., Nemhauser, G.L., Savelsbergh, M.W.P.: Information-based branching schemes for binary linear mixed-integer programs. Mathematical Programming Computation **1**(4) (2009) 249–293

[14] Fischetti, M., Monaci, M.: Backdoor branching. In Günlük, O., Woeginger, G.J., eds.: Integer Programming and Combinatoral Optimization - 15th International Conference, IPCO 2011, New York, NY, USA, June 15-17, 2011. Proceedings. Volume 6655 of Lecture Notes in Computer Science., Springer (2011) 183–191

[15] Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010 – Mixed Integer Programming Library version 5. Mathematical Programming Computation **3**(2) (2011) 103–163 http://miplib.zib.de.

[16] Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. In: Theory Driven by Influential Applications. INFORMS (2013) 1–12

[17] Berthold, T., Salvagnin, D.: Cloud branching. In Gomes, C., Sellmann, M., eds.: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Berlin, Heidelberg, Springer Berlin Heidelberg (2013) 28–43

[18] Orchard-Hays, W.: Evolution of linear programming computing techniques. Management Science **4**(2) (1958) 183–190

[19] Chvatal, V.: Linear programming. Macmillan (1983)

[20] Achterberg, T.: LP relaxation modification and cut selection in a MIP solver (June 11, 2013) US Patent 08463729.

[21] Gleixner, A., Eifler, L., Gally, T., Gamrath, G., Gemander, P., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Serrano, F., Shinano, Y., Viernickel, J.M., Vigerske, S., Weninger, D., Witt, J.T., Witzig, J.: The scip optimization suite 5.0. Technical Report 17-61, ZIB, Takustr. 7, 14195 Berlin (2017)

[22] Fischetti, M., Lodi, A., Monaci, M., Salvagnin, D., Tramontani, A.: Improving branch-and-cut performance by random sampling. Mathematical Programming Computation **8**(1) (2016) 113–132

[23] Bajgiran, O.S., Cire, A.A., Rousseau, L.M.: A first look at picking dual variables for maximizing reduced cost fixing. In Salvagnin, D., Lombardi, M., eds.: Integration of AI and OR Techniques in Constraint Programming, Springer International Publishing (2017) 221–228

[24] Achterberg, T.: Exploiting Degeneracy in MIP. Presentation slides from Aussois Workshop 2018. www.iasi.cnr.it/aussois/web/uploads/2018/slides/achterbergt.pdf (2018)

[25] Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.: An updated mixed integer programming library: MIPLIB 3.0. Optima (58) (1998) 12–15 http://miplib.zib.de/miplib3/miplib.html.

[26] Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters **34**(4) (2006) 1–12 http://miplib.zib.de/miplib2003/.

[27] COR@L: MIP Instances (2010) http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/.

[28] Czyzyk, J., Mesnier, M., Moré, J.: The NEOS server. Computational Science & Engineering, IEEE **5**(3) (1998) 68–75 http://www.neos-server.org/neos/.

[29] Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. PhD thesis, Technische Universität Berlin (1996)

[30] Zamora, J.M., Grossmann, I.E.: A branch and contract algorithm for problems with concave univariate, bilinear and linear fractional terms. Journal of Global Optimization **14** (1999) 217–249 doi:10.1023/A:1008312714792.

[31] Caprara, A., Locatelli, M.: Global optimization problems and domain reduction strategies. Mathematical Programming **125** (2010) 123–137 doi:10.1007/s10107-008-0263-4.

[32] Gleixner, A.M., Berthold, T., Müller, B., Weltge, S.: Three enhancements for optimization-based bound tightening. Journal of Global Optimization **67**(4) (2017) 731–757

[33] Hendel, G.: IPET interactive performance evaluation tools. https://github.com/GregorCH/ipet

[34] Gamrath, G., Schubert, C.: Measuring the impact of branching rules for mixed-integer programming. In: Operations Research Proceedings 2017. (2018) 165–170

[35] Mehlhorn, K., Sanders, P.: Algorithms and Data Structures. Springer-Verlag Berlin Heidelberg (2008)

[36] Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming **104**(1) (2005) 91–104 doi:10.1007/s10107-004-0570-3.

[37] Brearley, A.L., Mitra, G., Williams, H.P.: Analysis of mathematical programming problems prior to applying the simplex algorithm. Mathematical Programming **8** (1975) 54–83

[38] Gamrath, G.: Improving strong branching by domain propagation. EURO Journal on Computational Optimization **2**(3) (2014) 99 – 122