Branching on multi-aggregated variables

Gerald Gamrath[1], Anna Melchiori[2], Timo Berthold[3], Ambros M. Gleixner[1], and Domenico Salvagnin[4]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, {gamrath,gleixner}@zib.de
[2] University of Padua, Via Trieste 63, 35121 Padua, Italy, melch.anna@gmail.com
[3] Fair Isaac Europe Ltd, c/o ZIB, Takustr. 7, 14195 Berlin, Germany, timoberthold@fico.com
[4] DEI, University of Padua, Via Gradenigo 6/b, 35121 Padua, Italy, salvagni@dei.unipd.it

Abstract. In mixed-integer programming, the branching rule is a key component to a fast convergence of the branch-and-bound algorithm. The most common strategy is to branch on simple disjunctions that split the domain of a single integer variable into two disjoint intervals. Multi-aggregation is a presolving step that replaces variables by an affine linear sum of other variables, thereby reducing the problem size. While this simplification typically improves the performance of MIP solvers, it also restricts the degree of freedom in variable-based branching rules.
We present a novel branching scheme that tries to overcome the above drawback by considering general disjunctions defined by multi-aggregated variables in addition to the standard disjunctions based on single variables. This natural idea results in a hybrid between variable- and constraint-based branching rules. Our implementation within the constraint integer programming framework SCIP incorporates this into a full strong branching rule and reduces the number of branch-and-bound nodes on a general test set of publicly available benchmark instances. For a specific class of problems, we show that the solving time decreases significantly.

1 Introduction

Since the invention of the branch-and-bound method for solving mixed-integer linear programming in the 1960s [1,2], branching rules have been an important field of research, being one of its core components. For surveys, see [3,4,5]. In this paper we address branching strategies for mixed-integer linear programs (MIPs) of the form

$$\min\{c^T x : Ax \leq b, \ell \leq x \leq u, x_i \in \mathbb{Z} \text{ for all } i \in \mathcal{I}\} \tag{1}$$

with $c \in \mathbb{R}^n, A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $\ell, u \in \bar{\mathbb{R}}^n$ where $\bar{\mathbb{R}} := \mathbb{R} \cup \{\pm\infty\}$, and $\mathcal{I} \subseteq \mathcal{N} = \{1, \dots, n\}$ being the index set of integer variables. When removing the integrality restrictions, we obtain the *linear programming (LP) relaxation* of the problem.

If the solution $\tilde{x}$ to the LP relaxation of (1) is fractional, i.e., if the index set $\tilde{\mathcal{I}} := \{i \in \mathcal{I} : \tilde{x}_i \notin \mathbb{Z}\}$ of *fractional variables* is not empty, the task of a branching rule is to split the problem into two or more subproblems. The strategy is typically to exclude the LP solution from all subproblems while keeping the feasible integer solutions, each being present in exactly one subproblem.

The choice of which subproblems to create is crucial for the performance of the algorithm. The approach most widely used by MIP solvers is to branch on *simple disjunctions*

$$x_k \leq \lfloor \tilde{x}_k \rfloor \quad \bigvee \quad x_k \geq \lceil \tilde{x}_k \rceil. \tag{2}$$

each side being enforced in one subproblem. As this procedure splits the domain of a single variable at a time, it is also called *branching on variables*. Alternatively, branching can be performed on a *general disjunction*

$$\pi^T x \leq \pi_0 \quad \bigvee \quad \pi^T x \geq \pi_0 + 1. \tag{3}$$

where $(\pi, \pi_0) \in \mathbb{Z}^n \times \mathbb{Z}$, and $\pi_i = 0$ for all $i \notin \mathcal{I}$.

Branching on variables can be seen as the special case in which all considered disjunctions are of the form $(\pi, \pi_0) = (e_j, \lfloor \tilde{x}_j \rfloor)$, e_j being the j-th unit vector. Note that for branching on variables the set of *branching candidates* among which a branching rule chooses is usually the list of fractional variables $\tilde{\mathcal{I}}$. For branching on general disjunctions, the branching candidates consist of a potentially much larger list of disjunctions of form (3). Research on general branching disjunctions has largely been dedicated to determine a short list of promising candidates, see our literature overview in Sec. 2.

Another key component of state-of-the-art MIP solvers is *presolving*. It is applied before the branch-and-bound process and transforms a given MIP instance into a typically smaller instance with a tighter relaxation, which is hopefully easier to solve. These reductions can be based on pure feasibility arguments (keeping the set of feasible solutions unchanged) as well as optimality arguments (excluding also feasible solutions as long as one optimal solution remains).

Important presolving operations are fixings, aggregations, and multi-aggregations of variables. Here, *fixing* means that a variable gets permanently assigned to a constant value, *aggregation* means that a variable is replaced by (a constant value plus a scalar multiple of) another variable, and *multi-aggregation* means that a variable gets replaced by an affine linear combination of several variables. Hence, a *multi-aggregated* variable is a variable that is present in the original formulation, but is represented by an affine linear sum of variables in the presolved problem.

Contribution. The intuitive appeal of branching on general disjunctions is the increased degree of freedom that promises the creation of more balanced subproblems with tighter relaxations. This obvious advantage comes with the main challenge of determining promising candidate disjunctions. We address this difficulty by considering specifically the subset of disjunctions that are defined by the affine combinations stemming from multi-aggregations performed during the presolving stage. These disjunctions are naturally available in state-of-the-art MIP solvers at no cost and branching on them mimics branching on decision variables in the original model formulation.

Note that while the set of all general disjunctions of form (3) is exponentially large even when restricting π to $\{-1, 0, 1\}^n$, the set of multi-aggregated variables provides a list of potential candidates that is linear w.r.t. the size of the original model. Our experiments show that—in combination with standard single-variable disjunctions—this restriction yields not only a managable, but also computationally promising set of candidate disjunctions.

The remainder of the article is organized as follows. In Sec. 2, we give an overview of the literature on branching in MIP, with a particular focus on branching on general disjunctions. Sec. 3 introduces in more detail the concept of multi-aggregation, and Sec. 4 describes the idea of our new branching strategy and details about the implementation in the constraint integer programming framework SCIP [6,7]. In Sec. 5 we presents our computational study and Sec. 6 contains our conclusions and gives an outlook on potential extensions of branching on multi-aggregated variables.

2 Related work

Various criteria for selecting fractional variables for branching on simple disjunctions have been presented in the literature. Most selection rules focus on the improvement in the dual bound that the branching restrictions produce in the created child nodes since this helps to tighten the global dual bound and prune nodes early. A fundamental strategy of this type is *strong branching* [8], which tentatively restricts the bound of a candidate variable and explicitly computes the resulting dual bound of the potential child node by solving the LP relaxation.

The *full strong branching* rule applies this at every node for each fractional variable. This typically leads to very small branch-and-bound trees, but on the other hand invests considerable effort in analyzing candidates. On average, this usually results in an overall performance deterioration w.r.t. computing time [5]. Nevertheless, the default branching rules in most state-of-the-art MIP solvers use some restricted form of strong branching and combine it with history information to reduce the computational effort for branching in later solving stages. Further strategies based on the same criteria can be found in [9,4,7,10,11,12]. Recent research efforts on different criteria for variable-based branching rules include, e.g., [13,14,15,16,17,18].

Branching on general disjunctions dates back to the 1980s [19], and has been addressed by various researchers in the last 15 years, see, e.g., [20,21,22,23,24]. The main challenge is to find a good class of general disjunctions that can lead to a better and more accurate tightening process of the feasible region, and consequently to a faster convergence of the dual bound to the optimal solution value, ideally without requiring a high computational effort for its generation and evaluation.

Owen and Mehrotra [20] present an algorithm that determines the branching disjunction via a neighborhood search heuristic. They prove that their algorithm is finite, if all variables have finite bounds and the size of the coefficients in the used disjunctions is bounded. As a consequence, they restrict their search to coefficients $\pi_i \in \{-1, 0, 1\}$. Combining this idea with [13], Mahmoud and Chinneck [24] choose a constraint that is active for the current LP optimum and construct a general disjunction with coefficients in $\{-1, 0, 1\}$ that is as perpendicular or as parallel as possible to the chosen active constraint.

Karamanov and Cornuéjols [22] consider disjunctions which correspond to Gomory mixed integer cuts (GMICs) [25]. They filter the GMICs to only keep the ten deepest cuts, and apply a strong-branching-like procedure on the corresponding candidate disjunctions. An extension of [22] is proposed by Cornuéjols et al. [23] who not only consider GMICs on tableau rows, but also on linear combinations of the tableau rows.

On the theoretical side, Mahajan and Ralphs proved that the problem of finding a general disjunction with maximal objective gain is $\mathcal{NP}$-hard [26]. Finally, Local Branching by Fischetti and Lodi [27] is a strategy to interleave variable-based branching with branching on general $\{-1, 0, 1\}$-disjunctions. These disjunctions measure the distance to the incumbent solution.

A typical result when branching on general disjunctions in MIP is that the generated branching trees are smaller on average, but the performance deteriorates w.r.t. running time. One major reason for this computational overhead is that the set of candidate disjunction for branching is much larger, so that a lot of time is spent determining the best one to choose at each node. However, this could in principle be overcome if we had more efficient (implicit) algorithms for evaluating the set of candidates, and it is of course not an issue when such set is still relatively small.

Another, more structural reason is that branching on variables changes a variable bound, which often fixes the variable to the other bound (in particular when branching on binary variables). This *decreases* the size of the LP relaxation for the subproblems by (at least) one column, whereas branching on general disjunctions potentially *increases* the LP's size by one row. This affects the simplex algorithm, which in most cases is the method of choice for solving the LP relaxations during LP-based branch-and-bound. Because the dimension of the basis matrix increases when adding a new row, most simplex implementations will have to recompute its factorization, causing computational overhead. In addition, many performance-relevant components of state-of-the-art MIP solvers such as domain propagation and conflict analysis are currently designed to benefit from branching on variables and become less effective when branching is performed on general disjunctions.

3 Multi-aggregations of variables

Before the branch-and-bound process is started, state-of-the-art MIP solvers perform a presolving phase during which they analyze the problem and remove redundancies, tighten the formulation, and collect information about the problem structure, see [28,29,30,7,31] for examples. This procedure is exact in the sense that each optimum of the simplified problem can be mapped to an optimal solution of the original problem.

The presolving technique which forms the basis of our newly developed branching rule is the multi-aggregation of variables. It reduces the number of variables by

1. detecting that—in at least one optimal solution—variable x_k equals an affine linear combination of other variables, i.e.,
$$x_k = \sum_{j \in \mathcal{S}_k} \alpha_j^k x_j + \beta^k, \tag{4}$$
with $\mathcal{S}_k \subseteq \mathcal{N}$, $k \notin \mathcal{S}_k$,
2. replacing every occurrence of x_k in constraints and objective function by the right-hand side in (4), and
3. enforcing the bounds on x_k—if finite—by adding the new constraint
$$\ell_k \leq \sum_{j \in \mathcal{S}_k} \alpha_j^k x_j + \beta^k \leq u_k. \tag{5}$$

Equation (4) may either be explicitly present as one of the problem constraints[5] or implied by a combination of constraints and optimality conditions. An example for the latter is the case when x_k appears in exactly one constraint and its objective function coefficient ensures that this constraint will be fulfilled with equality in an optimal solution. The constraint integer programming framework SCIP, which we use for our computational experiments, has five different presolving operations in which multi-aggregation is performed.

After this step, one of the constraints implying (4) usually becomes void or is modified to enforce (5). If x_k is an integer variable, multi-aggregations are only performed if the integrality is enforced by the multi-aggregation. This holds, e.g., if (4) is an integer combination of integer variables, i.e., $\mathcal{S}_k \subseteq \mathcal{I}$, $\alpha_j^k \in \mathbb{Z}$ for all $j \in \mathcal{S}_k$, and $\beta^k \in \mathbb{Z}$.

In order to avoid a deterioration of performance and potential numerical problems during LP solving, it is crucial to safe-guard against fill-in in the constraint matrix. This can be done a priori by comparing the number of non-zeros that would be removed to the number of non-zeros that would get introduced in the constraint matrix, the latter of which can be bounded from above by the cardinality of $\mathcal{S}$ times the number of occurences of x_k.

In the following, we call a variable *inactive*, if presolving removed it from the problem. This includes variables which are already fixed to some value as well as aggregated and multi-aggregated variables. All other variables are called *active*. During the subsequent solving process, inactive variables are disregarded since

[5] Although in (1) we have formulated MIPs in terms of inequalities, this also includes equality constraints formulated via two inequalities.

their solution value is uniquely defined by the value of the active variables. In the remainder of this article, a MIP of form (1) always refers to the presolved problem containing only active variables. When referencing the original problem, we are using the following notation: the index sets of original and corresponding integer variables are denoted by $\mathcal{N}'$ and $\mathcal{I}'$, respectively. Original variables are written as x'_i and the variable on the left-hand side of a multi-aggregation (4) is an original variable x'_k, while all variables on the right-hand side are active variables x_j.[6]

4 Branching on multi-aggregated variables

Simple aggregations of form $x'_k = \alpha^k_j x_j + \beta^k$ performed during presolving do not restrict the choices of variable-based branching rules since branching on the subsequently inactive variable x'_k remains implicitly possible by branching on x_j. In contrast, branching on multi-aggregated variables cannot be realized via branching on active variables. We are not aware of any study that has investigated the effect of multi-aggregation on the performance of branching rules and note that this restriction may indeed have negative performance impact—especially since this effect is currently not considered during presolving.

Our new branching strategy considers the general disjunctions defined by all multi-aggregations (4) for which $k \in \mathcal{I}'$ but $\sum_{j \in \mathcal{S}_k} \alpha^k_j x_j + \beta^k$ evaluates to a fractional value in the current LP solution. In a strong branching fashion, we tentatively test which improvement in the local dual bounds we would obtain by adding one part of the corresponding general disjunction. We compare this to the improvements obtained by simple disjunctions on fractional active variables and choose the best among all branching disjunctions.

The motivation is twofold: first, to compensate for the above drawback, and second, to obtain a set of candidates for general branching disjunctions that is available at no cost in state-of-the-art MIP solvers and computationally managable. As mentioned earlier, the set of all general disjunctions of form (3) is exponentially large even when restricting π to $\{-1, 0, 1\}^n$, in contrast to that, the number of multi-aggregations is linear w.r.t. the size of the original model.

In an LP-based branch-and-bound algorithm, the multi-aggregated branching rule is called whenever the optimal solution $\tilde{x}$ to the linear relaxation of the current node is fractional. Its procedure is outlined in Algorithm 1.

First, strong branching is performed on all elements in the set of fractional variables $\tilde{\mathcal{I}}$. For each candidate variable x_i, two auxiliary LPs are solved to compute dual bounds $\tilde{z}^-$ and $\tilde{z}^+$ for the potential child nodes. If both are larger than or equal the given upper bound (usually the objective function value of the incumbent solution), we can stop since no better solution can be found in the current subproblem and the node can be cut off. If only one of the two dual bounds is smaller than the upper bound, the corresponding bound change

[6] Note that nested multi-aggregations can be transferred into this form by (recursively) replacing inactive variables in the right-hand side of a multi-aggregation (4) by the corresponding constant or affine linear combination of variables.

Algorithm 1: Multi-aggregated branching rule

```
input  :  – a MIP of form (1),
          – an optimal solution $\tilde{x}$ of the LP relaxation,
          – an upper bound $z^*$ on the objective value of solutions, and
          – the index set $\mathcal{A}' \subseteq \mathcal{N}'$ of multi-aggregations of form (4),
            $x'_k = \sum_{j \in \mathcal{S}^k} \alpha^k_j x_j + \beta^k$, $k \in \mathcal{A}'$, $\mathcal{S}^k \subseteq \mathcal{N}$
output :  – a branching disjunction of form (3) given as $(\tilde{\pi}, \tilde{\pi}_0) \in \mathbb{Z}^n \times \mathbb{Z}$, or
          – a valid inequality, or
          – the conclusion that the current node can be pruned
1  begin
       // 0. initialization
2      for $k \in \mathcal{A}' \cap \mathcal{I}'$ do      // compute LP values of multi-aggregated vars
3          $\tilde{x}'_k := \sum_{j \in \mathcal{S}^k} \alpha^k_j x_j + \beta^k$
4      $\tilde{\mathcal{I}} := \{i \in \mathcal{I} : \tilde{x}_i \notin \mathbb{Z}\}$                    // single-variable candidates
5      $\tilde{\mathcal{A}} := \{k \in \mathcal{A}' \cap \mathcal{I}' : \tilde{x}'_k \notin \mathbb{Z}\}$              // multi-aggregated candidates
6      $(\tilde{\pi}, \tilde{\pi}_0) := (0, 0)$                               // incumbent disjunction
7      $s_{(\tilde{\pi}, \tilde{\pi}_0)} := -\infty$                                    // incumbent score
       // 1. full strong branching on simple disjunctions
8      for $i \in \tilde{\mathcal{I}}$ do
9          $\tilde{z}^- \leftarrow \min\{c^T x : Ax \leq b, \ell \leq x \leq u, x_i \leq \lfloor \tilde{x}_i \rfloor\}$
10         $\tilde{z}^+ \leftarrow \min\{c^T x : Ax \leq b, \ell \leq x \leq u, x_i \geq \lfloor \tilde{x}_i \rfloor + 1\}$
11         if $\min\{\tilde{z}^-, \tilde{z}^+\} \geq z^*$ then return *current node can be pruned*
12         else if $\tilde{z}^- \geq z^*$ then return *valid inequality* $x_i \geq \lfloor \tilde{x}_i \rfloor + 1$
13         else if $\tilde{z}^+ \geq z^*$ then return *valid inequality* $x_i \leq \lfloor \tilde{x}_i \rfloor$
14         else if $\mathrm{score}(\tilde{z}^-, \tilde{z}^+) > s_{(\tilde{\pi}, \tilde{\pi}_0)}$ then
15             $(\tilde{\pi}, \tilde{\pi}_0) := (e_i, \lfloor \tilde{x}_i \rfloor)$
16             $s_{(\tilde{\pi}, \tilde{\pi}_0)} := \mathrm{score}(\tilde{z}^-, \tilde{z}^+)$
       // 2. full strong branching on multi-aggregated disjunctions
17     for $k \in \tilde{\mathcal{A}}$ do
18         $\tilde{z}^- \leftarrow \min\{c^T x : Ax \leq b, \ell \leq x \leq u, \sum_{j \in \mathcal{S}^k} \alpha^k_j x_j \leq \lfloor \tilde{x}'_k \rfloor - \beta^k\}$
19         $\tilde{z}^+ \leftarrow \min\{c^T x : Ax \leq b, \ell \leq x \leq u, \sum_{j \in \mathcal{S}^k} \alpha^k_j x_j \geq \lfloor \tilde{x}'_k \rfloor - \beta^k + 1\}$
20         if $\min\{\tilde{z}^-, \tilde{z}^+\} \geq z^*$ then return *current node can be pruned*
21         else if $\tilde{z}^- \geq z^*$ then return $\sum_{j \in \mathcal{S}^k} \alpha^k_j x_j \geq \lfloor \tilde{x}'_k \rfloor - \beta^k + 1$ *valid*
22         else if $\tilde{z}^+ \geq z^*$ then return $\sum_{j \in \mathcal{S}^k} \alpha^k_j x_j \leq \lfloor \tilde{x}'_k \rfloor - \beta^k$ *valid*
23         else if $\mathrm{score}(\tilde{z}^-, \tilde{z}^+) > s_{(\tilde{\pi}, \tilde{\pi}_0)}$ then
24             $(\tilde{\pi}, \tilde{\pi}_0) := (\sum_{j \in \mathcal{S}^k} \alpha^k_j e_j, \lfloor \tilde{x}'_k \rfloor - \beta^k)$
25             $s_{(\tilde{\pi}, \tilde{\pi}_0)} := \mathrm{score}(\tilde{z}^-, \tilde{z}^+)$
26     return *branching disjunction* $(\tilde{\pi}, \tilde{\pi}_0)$
27 end
```

can directly be applied at the current problem, since the other child node does not contain an improving solution. If both dual bounds are smaller than the upper bound, the score for the candidate variable is computed and the simple disjunction $(e_i, \lfloor \tilde{x}_i \rfloor)$ corresponding to branching on this variable is stored as new best candidate if its score exceeds the best one found so far. The branching score used in SCIP is the product of the objective gains of the two child nodes, more specifically,

$$\text{score}(\tilde{z}^-, \tilde{z}^+) = \max\{\Delta_j^-, \epsilon\} \cdot \max\{\Delta_j^+, \epsilon\} \tag{6}$$

with $\epsilon = 10^{-6}$ and $\Delta_j^- = \tilde{z}^- - c^T\tilde{x}$ and $\Delta_j^+ = \tilde{z}^+ - c^T\tilde{x}$ being the objective gains in the child nodes when branching on x_j.

In the second step of the algorithm, full strong branching is performed on the general disjunctions defined by the multi-aggregated variables of the original problem. To this end, all integer multi-aggregated variables x'_k are taken into account for which the LP solution translates into a fractional solution $\tilde{x}'_k$. Analogously to the first step, two auxiliary LPs are solved with the potential branching disjunction added and the computed dual bounds are compared to the upper bound in order to prune the node or identify valid constraints. The score of the candidate disjunction is evaluated and compared to the best score found so far. If it is higher, the candidate disjunction is updated. Note that possible ties are broken in favor of candidate variables, since those are evaluated first and we are looking for strict improvements.

In the case that a valid bound change or inequality was found, we stop the branching rule, tighten the formulation, and return to the MIP solving process, which will continue by applying domain propagation, reoptimizing the LP, and calling the branching rule again if needed. After the evaluation of all candidate variables and disjunctions, and if no such valid bound or inequality was found, the best disjunction is returned and branching is performed on it.

5 Computational results

In the following, we present our experiments with branching on multi-aggregated variables. We used the academic constraint integer programming framework SCIP 3.1.0 [6,7] with SoPlex 1.7.0.4 [32] as underlying LP solver and implemented Algorithm 1 as a branching rule plug-in. Our new method builds on the full strong branching scheme and extends it by choosing as the set of candidates to evaluate via strong branching not only candidate variables, but also candidate disjunctions given by multi-aggregations. Therefore, it is consequential to compare our strategy with the basic full strong branching rule of SCIP.

All results were obtained on a cluster of 3.2 GHz Intel Xeon X5672 CPUs with 48 GB main memory, running each job exclusively on one node. To keep the computation time under control, a time limit of 7200 seconds for each instance was imposed.

Settings. We compare the methods for two different settings. The first one, called *pure*, focuses on the main goal of a branching rule, namely proving the optimality of a solution. To this end, it disables cutting plane separation, primal heuristics, domain propagation, restarts, and conflict analysis. Additionally, we provide the optimal objective value as a cutoff bound at the beginning of the solving process. This is done in order to measure only the impact of branching without side-effects to and from other solver components. In particular, this reduces performance variability, cf. [33]. The second setting is called *default* and runs full strong branching (SB) and multi-aggregated branching (MA) in the SCIP default environment.

Instances. Our first experiments were performed on a test set of scheduling [34,35] instances. More specifically, we were investigating *resource allocation and scheduling* problems, where jobs are assigned to machines, thereby minimizing the processing costs which depend on the machine on which a job is performed. Given sets $\mathcal{J}$ of jobs and $\mathcal{M}$ of machines, the capacity $C \in \mathbb{N}$ of the machines, and assignment cost $c_{j,m}$, resource allocation and scheduling can be expressed via the following MIP model [36]:

$$\begin{aligned}
\min \sum_{m\in\mathcal{M}} \sum_{j\in\mathcal{J}} & c_{j,m} x_{j,m} \\
\text{s.t.} \quad \sum_{m\in\mathcal{M}} x_{j,m} &= 1 && \text{for all } j \in \mathcal{J}, \\
\sum_{t\in\mathcal{T}_{j,m}} x^t_{j,m} &= x_{j,m} && \text{for all } m \in \mathcal{M},\ j \in \mathcal{J}, \\
\sum_{j\in\mathcal{J}} \sum_{\bar{t}\in\mathcal{T}^t_{j,m}} c_j x^{\bar{t}}_{j,m} &\le C && \text{for all } m \in \mathcal{M},\ t \in \mathcal{T}, \\
x^t_{j,m} &\in \{0,1\} && \text{for all } m \in \mathcal{M},\ j \in \mathcal{J},\ t \in \mathcal{T}_{j,m}, \\
x_{j,m} &\in \{0,1\} && \text{for all } m \in \mathcal{M},\ j \in \mathcal{J}.
\end{aligned}$$

The formulation uses binary variables $x_{j,m}$ and $x^t_{j,m}$, which represent the decision whether job $j \in \mathcal{J}$ is processed on machine $m \in \mathcal{M}$, and whether the processing of job $j \in \mathcal{J}$ on machine $m \in \mathcal{M}$ is started at time $t \in \mathcal{T}$, respectively. We use two subsets of the time periods: $\mathcal{T}_{j,m}$ which contains all time steps in which job j can start on machine m, and $\mathcal{T}^t_{j,m}$ which further restricts $\mathcal{T}_{j,m}$ to those starting times causing j to be (still) running in period t. When solving these instances, the $x_{j,m}$ variables are frequently multi-aggregated, which makes this problem an interesting test case for our first experiments.

We used a collection of 335 scheduling instances modeled this way in [36]. We excluded all instances that were solved either during presolving or at the root node. This left a total of 263 problem instances with the default setting and 276 instances with the pure setting.

In our second experiment, we used a test set of general MIP instances from different sources, including MIPLIB [37,38,33] and the Cor@l test set [39]. We removed some instances which to the best of our knowledge have never been solved so far and two numerically unstable instances giving slightly different results with both branching rules. Additionally, we restricted the test set to instances in which presolving performed multi-aggregations and removed instances which were solved during presolving or at the root node without branching. This gave us two test sets for the pure and default settings of 76 and 107 instances, respectively.

In the following, we present aggregated results over these test sets. Detailed computational results for each instance can be found in the appendix of the preprint version of this article.[7]

5.1 Results for scheduling instances

Table 1 compares the multi-aggregated branching strategy (MA) against the basic version of full strong branching (SB) available in SCIP with both pure and default settings, as indicated in the first column.

The remainder of the table is split into two parts: The four columns below the "scheduling test set" label display numbers about the performance on the complete scheduling test set. Column "size" shows the number of instances in the test set, "solved" gives the number of instances solved to proven optimality within the time limit of two hours. Column "faster" ("slower") show the number of instances that the MA strategy solved at least 10 % faster (slower) than standard full strong branching.

The right side of the table, labeled "all optimal", shows results for the subset of instances that both variants in the respective setting solved to optimality. Column "size" shows the number of instances in this subset, "nodes" the shifted geometric mean of the B&B nodes and "time (s)" the shifted geometric mean of the running time in seconds. We use shifts of 100 and 10 for the number of nodes and the solving time, respectively. For a discussion of the shifted geometric mean, we refer to [40, Appendix A3].

Let us first look at the results with the pure settings, which focus on the plain branch-and-bound performance. They are promising: 25 more instances (142 vs. 117) can be solved by branching on multi-aggregated variables compared to standard strong branching; this corresponds to an increase of more than 20 %. Furthermore, 100 instances are solved at least 10 % faster with the new method, compared to 13 which slow down by 10 % or more. This corresponds to 70 % of the instances being solved faster with branching on multi-aggregated variables. Looking at the instances that were solved to optimality by both variants, both the number of nodes and the requested time are reduced by a factor of two on average: 58 % less nodes are needed and 49 % less time.

When looking at the results with default settings, the effect is smaller, but still significant: the multi-aggregated branching strategy is able to solve 9 more instances to optimality, with 56 instances being solved faster and 31 slower. On

[7] Available electronically under `http://www.zib.de/gamrath/GamrathEtAl2014.pdf`.

Table 1: Results for scheduling instances with default and pure settings

	scheduling test set				**all optimal**		
setting	size	solved	faster	slower	size	nodes	time (s)
SB-pure	276	117			115	472	51.8
MA-pure	276	142	100	13	115	196	26.4
SB-default	263	126			122	349	84.6
MA-default	263	135	56	31	122	221	70.3

instances that both variants solve to optimality, it needs 37 % less nodes and reduces the solving time by 17 %.

Let us note that the positive effect of branching on multi-aggregated variables grows stronger the harder an instance is. This seems reasonable since the additional overhead might not pay off if a standard strong branching is able to solve an instance within a few nodes. When taking into account only instances which needed more than 100 seconds to solve by at least one setting, the reduction in the number of nodes and the solving time goes up to 42 % and 25 %, respectively.

This first computational experiment shows that branching on multi-aggregated variables can significantly improve the performance of SCIP compared to a pure variable-based branching rule: more instances are solved, with less enumeration, in shorter time. Note that in all cases the relative reduction in running time was smaller than the relative reduction in the number of branch-and-bound nodes, which is a typical result for branching strategies that involve general disjunctions (see Sec. 2).

In order to analyze the impact of the new branching rule in more detail, we collected some statistics during the execution of SCIP. On average over the test set, the number of integer multi-aggregations is only 5.7 % of the number of integer variables. Thus, the list of branching candidates is only slightly extended in most cases, which overcomes a typical issue for branching on general disjunctions. Interestingly, despite this relatively small number of multi-aggregations, 39 % of the branching decisions select a multi-aggregated disjunction for branching. Even more, in 85 % of the cases, the first branching on a multi-aggregated disjunction was performed at the root node.

Finally, each time we perform a multi-aggregated branching, we store the ratio of the gain that we would have obtained when branching on the best fractional variable compared to the gain obtained by branching on the current multi-aggregated variable. The gain is computed as the square root of the SCIP branching score value and thus measures the improvement in the score SCIP tries to maximize. On average over all calls where we branched on a multi-aggregated disjunction, the gain would have been reduced to 22 % by branching on the best variable instead.

Table 2: Results for general MIP instances with default and pure settings

	MIP test set				**all optimal**		
setting	size	solved	faster	slower	size	nodes	time (s)
SB-pure	76	33			32	983	150.9
MA-pure	76	32	0	26	32	852	188.9
SB-default	107	55			49	253	100.4
MA-default	107	57	1	33	49	269	126.3

5.2 Results for general MIP instances

The results for our collection of general MIP instances are presented in Table 2. The columns and rows show the same statistics as described in Sec. 5.1. We can see that on these instances, multi-aggregated branching is significantly slower and solved one less instance in both settings, compared to standard strong branching. With pure settings, the solving time increases by 25 % while the number of branch-and-bound nodes is decreased by 13 %. Compared to the scheduling instances, multi-aggregated variables are much less effective for branching. That the increased effort in strong branching outweighs the observed node reduction seems plausible. These results confirm our observation from the scheduling instances in the sense that the impact on the number of branch-an-bound nodes was better than the impact on the overall running time. For the scheduling instances, the additional candidates were structurally different and allowed different, higher-level decisions which had an enormous effect on the tree size that even allowed for a running time reduction. For standard MIPs, however, such a large effect is apparently obtained rarely, thus, the performance deteriorates on average. The picture looks even worse for the default settings. Here, the solving time increases by 26 % and the number of nodes now increases by 6 % as well.

This increase in the number of nodes is surprising as the branching disjunctions we are using have a better score and should therefore lead to a faster convergence. It can be explained, however, by the tailoring of many MIP solving algorithms towards variable-based branching. Domain propagation (or *node preprocessing*, see, e.g., [28] for MIP), for example, tries to tighten the local domains of variables by inspecting the constraints and current domains of other variables at the local subproblem. Tightening or fixing variables by branching is naturally beneficial for domain propagation, the impact of adding general disjunctions is rather opaque. Furthermore, techniques like primal heuristics, cutting plane separation, or conflict analysis profit from tightened variable bounds rather than from added general disjunctions. Since all these techniques help to reduce the size of the branch-and-bound tree, branching on general disjunctions with a high branching score can even increase the number of nodes, since as a side effect it makes the named procedure less effective.

We see our results for general MIPs as an important negative result that confirms previous observations by other authors that it is hard to find a branching rule on general disjunctions which is competitive on standard MIP benchmarks. Our results indicate that this holds even when restricting the selection to relatively few additional candidates that are naturally obtained from the problem structure. Finally, adapting procedures like primal heuristics or conflict analysis in such a way that they benefit from added constraints as much as from tightened or fixed variables might be a prerequisite to excel with constraint-based branching schemes in state-of-the-art MIP solvers.

6 Conclusions and outlook

In this paper, we presented a new branching rule which takes into account a specific type of general disjunctions. These general disjunctions, so-called multi-aggregations, are the affine linear sums of active variables in the presolved problem, which correspond to a decision variable in the original problem. We extended the full strong branching rule of SCIP by taking additionally into account all general disjunctions induced by multi-aggregations. On a set of scheduling instances, this significantly improved the performance of SCIP w.r.t. the tree size as well as the solving time and the number of solved instances.

We tested the same branching rule on standard MIP benchmark sets. The results were much less convincing, but a certain potential for branching on multi-aggregated variables was indicated by the observation that in a "pure" setting, it led to a reduction in the number of branch-and-bound nodes for general MIPs. However, before this potential can be harnessed, we conclude that many advanced solution techniques applied in state-of-the-art MIP solvers—domain propagation, conflict analysis, etc.—must be extended towards a more efficient handling of general disjunctions. An additional performance bias is the slow-down in current simplex implementations when adding and removing constraints. This bottleneck may be alleviated by the recent developments of [41,42], which improve the underlying linear algebra routines such that the factorization of the basis matrix is preserved when adding new rows. We identify these points as important directions for future research.

The proposed strategy has been studied and implemented for the first time in the constraint integer programming framework SCIP. Since it proved its effectiveness for certain problem classes, it will be available in the next release of SCIP.

References

1. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**(3) (1960) 497–520
2. Dakin, R.J.: A tree-search algorithm for mixed integer programming problems. The Computer Journal **8**(3) (1965) 250–255

3. Mitra, G.: Investigation of some branch and bound strategies for the solution of mixed integer linear programs. Mathematical Programming **4** (1973) 155–170
4. Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies in mixed-integer programming. INFORMS Journal on Computing **11**(2) (1999) 173–187
5. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters **33** (2005) 42–54
6. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In Perron, L., Trick, M.A., eds.: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008. Volume 5015 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (May 2008) 6–20
7. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1) (2009) 1–41
8. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: On the solution of traveling salesman problems. Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung **Extra Volume ICM III** (1998) 645–656
9. Gauthier, J.M., Ribière, G.: Experiments in mixed-integer linear programming using pseudo-costs. Mathematical Programming **12**(1) (1977) 26–47
10. Fischetti, M., Monaci, M.: Branching on nonchimerical fractionalities. OR Letters **40**(3) (2012) 159–164
11. 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. Volume 7874 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 28–43
12. Gamrath, G.: Improving strong branching by domain propagation. EURO Journal on Computational Optimization **2**(3) (2014) 99 – 122
13. Patel, J., Chinneck, J.: Active-constraint variable ordering for faster feasibility of mixed integer linear programs. Mathematical Programming **110** (2007) 445–474
14. Kılınç Karzan, F., Nemhauser, G.L., Savelsbergh, M.W.: Information-based branching schemes for binary linear mixed integer problems. Mathematical Programming Computation **1** (2009) 249–293
15. 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 (May 2009) 309–311
16. Fischetti, M., Monaci, M.: Backdoor Branching. In Günlück, O., Woeginger, G.J., eds.: Integer Programming and Combinatorial Optimization. Volume 6655 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2011) 183–191
17. Gilpin, A., Sandholm, T.: Information-theoretic approaches to branching in search. Discrete Optimization **8**(2) (2011) 147–159
18. Pryor, J., Chinneck, J.W.: Faster integer-feasibility in mixed-integer linear programs by branching to force change. Computers & OR **38**(8) (2011) 1143–1152
19. Ryan, D.M., Foster, B.A.: An integer programming approach to scheduling. In Wren, A., ed.: Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling. North Holland, Amsterdam (1981) 269–280
20. Owen, J.H., Mehrotra, S.: Experimental results on using general disjunctions in branch-and-bound for general-integer linear programs. Computational Optimization and Applications **20** (2001) 159–170

21. Mahajan, A., Ralphs, T.K.: Experiments with branching using general disjunctions. In Chinneck, J.W., Kristjansson, B., Saltzman, M.J., eds.: Operations Research and Cyber-Infrastructure. Volume 47 of Operations Research/Computer Science Interfaces Series. Springer US (2009) 101–118
22. Karamanov, M., Cornuéjols, G.: Branching on general disjunctions. Mathematical Programming **128** (2011) 403–436
23. Cornuéjols, G., Liberti, L., Nannicini, G.: Improved strategies for branching on general disjunctions. Mathematical Programming **130** (2011) 225–247
24. Mahmoud, H., Chinneck, J.W.: Achieving milp feasibility quickly using general disjunctions. Computers & OR **40**(8) (2013) 2094–2102
25. Gomory, R.E.: An algorithm for the mixed integer problem. Technical report, RAND Corporation (1960)
26. Mahajan, A., Ralphs, T.K.: On the complexity of selecting disjunctions in integer programming. SIAM Journal on Optimization **20**(5) (2010) 2181–2198
27. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1-3) (2003) 23–47
28. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing **6** (1994) 445–454
29. Brearley, A.L., Mitra, G., Williams, H.P.: Analysis of mathematical programming problems prior to applying the simplex algorithm. Mathematical Programming **8**(1) (1975) 54–83
30. Bixby, R.E., Wagner, D.K.: A note on detecting simple redundancies in linear systems. Operation Research Letters **6**(1) (1987) 15–17
31. Gamrath, G., Koch, T., Martin, A., Miltenberger, M., Weninger, D.: Progress in presolving for mixed integer programming. Technical Report 13-48, ZIB, Takustr.7, 14195 Berlin (2013)
32. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. PhD thesis, Technische Universität Berlin (1996)
33. 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. Mathematical Programming Computation **3**(2) (2011) 103–163
34. Baker, K.: Introduction to sequencing and scheduling. Wiley (1974)
35. Baker, K.R., Trietsch, D.: Principles of Sequencing and Scheduling. Wiley (2009)
36. Heinz, S., Ku, W.Y., Beck, J.C.: Recent improvements using constraint integer programming for resource allocation and scheduling. In Gomes, C., Sellmann, M., eds.: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. Volume 7874 of LNCS. Springer (2013) 12–27
37. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima (58) (June 1998) 12–15
38. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters **34**(4) (2006) 1–12
39. COR@L: MIP Instances (2014) `http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/`.
40. Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universität Berlin (2007)
41. Gleixner, A.M.: Factorization and update of a reduced basis matrix for the revised simplex method. ZIB-Report 12-36, Zuse Institute Berlin (October 2012)
42. Wunderling, R.: The kernel simplex method. Talk at the 21st International Symposium on Mathematical Programming, Berlin, Germany (August 2012)