Local Rapid Learning for Integer Programs

Timo Berthold[1], Peter J. Stuckey[2], and Jakob Witzig[3]

[1] Fair Isaac Germany GmbH, Takustr. 7, 14195 Berlin, Germany
timoberthold@fico.com
[2] Monash University and Data61, Melbourne, Australia
Peter.Stuckey@monash.edu
[3] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
witzig@zib.de

Abstract. Conflict learning algorithms are an important component of modern MIP and CP solvers. But strong conflict information is typically gained by depth-first search. While this is the natural mode for CP solving, it is not for MIP solving. Rapid Learning is a hybrid CP/MIP approach where CP search is applied at the root to learn information to support the remaining MIP solve. This has been demonstrated to be beneficial for binary programs. In this paper, we extend the idea of Rapid Learning to integer programs, where not all variables are restricted to the domain $\{0, 1\}$, and rather than just running a rapid CP search at the root, we will apply it repeatedly at local search nodes within the MIP search tree. To do so efficiently, we present six heuristic criteria to predict the chance for local Rapid Learning to be successful. Our computational experiments indicate that our extended Rapid Learning algorithm significantly speeds up MIP search and is particularly beneficial on highly dual degenerate problems.

1 Introduction

Constraint programming (CP) and integer programming (IP) are two complementary ways of tackling discrete optimization problems. Hybrid combinations of the two approaches have been used for many years, see, e.g., [2,9,10,17,37,42,22]. Both technologies have incorporated *conflict learning* capabilities [21,27,38,1,35] that derive additional valid constraints from the analysis of infeasible subproblems extending methods developed by the SAT community [33].

Conflict learning is a technique that analyzes infeasible subproblems encountered during a tree search algorithm. In a tree search, each subproblem can be identified by its local variable bounds, i.e., by local bound changes that come from branching decisions and propagation at the current node and its ancestors. If propagation detects infeasibility, conflict learning will traverse this chain of decisions and deductions reversely, reconstructing which bound changes led to which other bound changes. In this way, conflict learning identifies explanations for the infeasibility. If it can be shown that a small subset of the bound changes suffices to prove infeasibility, a so-called conflict constraint is generated that can be exploited in the remainder of the search to prune parts of the tree.

In the context of constraint programming, conflict constraints are also referred to as *no-goods*. For binary programs (BPs), i.e., mixed integer (linear) programs for which all variables have domain $\{0, 1\}$, conflict constraints will have the form of *set covering* constraints. These are linear constraints of the form "sum of variables (or their negated form) is greater than or equal to one".

Rapid Learning [13] is a heuristic algorithm for BPs that searches for valid conflict constraints, global bound reductions, and primal solutions. It is based on the observation that a CP solver can typically perform an incomplete search on a few thousand nodes in a fraction of the time that a MIP solver needs for processing the root node. In addition, CP solvers make use of depth-first search, as opposed to the hybrid best-first/depth-first search of MIP solvers, which more rapidly generates strong no-goods. Typically CP solvers do not differentiate the root node from other nodes. They apply fast (at least typically) propagation algorithms to infer new information about the possible values variables can take, and then take branching decisions. In contrast, a MIP solver invests a substantial amount of time at the root node to gather global information about the problem and to initialize statistics that can help for the search. A significant portion of root node processing time comes from the computational effort needed to solve the initial LP relaxation from scratch. Further aspects are the LP resolves during cutting plane generation, strong branching [7] for branching statistic evaluation, and primal heuristics, see, e.g., [11].

The idea of Rapid Learning is to apply a fast CP depth-first branch-and-bound search for a few hundred or thousand nodes, generating and collecting valid conflict constraints at the root node of a MIP search. Using this, the MIP solver is already equipped with the valuable information of which bound changes will lead to an infeasibility, and can avoid them by propagating the derived constraints. Just as important, the partial CP search might find primal solutions, thereby acting as a primal heuristic. Furthermore, the knowledge of conflict constraints can be used to initialize branching statistics, just like strong branching. In this paper, we will extend Rapid Learning to integer programs and to nodes beyond the root.

The remainder of the paper is organized as follows. In Section 2, we provide more background on conflict learning for MIPs, in particular the extension to general integer variables, which is important for our extended Rapid Learning algorithm. In Section 3, we describe details of the Rapid Learning algorithm for general integer programs, extending the work of Berthold et al. [13]. In Section 4, we discuss what special considerations have to be taken when applying Rapid Learning repeatedly at local subproblems during the MIP tree search instead of using it as a onetime global procedure. We introduce six criteria to predict the benefit of local Rapid Learning. Section 5 presents our computational study, in which we apply our extended Rapid Learning algorithm to a set of integer programs from the well-known benchmark sets of Miplib 3, Miplib 2003, and Miplib 2010 [28]. The experiments have been conducted with the constraint integer programming solver SCIP [24] and indicate that a significant speed-up can be achieved for (pure) integer programs, when using Rapid Learning locally. In Section 6, we conclude.

2 Conflict Learning in Integer Programming

A mixed integer program is a mathematical optimization problem defined as follows.

Definition 1 (mixed integer program). *Let $m, n \in \mathbb{Z}_{\geq 0}$. Given a matrix $A \in \mathbb{R}^{m \times n}$, a right-hand-side vector $b \in \mathbb{R}^m$, an objective function vector $c \in \mathbb{R}^n$, a lower and an upper bound vector $l \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{+\infty\})^n$ and a subset $\mathcal{I} \subseteq \mathcal{N} = \{1, \ldots, n\}$, the corresponding* mixed integer program (MIP) *is given by*

$$\begin{array}{lrl} \min & c^{\mathsf{T}} x & \\ s.t. & Ax \leq b & \\ & l_j \leq x_j \leq u_j & \text{for all } j \in \mathcal{N} \\ & x_j \in \mathbb{R} & \text{for all } j \in \mathcal{N} \setminus \mathcal{I} \\ & x_j \in \mathbb{Z} & \text{for all } j \in \mathcal{I}. \end{array} \tag{1}$$

Mixed integer programs can be categorized by the classes of variables that are part of their formulation:

- If $\mathcal{N} = \mathcal{I}$, problem (1) is called a *(pure) integer program (IP)*.
- If $\mathcal{N} = \mathcal{I}$, $l_j = 0, j \in \mathcal{N}$ and $u_j = 1, j \in \mathcal{N}$, problem (1) is called a *(pure) binary program (BP)*.
- If $\mathcal{I} = \emptyset$, problem (1) is called a *linear program (LP)*.

Conflict analysis techniques were originally developed by the artificial intelligence research community [40] and, later extended by the SAT community [33]; they led to a huge increase in the size of problems modern SAT solvers can handle [31,33,43]. The most successful SAT learning approaches use so-called *one-level first unique implication point (1-UIP)* [43] learning which in some sense captures the conflict constraint "closest" to the infeasibility. Conflict analysis also is successfully used in the CP community [25,26,35] (who typically refer to it as no-good learning) and the MIP world [1,21,38,41]. Nowadays, commercial MIP solvers like FICO Xpress [23] employ conflict learning by default.

Constraint programming and mixed integer programming are two complementary ways of tackling discrete optimization problems. Because they have different strengths and weaknesses hybrid combinations are attractive. One notable example, the software SCIP [3], is based on the idea of *constraint integer programming* (CIP) [2,6]. CIP is a generalization of MIP that supports the notion of general constraints as in CP. SCIP itself follows the idea of a very low-level integration of CP, SAT, and MIP techniques. All involved algorithms operate on a single search tree and share information and statistics through global storage of, e.g., solutions, variable domains, cuts, conflicts, the LP relaxation and so on. This allows for a very close interaction amongst CP and MIP (and other) techniques.

There is one major difference between BPs and IPs in the context of Rapid Learning: in IP, the problem variables are not necessarily binary. To deal with this, the concept of a *conflict graph* needs to be extended. A conflict graph gets constructed whenever infeasibility is detected in a local search node; it represents the logic of how the set of branching decisions led to the detection of infeasibility.

More precisely, the conflict graph is a directed acyclic graph in which the vertices[4] represent bound changes of variables, e.g., $x_i \leq \lambda_i$ or $x_i \geq \mu_i$. The conflict graph is built such that when the solver infers a bound change v as a consequence of a set of existing bound changes U, i.e., $U \to v$, then we have an arc (u, v) from each $u \in U$ to v. Bound changes caused by branching decisions are vertices without incoming edges. Finally the conflict graph includes a dummy vertex *false* representing failure which is added when the solver infers unsatisfiability.

Given a conflict graph, each cut that separates the branching decisions from the artificial infeasibility vertex *false* gives rise to a valid conflict constraint. A *unique implication point (UIP)* is an (inner) vertex of the conflict graph which is traversed by all paths from the branching vertices to the conflict vertex. Or, how Zhang et al. [43] describe it: "Intuitively, a UIP is the *single* reason that implies the conflict at [the] current decision level." UIPs are natural candidates for finding small cuts in the conflict graph. The *1-UIP* is the first cut separating the conflict vertex from the branching decisions when traversing in reverse assignment order.

For integer programs, conflict constraints can be expressed as so-called *bound disjunction* constraints:

Definition 2. *For an IP, let $\mathcal{L} \subseteq \mathcal{I}, \mathcal{U} \subseteq \mathcal{I}$ be disjoint index sets of variables, let $\lambda \in \mathbb{Z}^{\mathcal{L}}$ with $l_i \leq \lambda_i \leq u_i$ for all $i \in \mathcal{L}$, and $\mu \in \mathbb{Z}^{\mathcal{U}}$ with $l_i \leq \mu_i \leq u_i$ for all $i \in \mathcal{U}$. Then, a constraint of the form*

$$\bigvee_{i \in \mathcal{L}} (x_i \geq \lambda_i) \vee \bigvee_{i \in \mathcal{U}} (x_i \leq \mu_i)$$

is called a bound disjunction *constraint.*

For details on bound disjunction constraints, see Achterberg [1]. If all involved conflict values λ, μ correspond to global bounds of the variables, the bound disjunction constraint can be equivalently expressed as a knapsack constraint of form

$$\sum_{i \in \mathcal{U}} x_i - \sum_{i \in \mathcal{L}} x_i \leq \sum_{i \in \mathcal{U}} u_i - \sum_{i \in \mathcal{L}} l_i - 1. \tag{2}$$

Note that for BPs all conflicts only involve global bounds.

[4] For disambiguation, we will use the term *vertex* for elements of the conflict graph, as opposed to *nodes* of the search tree.

The power of conflict learning arises because often branch-and-bound based algorithms implicitly repeat the same search in a slightly different context in another part of the tree. Conflict constraints help to avoid redundant work in such situations. As a consequence, the more search is performed by a solver and the earlier conflicts are detected, the greater the chance for conflict learning to be beneficial. Note that conflict generation has a positive interaction with depth-first search. Depth-first search leads to the creation of no-goods that explain why a whole subtree contains no solutions, and hence the no-goods generated by depth-first search are likely to prune more of the subsequent search.

3 Rapid Learning for Integer Programs

The principle motivation for Rapid Learning [13] is the fact that a CP solver can typically search hundreds or thousand of nodes in a fraction of the time that a MIP solver needs for processing the root node of the search tree. Rapid Learning applies a fast CP search[5] for a few hundred or thousand nodes, before starting the MIP search. Using this approach, conflict constraints can be learnt before, and not only during, MIP search. Very loosely speaking: while the aim of conflict learning is to avoid making mistakes a second time, Rapid Learning tries to avoid making them the first time (during MIP search).

Rapid Learning is related to large neighborhood search heuristics, such as RINS and RENS [12,20]. But, rather than doing an incomplete search on a sub-problem using the same (MIP search) algorithm, Rapid Learning performs an incomplete search on the same problem using a much faster algorithm (CP search). Rapid Learning differs from primal heuristics in that it aims at improving the dual bound by collecting information on infeasibility rather than searching for feasible solutions.

Each piece of information collected in a rapid CP search can be used to guide the MIP search or even deduce further reductions during root node processing. Since the CP solver is solving the same problem as the MIP solver

– each generated conflict constraint is valid for the MIP search,
– each global bound change can be applied at the MIP root node,
– each feasible solution can be added to the MIP solver's solution pool,
– the branching statistics can initialize a hybrid MIP branching rule, see [4], and
– if the CP solver completely solves the problem, the MIP solver can abort.

All five types of information may be beneficial for a MIP solver, and are potentially generated by our algorithm which we now describe more formally.

The Rapid Learning algorithm is outlined in Figure 1. Here, $l(P)$ and $u(P)$ are lower and upper bound vectors, respectively, of the problem at hand, P. For the moment we assume P is the root problem, in the next section we will

[5] By CP search we mean applying a depth-first search using only propagation for reasoning, no LP relaxation is solved during the search.

examine the use of Rapid Learning at subproblem nodes. The symbol C refers to a single globally valid conflict constraint explaining the infeasibility of the current subproblem. Rapid Learning is an incomplete CP search: a branch-and-bound algorithm which traverses the search space in a depth-first manner (Line 3), using propagation (Line 4) and conflict analysis (Line 7), but no LP relaxation. Instead, the *pseudo-solution* [2], i.e., an optimal solution of a relaxation consisting only of the variable bounds (Line 5), is used for the bounding step.

Propagation of linear constraints is conducted by the bound strengthening technique of Brearley et al. [18] which uses the residual activity of linear constraints within the local bounds. For special cases of linear constraints, SCIP implements special, more efficient propagators. Knapsack constraints use efficient integer arithmetic instead of floating point arithmetic, and sort by coefficient values to propagate each variable only once. SCIP also features methods to extract clique information about the binary variables of a problem. A clique is a set of binary variables of which at most one variable can take the value 1 in a feasible solution. Clique information can be used to strengthen the propagation of knapsack constraints. Set cover constraints are propagated by the highly efficient two-watched literal scheme [33], which is based on the fact that the only domain reduction to be inferred from a set cover constraint is to fix a variable to 1 if all other variables have already been fixed to 0.

Variable and value selection takes place in Line 14; inference branching [2] is used as branching rule. Inference branching maintains statistics about how often the fixing of a variable led to fixings of other variables, i.e., it is a history rule, its essentially a MIP equivalent of impact-based search [29,36]. Since history rules are often weak in the beginning of the search, we seed the CP solver with statistics that the MIP solver has collected in probing [39] during MIP presolving.

We assume that the propagation routines in Line 4 may also deduce global bound changes and modify the global bound vectors $l(P)$ and $u(P)$. Single-clause conflicts are automatically upgraded to global bound changes in Line 9. Note that it suffices to check constraint feasibility in Line 11, since the pseudo-solution $\bar{x}$ (see Line 5) will always take the value of one of the (integral) bounds for each variable.

Table 1: Settings for Rapid Learning sub-SCIP.

parameter name	value	effect
lp/solvefreq	-1	disable LP
conflict/fuiplevels	1	use 1-UIP
nodeselection/dfs/stdpriority	INT_MAX/4	use DFS
branching/inference/useweightedsum	FALSE	pure inference, no VSIDS
constraints/disableenfops	TRUE	no extra checks
propagating/pseudoobj/freq	-1	no objective propagation
conflict/maxvarsfac	0.05	only short conflicts
history/valuebased	TRUE	extensive branch. statistics

Fig. 1: Rapid Learning algorithm

```
Input   : IP $P$ as in (1) (with $\mathcal{R} = \emptyset$),
          node limit $\lim_{\text{node}}$,
          primal bound $\bar{c}$ for $P$ (might be $\infty$)
Output: set of valid conflict constraints $\mathcal{L}_C$ for $P$,
        valid global domain box $[l, u]$ for $P$,
        feasible solution $\tilde{x}$ for $P$ or $\emptyset$
1  $\mathcal{L} \leftarrow \{P\}$, $\text{n}_{\text{node}} \leftarrow 0$, $\mathcal{L}_C \leftarrow \emptyset$, $\tilde{x} \leftarrow \emptyset$;
2  while $\mathcal{L} \neq \emptyset \wedge \text{n}_{\text{node}} < \lim_{\text{node}}$ do
3      $\tilde{P} \leftarrow$ select_dfs($\mathcal{L}$), $\mathcal{L} \leftarrow \mathcal{L} \setminus \tilde{P}$, $\text{n}_{\text{node}} \leftarrow \text{n}_{\text{node}} + 1$;
4      $[l(\tilde{P}), u(\tilde{P})] \leftarrow$ propagate($[l(\tilde{P}), u(\tilde{P})]$);
5      $\bar{x} \leftarrow \text{argmin}\{c^{\mathsf{T}} x \mid x \in [l(\tilde{P}), u(\tilde{P})]\}$;

       /* analyze infeasible subproblem, potentially store globally
          valid conflict constraint                                  */
6      if $[l(\tilde{P}), u(\tilde{P})] = \emptyset$ or $c(\bar{x}) \geq \bar{c}$ then
7          $C \leftarrow$ analyze($\tilde{P}$);
8          if $C \neq \emptyset$ then $\mathcal{L}_C \leftarrow \mathcal{L}_C \cup \{C\}$;
9          if $|C| = 1$ then tighten($[l(P), u(P)]$);
10         continue;

       /* check for new incumbent solution                           */
11     if $A\bar{x} \leq b$ and $c^T \bar{x} < \bar{c}$ then
12         $\tilde{x} \leftarrow \bar{x}$, $\bar{c} \leftarrow c^T \bar{x}$;
13         continue;

14     $(x_i, v) \leftarrow$ select_infer($\tilde{P}$, $\bar{x}$);
15     $\tilde{P}_l \leftarrow \tilde{P} \cup \{x_i \leq v\}$, $\tilde{P}_r \leftarrow \tilde{P} \cup \{x_i \geq v\}$;
16     $\mathcal{L} \leftarrow \mathcal{L} \cup \{\tilde{P}_l, \tilde{P}_r\}$;
17 return $(\mathcal{L}_C, [l(P), u(P)], \tilde{x})$;
```

Our implementation of the Rapid Learning heuristic uses a secondary SCIP instance to perform the CP search. Only a few parameters need to be altered from their default values to turn SCIP into a CP solver, an overview is given in Table 1. Most importantly, we disabled the LP relaxation and use a pure depth-first search with inference branching (but without any additional tie breakers). Further, we switch from All-UIP to 1-UIP in order to generate only one conflict per infeasibility. This is a typical behavior of CP solvers, but not for MIP solvers. Expensive feasibility checks and propagation of the objective function as a constraint are also avoided.

In order to avoid spending too much time in Rapid Learning, the number of nodes explored during the CP search is limited to at most 5000. The actual number of allowed nodes is determined by the number of simplex iterations iter_{LP}

performed so far in the main SCIP but at least 500, i.e.,

$$\text{lim}_{\text{node}} = \min\{5000, \max\{500, \text{iter}_{\text{LP}}\}\}.$$

The idea is to restrict Rapid Learning more rigorously for problems where processing of a single MIP node is cheap already. The number of simplex iterations is a deterministic estimate for node processing cost.

We aim to generate short conflict constraints, since these are most likely to frequently trigger propagations in the upcoming MIP search. Thus, we only collect conflicts that contain at most 5 % of the problem variables. Finally, we adapt the collection of branching statistics such that history information on general integer variables are collected per value in the domain rather than having one counter for down- and one for up-branches regardless of the value on which was branched. This can be essential for performing an efficient CP search on general integer variables, and was a building block that enabled us to use Rapid Learning on IPs rather than solely on BPs, as in [13].

In addition to the particular parameters listed in Table 1, we set the emphasis[6] for presolving to "fast". Emphasis settings for cutting are not necessary, since no LP relaxation is solved, from the armada of primal heuristics only a few are applied that do not require an LP relaxation, see [5]. Note that since Rapid Learning will be called at the end of the MIP root node, or even locally, see next Section, the problem that the CP solver considers has already been presolved, might contain cutting planes as additional linear constraints and have an objective cutoff constraint if a primal solution has been found by a primal heuristic during root node processing.

4 Local Rapid Learning

The original Rapid Learning algorithm [13] was used as part of a root preprocessing, i.e., for every instance it was run exactly once at the end of the root node. But only running Rapid Learning at the root limits its effectiveness. We now discuss the factors that arise when we allow Rapid Learning to be run at local nodes inside the search tree

When running in the root only all information returned by the CP solver is globally valid, and the overhead to maintain the information gathered by Rapid Learning is negligible [13]. In contrast, when applying Rapid Learning at a local node within the tree conflicts and bound changes will only be locally valid in general. Since Rapid Learning uses a secondary SCIP instance to perform the CP search, all local information of the current node becomes part of the initial problem formulation for the CP search. Thus, conflicts gathered by Rapid Learning do not include bound changes made along the path from the root to the current node, they are simply considered as valid for this local node. As a consequence, these conflicts will only be locally valid and hence only applied to the current

[6] In SCIP, emphasis settings correspond to a group of individual parameters being changed.

node of the MIP search. Using an assumption interface [34], local conflicts could be lifted to be globally valid. However, this is subject to future investigation and not considered in the current implementation of Rapid Learning.

In practice, all local information needs to be maintained when switching from one node of the tree to another. In CP solvers, switching nodes is typically very cheap, because depth-first search is used. However, a MIP solver frequently "jumps" within the tree. Therefore, two consecutively processed nodes can be quite different. In what follows, we will refer to the time spent for moving from one node to another node as *switching time*. The switching time can be used as an indicator to quantify the overhead introduced by all locally added information found by Rapid Learning.

To ensure that the amount of locally added information does not increase the switching time too much, we apply Rapid Learning very rarely by using a exponentially decreasing frequency of execution. Rapid Learning is executed at every node of depth d with

$$\log_\beta(d/f) \in \mathbb{Z}, \tag{3}$$

where β and f are two parameters to control the speed of decrease. For example, if $\beta = 1$ Rapid Learning is executed at every depth $d = i \cdot f$ with $i \in \mathbb{Z}_+$.

Unfortunately, the amount of locally valid information produced by Rapid Learning still leads to an increase of switching time by 21 %. Consequently, the overall performance decreased by 20 % in our first experiments. At the same time the number of explored branch-and-bound nodes decreased by 16 %. This indicates the potential gains possible using local Rapid Learning.

To control at which subproblem Rapid Learning is applied we propose six criteria to forecast the potential of Rapid Learning. These criteria aim at identifying one of two situations. The first is to estimate whether the (sub)problem is infeasible or a pure feasibility problem. In these cases propagating conflicts is expected to be particularly beneficial. The second is to estimate the dual degeneracy of a problem. In this case, VSIDS branching statistics are expected to be particularly beneficial. The VSIDS [31] (variable state independent decaying sum) statistics takes the contribution of every variable (and its negated complement) in conflict constraints found so far into account. For every variable, the number of clauses (in MIP speaking: conflict constraints) the variable is part of is counted. In the remainder of the search the VSIDS are periodically scaled by a predefined constant. By this, the weight of older clauses is reduced over time and more recent observations have a bigger impact.

A basic solution of an LP is called *dual degenerate*, when it has nonbasic variables with zero reduced costs. One can define the dual degeneracy of a MIP as the average number of nonbasic variables with zero reduced costs appearing in a basic solution of its LP relaxation. The higher the dual degeneracy, the higher the chance that the LP objective will not change by branching and hence many of the costs involved in the pseudo-cost computation are zero. Therefore, for highly dual degenerate problems, using other branching criteria, such as VSIDS or inference scores, is crucial for solving the problem.

We now describe the six criteria we use to identify infeasible or dual degenerate problems, already using the criteria abbreviations from the tables in Section 5:

Criterion I: Dual Bound Improvement. During the tree search a valid lower bound for each individual subproblem is given by the respective LP solution. A globally valid lower bound is given by the minimum over all individual lower bounds. This global bound is called the *dual bound*. If the dual bound has not changed after processing a certain number of nodes, i.e., the dual bound is equal to the lower bound of the root node, it might be the case that the MIP lies inside a level plane of the objective, i.e., all feasible LP (and MIP) solutions will have the same objective. In other words, the instance might be a feasibility instance for which Rapid Learning was already shown to be very successful [13]. Feasibility instances are typically highly dual degenerate. The `dualbound` criterion means to call local Rapid Learning if the dual bound never changed during the MIP search.

Criterion II: Leaves Pruned by Infeasibility or Exceeding the Cutoff bound. During the tree search every leaf node either provides a new incumbent solution (the rare case), is proven to be infeasible or to exceed the current cutoff bound which is given by the incumbent solution. The ratio of the latter two cases is used in SCIP's default branching rule. *Hybrid branching* [4] combines pseudo-costs, inference scores, and conflict information into one single branching score. The current implementation in SCIP puts a higher weight on conflict information, e.g., VSIDS [31], and a lower weight on pseudo-costs when the ratio of infeasible and cutoff nodes is larger than a predefined threshold. The `leaves` criterion means to call local Rapid Learning if the ratio of infeasible leaves over those exceeding the cutoff bound is larger than 10. The rationale is that we expect (local) conflicts to be most beneficial, when infeasibility detection appears to be the main driver for pruning the tree.

Criterion III: LP Degeneracy. As mentioned above, the more nonbasic variables are dual degenerate, the less information can be gained during strong branching or pseudo-cost computation. As a consequence, Berthold et al. [14] introduced a modification to strong branching that considers the dual degeneracy of the LP solution. In rough terms, if either the share of dual degenerate nonbasic variables or the variable-constraint ratio of the optimal face exceed certain thresholds, strong branching will be deactivated. We adapt this idea of using the dual degeneracy of the current LP solution. The `degeneracy` criterion means to call local Rapid Learning if more than 80 % of the nonbasic variables are degenerate or the variable-constraint ratio of the optimal face is larger than 2, as proposed in [14]. In both cases we expect that "strong conflict generation" will be useful.

Criterion IV: (Local) Objective Function. If all variables with non-zero objective coefficients are fixed at the local subproblem, i.e., the objective is constant, Criteria I and II will apply: every LP solution is fully dual degenerate and the only possibility to prune a leaf node is by infeasibility. If there are only very few unfixed variables with nonzero objective are left, the criteria might not apply. However, it is likely that the targeted situations occur frequently in the tree rooted at the current subproblem, at the latest, when all the variables occurring in the objective are fixed. The `obj` criterion means to call local Rapid Learning once the objective support is small enough, in anticipation of the current subproblem turning into a feasibility problem. In our implementation we apply this criterion very conservatively, and call Rapid Learning only if the local objective is zero.

Criterion V: Number of Solutions. The most obvious evidence, and indeed a necessary one, that a MIP instance is infeasible, is that no feasible solution has been found during the course of the MIP search. Note that for most (feasible) MIP instances, primal heuristics find a feasible solution at the root node [11] or at the latest during the first dive in the branch-and-bound. The `nsols` criterion means to call local Rapid Learning if no feasible solution has been found so far.

Criterion VI: Strong Branching Improvements. In the beginning of the tree search it is very unlikely that enough leaf nodes are explored to reliably guess whether the actual MIP is a feasibility instance. Therefore, we consider the subproblems evaluated during strong branching, which are concentrated at the top of the search tree. Similarly to Criterion II, we compute the ratio between the number of strong branching problems that gave no improvement in the objective or went infeasible to the number of strong branching problems where we observed an objective change. The `sblps` criterion means to call local Rapid Learning if this ratio exceeds a threshold of 10, hence strong branching does not appear to be efficient for generating pseudo-cost information.

In addition to the exponentially decreasing frequency and the six criteria above, we applied the following three changes to the original implementation of Rapid Learning used in [13].

– We limited the number of conflict constraints transferred from Rapid Learning back to the original search tree to ten. This corresponds to the SCIP parameter `conflict/maxconss` for the maximal allowed number of added conflicts per call of conflict analysis. We greedily use the shortest conflicts.
– We prefer conflict constraints that have a linear representation over bound disjunction constraints (see Definition 2).
– To exploit performance variability [19,30] every CP search is initialized with a different pseudo-random seed.

5 Computational results

To evaluate how local Rapid Learning impacts IP solving performance we used the academic constraint integer programming solver SCIP 6.0 [24] (with SoPlex 4.0 as LP solver) and extended the existing code of Rapid Learning. The original implementation of Rapid Learning was already shown to significantly improve the performance of SCIP 1.2.0.5 on pure binary instances [13]. In this setting, Rapid Learning was applied exactly once at the root node. However, during the last eight years SCIP has changed in many places. In SCIP 6.0, Rapid Learning is deactivated by default, since it led to a big performance variability.

Therefore, we use SCIP without Rapid Learning (as it is the current default) as a baseline. We will refer to this setting as `default`. In our computational experiments we evaluate the impact of local Rapid Learning if one or more of the criteria described in Section 4 are fulfilled. In the following, we will refer to the criteria I–VI as `dualbound`, `leaves`, `degeneracy`, `obj`, `nsols`, and `sblps`, respectively. Within the tree, Rapid Learning is applied with an exponentially decreasing frequency (see Section 4). In our experiments, we used $f = 5$ and $\beta = 4$, i.e., Rapid Learning is called at depths d with $log_4(d/5) \in \mathbb{Z}$, i.e., $d = 0, 5, 20, 80, 320 \ldots$, if one of the six criteria is fulfilled.

As a test set we used all pure integer problems of MIPLIB 3 [16], MIPLIB 2003 [8] and the MIPLIB 2010 [28] benchmark set. This test set consists of 71 publicly available instances, which we will refer to as `MMM-IP`. The experiments were run on a cluster of identical machines, each with an Intel Xeon E5-2690 with 2.6 GHz and 128 GB of RAM; a time limit of 3600 seconds was set.

In a first experiment we evaluated the efficacy of each individual criterion and global Rapid Learning as published in [13]. Aggregated results are shown in Table 2, section Exp.1. For detailed results we refer to the appendix of [15]. For every setting, the table shows the number of solved instances out of 71 (**solved**), shifted geometric means [3] of the absolute solving time in seconds (**time**, shift = 1) and number of explored nodes (**nodes**, shift = 100), as well as the relative solving time ($\mathbf{time_Q}$) and number of nodes ($\mathbf{nodes_Q}$) w.r.t. to `default` as a baseline. Local Rapid Learning without any of the presented criteria (`nochecks`) leads to a performance decrease of 21 % on the complete test set `MMM-IP` compared to `default`. Always applying Rapid Learning only at the root (`onlyroot`), which corresponds to Rapid Learning as published in [13], leads to slowdown of 10 % but solves one instance more. For this settings, we could observe a performance decrease of 29 % on the group of instances that are not affected[7] by Rapid Learning. To avoid a computational overhead and performance variability on instances where Rapid Learning is not expected to be beneficial, we apply the criteria `degeneracy`, `obj`, and `nsols` at the root node, too. Afterwards, the performance decrease of global Rapid Learning reduced to 3 %. The computational results indicate that almost all individual criteria are useful on their own. The solving time and generated nodes can be reduced by up to 7 % and 14 %, respectively, on the complete test set of 71. The exception is the `obj` criterion, which leads to a slowdown of 2 %, but solves one more instance than `default`. These results can be confirmed when repeating the experiments with five different random seeds [15]. On the group of affected instances the solving time can be reduced by up to 21 %, using the `leaves` criterion. The number of generated

[7] An instance is called affected when the solving path changes.

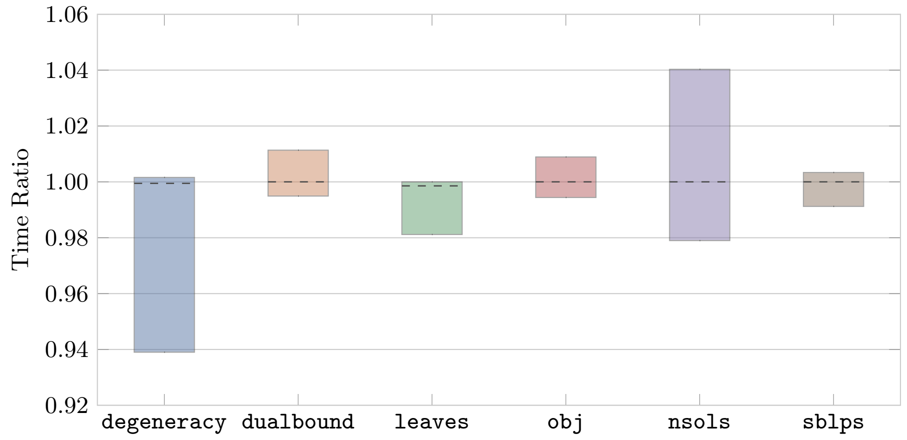

Fig. 2: Box-plot of the performance ratios of the individual criteria compared to `default` on the set of affected instances.

nodes can be reduced by up to 39 % (for `degeneracy`) on the same group of instances.

The impact of the individual criteria on the solving time is illustrated in Figure 2. For each criterion, the box plot [32] shows the median (dashed line), and the 1st and 3rd quartile (shaded box) of all observations. For all criteria the median time ratio is at most one; only for `degeneracy` and `leaves` the median is strictly smaller than one. Hence, these two settings improve the performance on more than 50 % of the affected instances. Furthermore, `degeneracy` and `leaves` have by far the smallest 1st and 3rd quartile, indicating that the corresponding settings often improve performance and rarely deteriorate it.

Grouping all instances of `MMM-IP` based on the degeneracy at the end of the root node shows the importance of this criterion. On the group of instances where at least 1 % of the variables is dual degenerate at the end of the root node Rapid Learning leads to a performance improvement of 9.1 %. On all instances where at least 80 % of the variable are dual degenerate at the root node, we could observe a reduction of solving time by 28.8 %. Note that this was one of the two thresholds for the `degeneracy` criterion.

In a second experiment (Table 2, section Exp.2) we combined all individual criteria. Combining two or more criteria leads to more aggressive version of Rapid Learning since it runs if at least one of the chosen criteria is satisfied. The two (out of fifteen) best pairwise combinations as well as the (most aggressive) combination of all six criteria are shown in Table 2. Interestingly, no combined setting is superior to `degeneracy`. The combination of `degeneracy` and `leaves`, which were the two outstanding criteria in the individual test, performs almost the same as the `degeneracy` criterion alone. These results can be confirmed when repeating the experiments with five different random seeds [15].

Table 2: Computational results for every individual heuristic criterion on `MMM-IP`.

		solved	**time**	**nodes**	**time$_\mathbf{Q}$**	**nodes$_\mathbf{Q}$**
Exp.1	`default`	61	50.34	2428	–	–
	`degeneracy`	**62**	47.23	2078	0.938	**0.856**
	`dualbound`	61	49.00	2284	0.973	0.941
	`leaves`	61	46.88	2199	**0.931**	0.906
	`obj`	**62**	51.51	2432	1.023	1.002
	`nsols`	61	47.95	2194	0.952	0.904
	`sblps`	61	48.14	2178	0.956	0.897
	`nochecks`	59	60.81	1995	1.208	0.822
	`onlyroot`	**62**	55.71	2476	1.107	1.020
Exp.2	`degeneracy` + `leaves`	**62**	47.37	2080	0.941	**0.857**
	`leaves` + `obj`	**62**	47.26	2167	**0.939**	0.893
	`all6criterion`	**62**	47.88	2104	0.951	0.867

For a final experiment we choose `degeneracy` as the best criterion, since it was one of two criteria that solved an additional instance, clearly showed the best search reduction, and was a close second to `leaves` with respect to running time. Our final experiment evaluates the impact of the individual information gained from local Rapid Learning. To this end, we individually deactivated transferring variable bounds, conflict constraints, inference information, and primal feasible solutions (see Table 3). This experiment indicates that primal solutions are the most important information for the remainder of the MIP search. When ignoring solutions found during the CP search, the overall solving time increased by 10.4 % (`primsols`). When ignoring conflict constraints, the original motivation of Rapid Learning, solving time increased by 2.4 % (`conflicts`). Both transferring variable bounds and inference information proved beneficial, with a 2.1 % (`variablebounds`) and 2.8 % (`infervals`) impact on performance, respectively. To take performance variability into account, we repeated the experiment with five different random seeds, see [15] for detailed results. This experiment indicated that conflict constraints are the second most important criterion. Over five seeds the solving time increased by 9.9 % (`primsols`), 4.4 % (`conflicts`), 1.4 % (`variablebounds`), and 0.6 % (`infervals`). It is not surprising that finding primal solutions has the largest effect. Firstly, they are applied globally, in contrast to bound changes and conflicts. Secondly, highly dual degenerate problems are known to be cumbersome not only for MIP branching but also for primal heuristics [11], which means that solution-generating procedures that do not rely on solving LPs are particularly promising for such problems.

6 Conclusion

In this paper, we extended the idea of Rapid Learning [13]. Firstly, we generalized Rapid Learning to integer programs and described the details that were

Table 3: Performance impact of individual gained information on `MMM-IP`.

		solved	time	nodes	$\mathbf{time_Q}$	$\mathbf{nodes_Q}$
Exp.3	`degeneracy`	62	47.23	2078	–	–
	`variablebounds`	62	48.23	2180	1.021	1.049
	`conflicts`	63	48.38	2213	1.024	1.065
	`infervals`	62	48.53	2230	1.028	1.073
	`primsols`	62	52.15	2400	1.104	1.155

necessary for doing so: value-based inference branching, additional propagators and generalized conflict constraints, most of which were already available in SCIP. Secondly, we applied Rapid Learning repeatedly during the search. This generates a true hybrid CP/MIP approach, with two markedly different search strategies communicating information forth and back. To this end, we introduced six heuristic criteria to decide when to start local Rapid Learning. Those criteria are based on degeneracy information, branch-and-bound statistics, and the local structure of the problem. Our computational experiments showed a speed-up of up to 7 % when applying local Rapid Learning in SCIP. Calling local Rapid Learning depending on the local degree of dual degeneracy is the best strategy found in our experiments.

Interesting future work in this direction includes: extending the CP search to generate global conflicts at local nodes using an assumption interface, running the CP search in a parallel thread where whenever the MIP solver moves to a new node the CP search restarts from that node, and extending the method to handle problems that include continuous variables.

Acknowledgments

The work for this article has been partly conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM). We thank the anonymous reviewers for their valuable suggestions and helpful comments.

References

1. T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007.
2. T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.
3. T. Achterberg. SCIP: Solving Constraint Integer Programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
4. T. Achterberg and T. Berthold. Hybrid branching. In W.-J. van Hoeve and J. N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009*, volume 5547 of *LNCS*, pages 309–311. Springer Berlin Heidelberg, May 2009.

5. T. Achterberg, T. Berthold, and G. Hendel. Rounding and propagation heuristics for mixed integer programming. In D. Klatte, H.-J. Lüthi, and K. Schmedders, editors, *Operations Research Proceedings 2011*, pages 71–76. Springer Berlin Heidelberg, 2012.
6. T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In L. Perron and M. A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 5th International Conference, CPAIOR 2008*, volume 5015 of *LNCS*, pages 6–20. Springer Berlin Heidelberg, May 2008.
7. T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.
8. T. Achterberg, T. Koch, and A. Martin. Miplib 2003. *Operations Research Letters*, 34(4):361–372, 2006.
9. E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. SCIL – symbolic constraints in integer linear programming. In *Algorithms – ESA 2002*, pages 75–87, 2002.
10. I. D. Aron, J. N. Hooker, and T. H. Yunes. SIMPL: A system for integrating optimization techniques. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, volume 3011 of *LNCS*, pages 21–36, 2004.
11. T. Berthold. *Heuristic algorithms in global MINLP solvers.* PhD thesis, Technische Universität Berlin, 2014.
12. T. Berthold. RENS – the optimal rounding. *Mathematical Programming Computation*, 6(1):33–54, 2014.
13. T. Berthold, T. Feydy, and P. J. Stuckey. Rapid learning for binary programs. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010*, volume 6140 of *LNCS*, pages 51–55. Springer Berlin Heidelberg, June 2010.
14. T. Berthold, G. Gamrath, and D. Salvagnin. Cloud Branching. In preparation.
15. T. Berthold, P. J. Stuckey, and J. Witzig. Local Rapid Learning for Integer Programs. Technical Report 18-56, ZIB, Takustr. 7, 14195 Berlin, 2018.
16. R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. Savelsbergh. An updated mixed integer programming library: Miplib 3.0. Technical report, 1998.
17. A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.
18. A. Brearley, G. Mitra, and H. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83, 1975.
19. E. Danna. Performance variability in mixed integer programming. Presentation slides from MIP 2008 workshop in New York City. http://coral.ie.lehigh.edu/~jeff/mip-2008/program.pdf, 2008.
20. E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2004.
21. B. Davey, N. Boland, and P. J. Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS Journal of Computing*, 14(4):373–386, 2002.
22. T. Davies, G. Gange, and P. J. Stuckey. Automatic logic-based Benders decomposition with minizinc. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI-17)*, pages 787–793. AAAI Press, 2017. https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14489.

23. FICO Xpress Optimizer. http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx.
24. A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical Report 18-26, ZIB, Takustr. 7, 14195 Berlin, 2018.
25. N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, 2000.
26. G. Katsirelos and F. Bacchus. Generalised nogoods in CSPs. In *Proceedings of AAAI-2005*, pages 390–396, 2005.
27. G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396. AAAI Press / The MIT Press, 2005.
28. T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.
29. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proc. of CP*, pages 342–356, Autriche, 1997. Springer.
30. A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. In *Theory Driven by Influential Applications*, pages 1–12. INFORMS, 2013.
31. J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions of Computers*, 48:506–521, 1999.
32. R. McGill, J. W. Tukey, and W. A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.
33. M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC'01*, pages 530–535, 2001.
34. A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *Proc SAT*, volume 7317 of *LNCS*, pages 242–255. Springer, 2012.
35. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
36. P. Refalo. Impact-based search strategies for constraint programming. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 557–571, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
37. R. Rodosek, M. G. Wallace, and M. T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86(1):63–87, 1999.
38. T. Sandholm and R. Shields. Nogood learning for mixed integer programming. In *Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization, Montréal*, 2006.
39. M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.
40. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

41. J. Witzig, T. Berthold, and S. Heinz. Experiments with conflict analysis in mixed integer programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 14th International Conference, CPAIOR 2017*, volume 10335 of *LNCS*, pages 211–220. Springer Berlin Heidelberg, May 2017.
42. T. H. Yunes, I. D. Aron, and J. N. Hooker. An integrated solver for optimization problems. *Operations Research*, 58(2):342–356, 2010.
43. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.