FULL LENGTH PAPER

Structure-driven fix-and-propagate heuristics for mixed integer programming

Gerald Gamrath[1] · Timo Berthold[2] · Stefan Heinz[2] · Michael Winkler[3]

Abstract

Primal heuristics play an important role in the solving of mixed integer programs (MIPs). They often provide good feasible solutions early and help to reduce the time needed to prove optimality. In this paper, we present a scheme for start heuristics that can be executed without previous knowledge of an LP solution or a previously found integer feasible solution. It uses global structures available within MIP solvers to iteratively fix integer variables and propagate these fixings. Thereby, fixings are determined based on the predicted impact they have on the subsequent domain propagation. If sufficiently many variables can be fixed that way, the resulting problem is solved first as an LP, and then as an auxiliary MIP if the rounded LP solution does not provide a feasible solution already. We present three primal heuristics that use this scheme based on different global structures. Our computational experiments on standard MIP test sets show that the proposed heuristics find solutions for about 60% of the instances and by this, help to improve several performance measures for MIP solvers, including the primal integral and the average solving time.

✉ Gerald Gamrath
   gamrath@zib.de

   Timo Berthold
   timoberthold@fico.com

   Stefan Heinz
   stefanheinz@fico.com

   Michael Winkler
   winkler@gurobi.com

1   Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany

2   Fair Isaac Germany GmbH, c/o ZIB, Takustr. 7, 14195 Berlin, Germany

3   Gurobi GmbH, c/o ZIB, Takustr. 7, 14195 Berlin, Germany

1 Introduction

Mixed integer linear programming problems (MIPs) minimize (or maximize) a linear objective function subject to linear constraints and integrality restrictions on some or all of the variables. More formally, a MIP is stated as follows:

$$z_{MIP} = \min\{c^T x : Ax \le b, \ell \le x \le u, x \in \mathbb{R}^n, x_i \in \mathbb{Z} \text{ for all } i \in \mathcal{I}\} \qquad (1)$$

with objective function $c \in \mathbb{R}^n$, constraint matrix $A \in \mathbb{R}^{m \times n}$, and constraint right-hand sides $b \in \mathbb{R}^m$. We allow lower and upper bounds $\ell, u \in \bar{\mathbb{R}}^n$ on variables, where $\bar{\mathbb{R}} := \mathbb{R} \cup \{\pm\infty\}$, and the restriction of a subset of variables $\mathcal{I} \subseteq \mathcal{N} = \{1, \ldots, n\}$ to integral values. In the remainder of this paper, we denote by $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$ a MIP of form (1) in dependence on the provided data.

Very powerful generic solvers for MIPs have been developed over the last decades, which are used widely in research and practice [8,22,50]. These solvers are based on the branch-and-bound algorithm [26,48], which is intertwined with various extensions, see [8].

Branch-and-bound profits directly from finding good solutions as early as possible. On the one hand, these solutions originate from integral solutions to the linear programming (LP) relaxation. The LP relaxation $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \emptyset)$ is obtained from (1) by omitting the integrality restrictions and is repeatedly solved for (sub-)problems during the branch-and-bound search to provide solution candidates and lower bounds. On the other hand, so-called *primal heuristics* try to construct new feasible solutions or improve existing ones. Primal heuristics are incomplete methods without any success or quality guarantee which nevertheless are beneficial on average. For more details on primal heuristics, we refer to [14,16,32]. In this paper, we introduce three novel heuristics which combine a fix-and-propagate scheme [8,20] with the large neighborhood search (LNS) paradigm, see [27]. The former is typically used for before-LP heuristics and iteratively fixes a variable and propagates this change to apply all implied changes to the domains of other variables. The latter defines a sub-problem, the neighborhood, by adding restrictions to the problem, and then solves this sub-problem as a MIP. A more detailed discussion of these heuristic concepts is given in Sect. 2.

By modeling a specific problem as a MIP and solving it with a MIP solver, one profits from the decades of developments within this area. However, knowledge about the structure of the problem which could be exploited by a problem specific approach can hardly be fed into a MIP solver due to the generality of the approach. MIP solvers try to partially compensate this by detecting some common structures within the problem and exploiting them in the solving process. Examples for this are multi-commodity flow subproblems [7] and permutation structures [59]. This detection is often done in the presolving phase, which is a preprocessing step trying to remove redundancies from the model and to tighten the formulation. An overview of different global structures in MIP solvers and details about three of them, the clique table, the variable bound graph, and the variable locks, are given in Sect. 3.

The heuristics presented in this paper use these global structures to determine the fixing order and fixing values for the variables. While it is a known approach in MIP heuristics to apply domain propagation to identify the direct consequences of a

fixing and tighten domains of other variables accordingly, our new heuristics take a step further than existing methods and make domain propagation their driving force. Rather than supporting the fixing scheme by domain propagation, the heuristics base their fixing scheme on the implications that a variable fixing will have, predicted via global structures. After the fix-and-propagate step, the remaining problem is solved as an LP and the LP solution is rounded. If this did not provide a feasible solution already, the problem obtained after the fix-and-propagate phase is solved as an auxiliary MIP (called sub-MIP in the following).

A detailed description of the general scheme of the structure-driven fix-and-propagate heuristics is discussed in Sect. 4. The three instantiations of the heuristic scheme for the three discussed global structures are presented in Sects. 5 to 7. They have been implemented within the academic MIP solver SCIP [2]. The impact of the heuristics on the overall solving process of SCIP is evaluated by the computational experiments presented in Sect. 8. Finally, Sect. 9 gives our conclusions and an outlook.

Previous work by the authors [36] introduced prior versions of primal heuristics based on the clique table and the variable bound graph and gave a preliminary computational evaluation of the heuristics. The present paper extends this work significantly. First, a third heuristic is introduced which is based on variable locks. Second, the two former heuristics have been significantly improved. The clique-based fixing scheme works directly on the cliques now rather than computing a clique partition. Cliques are also taken into account for the topological sorting of the variable bound graph (see Sect. 6). In all heuristics, infeasible fixings are now undone by a backtracking step in order to continue the fixing phase. This leads to higher fixing rates and more solutions being found by LP solving rather than the more expensive sub-MIP solve. Finally, we perform a thorough computational study to analyze the effort and success rates of the heuristics.

2 Primal heuristics and large neighborhood search for MIP

Primal heuristics are algorithms that try to find feasible solutions of good quality for a given optimization problem within a reasonably short amount of time. There is typically no guarantee that they will find any solution, let alone an optimal one.

For mixed integer linear programs (MIPs) it is well known that general-purpose primal heuristics like the Feasibility Pump [3,29,34] can find high-quality solutions for a wide range of problems. Over time, primal heuristics have become a substantial ingredient of state-of-the-art MIP solvers [14,24].

The last 15 years have seen various publications on general-purpose heuristics for MIPs, including [3,4,9,12,13,17–19,21,28,33,34,38,39,41–43,46,49,56,58,59,61]. For an overview, see [14,16,51].

Large neighborhood search lies at the heart of many MIP heuristics, such as Local Branching [30], RINS [27], Crossover [14], DINS [39], RENS [17], Proximity Search [33], and Analytic Center Search [21]. The main idea of LNS is to restrict the search for "good" solutions to a neighborhood centered at a particular reference point. This is typically the incumbent or another feasible solution, but it may as well be an infeasible integer point or a partial solution, see [31]. The hope is that the restricted search space

makes the sub-problem much easier to solve, while still providing solutions of high quality. Of course, these restricted sub-problems do not have to be solved to optimality; they are mainly searched for an improving solution.

DINS, RINS, RENS, Crossover, and Analytic Center Search define their neighborhoods by variable fixings. LNS heuristics that are based on variable fixings suffer from an inherent conflict: the original search space should be significantly reduced; thus, it seems desirable to fix a large number of variables. At the same time, the more variables get fixed, the higher is the chance that the sub-problem does not contain any improving solution or even becomes infeasible.

The present paper addresses this issue by applying a fix-and-propagate scheme that is guided by global structures. The hope is that this scheme maintains feasibility of the restricted search space while still reducing it significantly through the means of domain propagation. The fix-and-propagate procedure can potentially find a complete assignment of the variables or end up with an empty search space; in either case, the suggested heuristics will terminate. If neither is the case, a large neighborhood search will be conducted on the restricted search space.

3 Global structures in MIP solvers

Mixed integer programs are restricted to linear constraints, a linear objective, and integrality conditions. This makes MIP solvers easily accessible and exchangeable if a MIP model is at hand. From the modeling point of view, however, there is hardly any possibility to pass additional structural information to a solver, e.g., that and how certain model variables are connected via the combinatorics of a network structure. Modern MIP solvers aim at detecting common structures within a model and use them for heuristics, cutting plane separation or presolving. Examples of such global information include cliques, implications, and variable bound constraints (see [2,10,54]), multi-commodity flow structures [7], permutation structures [59], and symmetries [55]. Multi-commodity flows and permutations are examples of rather specific constructs that occur in only a handful of models—but are crucial for solving them. Cliques and variable bound constraints, in contrast, can be found in many MIPs of different types. So far, they have been mainly used for cutting plane generation and domain propagation, see, e.g., [2]. The remainder of the section explains three global structures in more detail: the clique table and the variable bound graph, both detected during presolving, and the variable locks, which capture how many constraints restrict a variable.

3.1 The clique table

A *clique* is a set $\mathcal{C}$ of binary variables of which at most one variable can be set to one, see [2,10]. A clique can be given directly as a linear inequality $\sum_{i \in \mathcal{C}} x_i \leq 1$ or derived from more general constraints such as knapsacks: given a constraint $\sum_{i \in J} w_i x_i \leq C$ with binary variables $x_i, i \in J$, each subset $\mathcal{C} \subseteq J$ for which $w_j + w_k > C$ for all $(j, k) \in \mathcal{C} \times \mathcal{C}$ defines a clique. In addition, presolving techniques such as probing [60]

can be used to detect cliques which are given implicitly and cannot be extracted directly from a single model constraint.

Similarly, *negated cliques* [62] can be extracted from the problem. A negated clique is a set of binary variables of which at most one variable can be set to zero. When combining these two types of cliques, we obtain the general form $\sum_{i \in \mathcal{C}^+} x_i + \sum_{i \in \mathcal{C}^-} (1 - x_i) \leq 1$. This can be transferred back to the first case by introducing *negated variables* of the form $x_i' := 1 - x_i$ for all $i \in J^-$. For the ease of presentation, we will therefore only consider simple cliques in the remainder of this paper.

In modern MIP solvers, the set of all detected cliques is stored in the so-called *clique table*. This global structure forms a relaxation of the MIP and is used by solver components, e.g., to create clique cuts [45] or to deduce stronger reductions in presolving and propagation [5,60]. In Sect. 5, we will show how the clique table can be used to guide a fix-and-propagate heuristic.

3.2 The variable bound graph

Variable bound constraints are linear inequalities which contain exactly two variables, see [2,54]. Typical examples for such constraints are precedence constraints on start time variables in scheduling or big-M constraints modeling fixed-costs in production planning. Depending on the sign of the coefficient, the variables bound each other. For example, a constraint $ax + by \geq c$ with $a > 0$ implies that x is bounded from below by $\frac{c}{a} - \frac{b}{a} y$. If $a < 0$, the latter provides an upper bound on x. These dependencies are called variable bound relations. They express the dependency of one bound of a variable on a bound of another variable. We will use the term *vbound* when referencing variable bound relations in the following.

Similar to the clique information, vbounds cannot only be deduced from variable bound constraints but can also be identified within more general constraints or during presolving, e.g., by probing. They are exploited by different solver components, e.g., for c-MIR cut separation, where they can be used to replace non-binary variables with binary ones [54]. In order to make vbounds available for those components, they can be stored in a global structure, the *variable bound graph*. In this directed graph, each node corresponds to the lower or upper bound of a variable and each vbound is represented by an arc pointing from the influencing bound to the dependent bound. This graph generalizes the mixed conflict graph introduced in [11] which is used to generate cutting planes for the mixed vertex packing problem. While the mixed conflict graph represents variable bounds between two binary variables or a binary and a continuous variable, the variable bound graph covers dependencies between all types of variables (possibly excluding those between binary variables which are already captured by the clique table).

For an example of a variable bound graph, see Fig. 1. We regard three constraints on variables x, y, and z, as shown in part (a). Each of these constraints provides two bounds on the involved variables as stated in part (b). Thereby, vbounds (1a) and (1b) are derived from constraint (1), (2a) and (2b) from (2), and (3a) and (3b) from (3).

$$x - 2y \leq 3 \quad (1)$$
$$x + 2z \leq 2 \quad (2)$$
$$x + 3y \leq 6 \quad (3)$$

(a) constraint set.

$$x \leq 2y + 3 \quad (1a)$$
$$y \geq \tfrac{1}{2}x - \tfrac{3}{2} \quad (1b)$$
$$x \leq 2 - 2z \quad (2a)$$
$$z \leq 1 - \tfrac{1}{2}x \quad (2b)$$
$$x \leq 6 - 3y \quad (3a)$$
$$y \leq 2 - \tfrac{1}{3}x \quad (3b)$$

(b) vbounds.

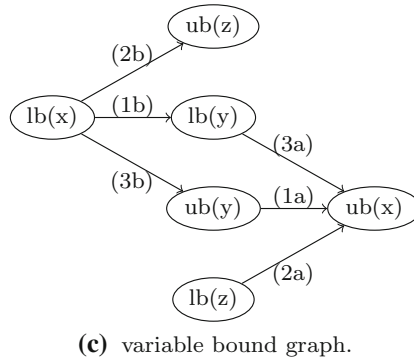**(c)** variable bound graph.

Fig. 1 Example of a variable bound graph

The resulting variable bound graph is illustrated in part (c). Each arc is labeled with the vbound it represents.

If a bound of a variable is tightened, implications can be read from this graph by following all paths starting at the corresponding node. Therefore, the graph can be used to compute an estimate of the impact that a bound change will have. This observation is the basis for the variable-bound-driven fix-and-propagate heuristic presented in Sect. 6.

3.3 Variable locks

In contrast to the two previous structures, variable locks are directly defined by the constraint matrix. They are a measure of how many constraints may block an increase or decrease of the value of a variable. In case of a MIP of form (1) with only $\leq$-constraints, the number ζ_i^+ of *up-locks* of a variable x_i is the number of constraints in which this variable has a positive coefficient a_{ri}, while the number ζ_i^- of *down-locks* counts the number of constraints with a negative coefficient of the variable. A more general definition for variable locks in constraint integer programming is given in [2]. In this paper, however, we focus on MIP and can thus use the simple definition above.

In the special case that a variable has no locks in one direction, its value in a solution can be moved into this direction without rendering a constraint infeasible. A simple rounding heuristic was introduced in [2] which is based on this argument. On the other hand, duality fixing [35] fixes variables to their bound if they have no locks in that direction and the objective coefficient has the right sign. The variable-locks-driven fix-and-propagate heuristic presented in Sect. 7 uses the variable locks to decide which variable is most influential and to which value it should be fixed to retain feasibility.

4 A framework for structure-driven fix-and-propagate heuristics

In this section, we present a new primal heuristic scheme for mixed integer programming which is based on global structures collected by MIP solvers. It forms the basis

for the three new primal heuristics discussed in the next sections which have been implemented in the academic MIP solver SCIP [2].

The general scheme is illustrated in Algorithm 1. In a first step, a subset of the integer variables is fixed based on the respective structure (lines 3–16). The `structure_fixing` method (line 5) is called in each iteration and determines the fixing based on the given global structure. It returns the index k of a variable that should be fixed to one of its two bounds and whether the variable is supposed to be fixed to the lower or upper bound. Additionally, the method gives back a return value `result`, which is either `continue`, `solve LP`, or `stop`. The default return value is `continue`, which just continues the fix-and-propagate process. On the other hand, `stop` and `solve LP` will both stop the fixing phase, see line 6, and go directly to the LP solving starting in line 17. The latter triggers the LP solve in any case, while with a `stop` return value, the usual checks that the problem was reduced sufficiently need to be passed, see below for more details. After the fixing is applied (line 7), domain propagation is performed (line 8). This uses a method `domain_propagation`, which performs domain propagation on the given MIP and returns the updated MIP as well as the information whether an infeasibility was detected during propagation. Propagation can work directly on the global structure, but also works on all other constraints of the model and can identify reductions that do not originate from the regarded global structure. By default, we limit the domain propagation call to perform only two rounds of propagation to avoid performance issues. Nevertheless, this is usually enough to detect trivial infeasibilities and apply implied bound changes on other variables—those contained in the global structure, but also other variables in the problem.

If domain propagation detects an infeasibility for the current assignment of variables, we backtrack one level, i.e., we undo the last fixing as well as the domain reductions deduced from it. Then, we remove the fixing value that led to the infeasibility from the domain of the respective variable and propagate this reduction (see lines 9–13). If the propagation detects infeasibility for this problem as well, one of the assignments we did before must have caused the infeasibility (or the global problem is already infeasible). We do not backtrack several levels in this case in order to avoid too much effort being spent before finding the invalid assignment but instead stop the heuristic immediately (line 14). If the updated problem is feasible, however, we continue with the fix-and-propagate procedure.

Note that the need to backtrack repeatedly indicates that the global structure used by the heuristics is missing essential components of the problem and directs the search to a wrong region. Therefore, Algorithm 1 is passed a limit κ on the number of backtracks performed. By default, we set $\kappa = 10$. If this number of backtracks is reached, the fix-and-propagate phase is stopped even if there are unfixed integer variables left in the structure (line 16). Otherwise, the fix-and-propagate phase is iterated until all variables in the global structure are fixed.

After the fixing phase, we check its success. For this, we compare the number of fixed integer variables to the total number of integer variables (line 17). Ideally, the heuristic fixed all integer variables, but it may happen that some of the variables are not contained in the global structure employed for the fixing process, or that the fixing phase stopped prematurely due to the backtrack limit or the `result` value being

---

Algorithm 1: Generic structure-driven fix-and-propagate heuristic

    input   : - MIP $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$
              - fixing thresholds α, β
              - backtrack limit κ
              - global structure $\mathcal{S}$
    output : - feasible solution or NULL, if no solution was found

1 begin

2   back $\leftarrow 0$, result $\leftarrow$ stop
        // 1. try to fix all integer variables in the structure

3   **while** $\{i \in \mathcal{I} \cap \mathcal{S} \mid \ell_i < u_i\} \neq \emptyset$ **do**

4 $(\tilde{\ell}, \tilde{u}) \leftarrow (\ell, u)$
            // get variable fixing based on global structure

5 $(k, \text{lower}, \text{result}) \leftarrow$ structure_fixing$(\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I}), \mathcal{S})$
            // fix variable

6   **if** $result \neq continue$ **then break**

7   **if** $lower$ **then** $\tilde{u}_k \leftarrow \ell_k$ **else** $\tilde{\ell}_k \leftarrow u_k$
            // perform domain propagation

8 $(\mathcal{P}(c, A, b, \tilde{\ell}, \tilde{u}, \mathcal{N}, \mathcal{I}), \text{inf}) \leftarrow$ domain_propagation$(\mathcal{P}(c, A, b, \tilde{\ell}, \tilde{u}, \mathcal{N}, \mathcal{I}))$
            // infeasibility detected: backtrack and exclude infeasible
               value

9   **if** inf **then**

10   back $\leftarrow$ back $+ 1$

11 $(\tilde{\ell}, \tilde{u}) \leftarrow (\ell, u)$

12   **if** $lower$ **then** $\tilde{\ell}_k \leftarrow \ell_k + 1$ **else** $\tilde{u}_k \leftarrow u_k - 1$
               // perform domain propagation

13 $(\mathcal{P}(c, A, b, \tilde{\ell}, \tilde{u}, \mathcal{N}, \mathcal{I}), \text{inf}) \leftarrow$
            domain_propagation$(\mathcal{P}(c, A, b, \tilde{\ell}, \tilde{u}, \mathcal{N}, \mathcal{I}))$

14   **if** inf **then return** $NULL$

15 $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I}) \leftarrow \mathcal{P}(c, A, b, \tilde{\ell}, \tilde{u}, \mathcal{N}, \mathcal{I})$

16   **if** $back \geq \kappa$ **then break**

        // 2. LP solving

17   **if** $|\{i \in \mathcal{I} \mid \ell_i = u_i\}| \geq \alpha|\mathcal{I}| \vee result = solve\ LP$ **then**

18 $(x^\star, \text{inf}) \leftarrow$ solve $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \emptyset)$

19   **if** inf **then return** $NULL$
            // try to round LP solution

20 $x^\star \leftarrow$ simple_round$(x^\star)$

21   **if** $x_i^\star \in \mathbb{Z}$ *for all* $i \in \mathcal{I}$ **then**

22   **return** $x^\star$

23   **else**
               // 3. LNS approach

24 $\mathcal{P}(\tilde{c}, \tilde{A}, \tilde{b}, \tilde{\ell}, \tilde{u}, \tilde{\mathcal{N}}, \tilde{\mathcal{I}}) \leftarrow$ presolve $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$

25   **if** $|\tilde{\mathcal{N}}| \leq \beta|\mathcal{N}|$ **then**

26 $x^\star \leftarrow$ solve $\mathcal{P}(\tilde{c}, \tilde{A}, \tilde{b}, \tilde{\ell}, \tilde{u}, \tilde{\mathcal{N}}, \tilde{\mathcal{I}})$ (with working limits)

27   **return** $x^\star$

28   **else**

29   **return** $NULL$

`stop`. The heuristic does not require all integer variables to be fixed at that point. It solves an LP on the remaining problem and, if feasible, tries to round the LP solution with the simple rounding heuristic mentioned in Sect. 3.3, see lines 18–22. If enough integer variables were fixed before, this LP is significantly smaller (and hopefully easier to solve) than the original LP relaxation. Additionally, not only the LP solving effort but also the success probability of the rounding heuristic depends on the success of the fixing step. Therefore, the heuristic scheme demands that a fixing rate of at least α after the fixing phase is reached (line 17) and stops otherwise. The only exception is the case that the `result` value was `solve LP`. Then, the heuristic continues without checking the fixing rate.

If rounding the LP solution was not successful, the heuristic employs an LNS approach to construct a feasible solution to the neighborhood defined by the remaining unfixed variables. However, since this is typically the most expensive step of the algorithm, we first apply fast presolving methods to the LNS sub-MIP defined by the fixings obtained in the previous phase, see line 24. These methods remove fixed variables and redundant constraints, apply bound tightening and duality fixing, and perform aggregations of variables, amongst others. While we checked before that we fixed a sufficient number of *integer* variables, we now require a reduction of β in the overall problem size in terms of *all* variables (line 25). Since the processing time of branch-and-bound nodes in the sub-MIP depends mainly on the LP solving time, a sufficiently decreased problem size is a good indicator for reasonable LNS times. Finally, the sub-MIP is solved, see line 26, and the best feasible solution found during sub-MIP solving is returned (line 27).

In order to limit the effort spent within the heuristics, we use working limits for the sub-MIP solving. First, the fixing thresholds α and β ensure that the problem is significantly easier after the fixing phase. Second, we aim at performing a quick partial solve of the sub-MIP. Therefore, we disable separation in the LNS sub-MIP solving, use only fast presolving algorithms, and disable all LNS heuristics to avoid recursion. Additionally, we disable strong branching and use the inference branching rule of SCIP [2]. If a primal feasible solution was found already, we set an objective limit such that the solution is improved by at least 1%. Finally, a node limit of 5000 is used together with a limit of 500 for the number of stalling nodes, i.e., consecutively processed nodes without finding a new best solution. These limits are chosen based on previous experiments with LNS heuristics in SCIP.

This general scheme is the same for all three heuristics proposed in this paper. The difference between them is how and in which order the variables are fixed in the first phase. This is defined by the global structures which represent interconnections between variables which can and will be propagated. The novel concept of the heuristics is that the order in which variables are fixed and the fixing values take into account the predicted impact a fixing will have on the domain propagation step. By this, domain propagation is not used as a supplementary subroutine to support the search, but as a driving mechanism to take decisions within the search: we choose fixings of which we know that they propagate well. How this is done for each of the three global structures is explained in the following sections.

5 The clique-driven fix-and-propagate heuristic

The idea behind the clique-driven fix-and-propagate heuristic is the following: Given a clique $\mathcal{C}$, at most one variable x_i, $i \in \mathcal{C}$ may be set to 1 in a feasible solution, all other variables need to be set to 0. Thus, by fixing one of the variables to 1, domain propagation will fix all other variables in the clique to 0, but will potentially also apply many more domain changes implied by any of the fixed variables. Consequently, it seems beneficial to choose large cliques in order to trigger many propagation changes. On the other hand, by choosing the cheapest variable, i.e., the variable with smallest objective coefficient as the one to fix to 1, we can aim at constructing solutions with small objective value.

Algorithm 2: `clique_fixing`

input : - MIP $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$
  - clique table $\mathcal{T}$

output : - index of binary variable x_k which should be fixed next
  - should x_k be fixed to 0?
  - result of the call: `continue`

1 **begin**
  // 1. select clique
2   $\mathcal{C}^\star \leftarrow \arg\max\{|\{x_i \in \mathcal{C} \mid \ell_i < u_i\}| \mid \mathcal{C} \in \mathcal{T}\}$
  // best index and corresponding smallest objective
3   $k^\star \leftarrow -1$
4   $c^\star \leftarrow \infty$
  // 2. find cheapest variable to fix
5   **for** $j \in \mathcal{C}^\star$ **do**
6   **if** $u_{c_j} = 1$ *and* $c_j < c^\star$ **then**
7       $k^\star \leftarrow j$
8       $c^\star \leftarrow c_j$

9   **return** $(k^\star, FALSE, continue)$

The fixing algorithm of the clique-driven fix-and-propagate heuristic is illustrated in Algorithm 2. In a first step, the next clique to process is selected (line 2). This is done with a greedy strategy: We select a clique with the largest number of unfixed variables. Note that the selection criterion implies that no variable contained in the clique is already fixed to 1 since propagation would have fixed all other variables to 0 otherwise. Additionally, we can assume that there always exists a clique with unfixed variables, since the algorithm is called in line 5 of Algorithm 1 directly after the while-loop starting in line 3 checks this condition. After choosing a clique, we select an unfixed variable in it with smallest objective coefficient and return that this variable should be fixed to 1, see lines 3–9. How successful this fixing strategy is and which fixing thresholds should be used, see Algorithm 1, is analyzed in the computational experiments presented in the remainder of this section.

5.1 Computational analysis

Our computational experiments are based on an implementation of the clique-driven fix-and-propagate heuristic within the academic MIP solver SCIP 4.0.0 [2,53] with SOPLEX 3.0.0 [53,64] as underlying LP solver. We use a modified version of the heuristic as described in this paper, which will replace the old version in the next release of SCIP. It is implemented as a primal heuristic plugin of SCIP and called once at the beginning of the root node processing. Note that finding new incumbent solutions is often most effective at the root node when a new primal bound might directly lead to global fixings, tighter cutting planes, and better initial branching decisions. This is the typical application of fix-and-propagate heuristics, since the diversification of the heuristic search for calls during the subsequent branch-and-bound phase is smaller than for other heuristics that rely on local LP optima.

In our first experiments, we ran SCIP with a node limit of 1, i.e., we let SCIP stop after the root node processing. This allows us to compare the heuristic runtime to the overall root node processing time. Additionally, we set a time limit of 3600 s and a memory limit of 16 GB. All results were obtained on a cluster of 3.2 GHz Intel Xeon X5672 CPUs with 12 MB cache and 128 GB main memory, running only one job per cluster node at a time. The experiments were performed on the MMMC test set which contains all instances from the last three MIPLIB benchmark sets [6,23,47] as well as the Cor@l test set [25]. We removed duplicates and the instances neos-1058477, neos-847051, and npmv07 because they caused numerical troubles. Note that the numerical troubles do not depend on the usage of the heuristic; they can also be observed if it is disabled. This left us with a total of 496 instances.

For our first experiment, we set the fixing thresholds α and β to 0 in order to always continue with LP and LNS sub-MIP solving. The backtracking limit of $\kappa = 10$ stayed unchanged. This experiment is meant to show the potential of the clique-driven fix-and-propagate heuristic and at the same time derive good default values for α and β.

Table 1 gives first aggregated results for this experiment. The rows list information for different subsets of the instances. All instances (row 1), those stopped before the LP was solved (row 2), and those where the fix-and-propagate phase was successful and the heuristic solved the subsequent LP, but no LNS sub-MIP (row 3). The last row shows statistics for the set of instances where the LNS sub-MIP was solved. For each subset, we list its size, i.e., the number of instances in this category, the number of

Table 1 Statistics for the clique-driven fix-and-propagate heuristic without fixing thresholds on the MMMC test set

Subset	Size	Sols	Root time	Heur time	F&P time	LP time	LNS time
All	496	189	83.93	2.28	0.11	0.08	2.09
Stopped early	226	–	91.49	0.12	0.12	–	–
Only LP solved	215	144	84.71	0.18	0.11	0.06	–
LP + LNS	55	45	49.76	19.39	0.04	0.50	18.85

instances for which the heuristic found a solution, and the average root node processing time (including presolving). Additionally, we show the average running time of the heuristic, as well as the average times for the fix-and-propagate phase, the LP solving, and the LNS sub-MIP solving. All averages are computed as arithmetic means.

Out of the 496 instances, 102 instances contain no cliques. Another 124 instances ran into a dead-end, where fixing the best variable in the current clique to 0 or 1 both led to infeasibility. These two cases together account for the row "Stopped early". On these instances, the clique-driven fix-and-propagate heuristic is fast and consumes only 0.13% of the root node running time.

For the remaining 270 instances, the heuristic solved the LP. The LP solution could be rounded to a feasible solution 144 times, and the LP was infeasible for 71 calls. If neither of the two happened, the LNS sub-MIP was solved. That was the case for 55 instances. If no sub-MIP was needed, the fix-and-propagate phase typically left few variables unfixed, which results in the LP solving time being about half of the fix-and-propagate time on average. The expensive case is the one that solves the LNS sub-MIP. Here, the LP is often harder to solve and needs about 1% of the root time, while the sub-MIP time dominates the heuristic time and accounts for 38% of the root node processing time. This is the main reason why the clique-driven fix-and-propagate heuristic without any fixing limits makes up for 2.7% of the total root time.

The fixing thresholds α and β are meant to reduce the running time of the heuristic by avoiding to spend too much time in LP and sub-MIP solving. To this end, we investigate the impact of the fixing rates on both effort and success of the heuristic in the LP and sub-MIP calls. Figure 2 illustrates the fixing rates for the 270 instances for which at least an LP was solved. Each bar of the histogram shows the number of instances for which the integer fixing rate after the fix-and-propagate step was within a certain range. More specifically, bar k represents all instances with fixing rate in $[5(k-1)\%, 5k\%)$, with an additional bar at 100% for the case that all integer variables were fixed. Each bar is further divided into two parts: the solid part illustrates the instances where the clique-driven fix-and-propagate heuristic was successful in constructing a feasible solution, while the checked part shows instances where no solution was found by the heuristic. Each of these parts is further divided into two segments. The dark one represents instances where the heuristic solved the LNS sub-MIP. For the instances illustrated by the light part, the LP solution could be rounded to a feasible solution (solid) or the LP was infeasible (checked). The effort spent within the clique-driven fix-and-propagate heuristic for different fixing rates is illustrated in Fig. 3. Again, each bar represents all instances with integer fixing rate within a certain 5% range, with one additional bar for instances where all integer variable were fixed. The height of the bars represents the average time spent in the heuristic compared to the total root (and preprocessing) time. The heuristic time counts into the root time so that this share is between 0 and 100

We observe in Fig. 2 that the success rate is high in particular at the two ends of the histogram. A total of 153 instances have a fixing rate of 90% or higher, the success rate for these instances is 85%. On the other hand, a solution is found for 75.8% of the 33 instances with a fixing rate of $< 20\%$. Figure 3 indicates that the low fixing rates, though successful in many cases, should be avoided. They require solving a sub-MIP of size similar to the original problem for most of the instances. This consumes 51.3%
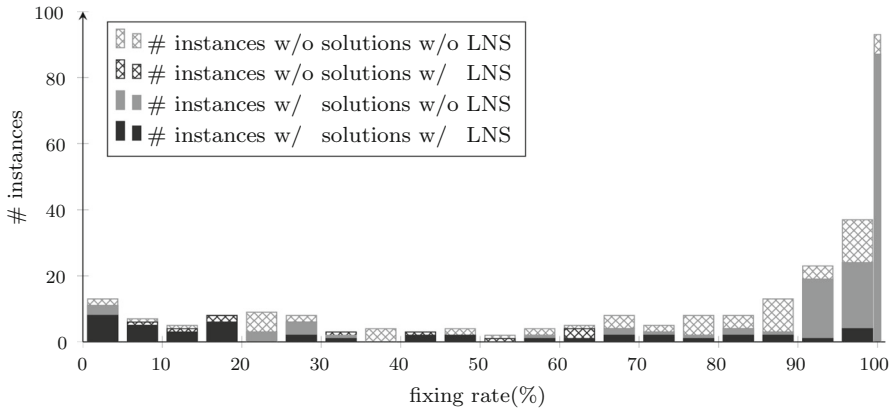
Fig. 2 Number of instances and solutions found by the clique-driven fix-and-propagate heuristic per integer fixing rate after the fix-and-propagate step. The solid part of each bar represents the instances for which the heuristic found a solution, the checked part the instances where it was unsuccessful. The darker (lighter) parts represent instances where a (no) sub-MIP was solved
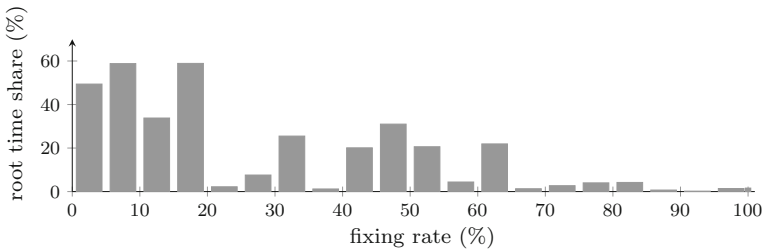

Fig. 3 Average effort of the clique-driven fix-and-propagate heuristic per fixing rate, relative to total root node processing time

of the aggregated root time on average— time that is better spent just continuing to solve the original problem. On the other hand, only 1.6% of the aggregated root node processing time is spent in the heuristic for instances with a fixing rate of 65% or higher. This threshold slightly increases the number of solutions constructed by the heuristic to 146 as compared to 130 for a limit of 90%. Many of the additional solutions are constructed via a sub-MIP; however, no unsuccessful sub-MIP calls can be observed. On the one hand, this can be seen as a success of the fixing scheme, on the other hand, the LP solving works very well here as a filter. It identifies all cases where the subproblem is infeasible so that the effort stays relatively low and accounts for only 1.6% of the root node processing time.

Figure 4 takes into account the problem size reduction of the presolved sub-MIP as well, which can be limited by parameter β in Algorithm 1. Since this is only relevant for instances which solve the LNS sub-MIP, the scatter plot illustrates only those 55 instances. Each instance is represented either by a circle if the heuristic found a solution, or by a triangle if it was not successful. The x-coordinate indicates the integer fixing rate after the fix-and-propagate phase, the y-coordinate the problem size reduction after presolve of the sub-MIP. Note that the latter includes all variables, also
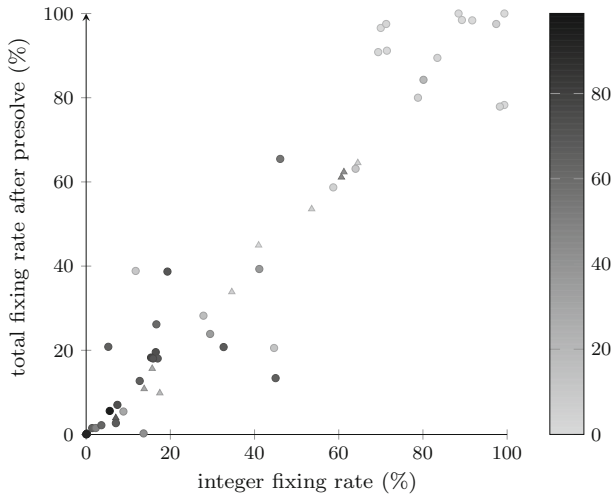
Fig. 4 Clique-driven fix-and-propagate heuristic time for LP and sub-MIP solving (shade) per fixing rate of integer variables (x-axis) and total fixing rate after sub-MIP presolving (y-axis). Each circle (triangle) represents one instance where the heuristic solved a sub-MIP and found a (no) solution

continuous ones, and can therefore be lower than the former which only considers integer variables. Finally, the shade shows the effort spent on solving the LP and sub-MIP, again relative to the total root node processing time.

Except for some outliers, the points are roughly located on the diagonal, meaning that a reduction in the number of integer variables often causes a similar reduction in the total problem size after presolve, as was to be expected. At the top right where both fixing rates are high, all sub-MIPs find a solution, and the joint time for LP and sub-MIP solving is below 5% of the root node processing time for each instance except for one where it amounts to about 20%. Overall choosing the limit β on the fixing rate after presolve similar to the limit α of integer variables fixed in the fixing phase seems reasonable.

This leads us to choose $\alpha = \beta = 65\%$ as default in the clique-driven fix-and-propagate heuristic for our following experiments as well as the version to be released. Table 2 lists the same information as Table 1, but for the heuristic with the updated limits. We see that the number of instances that were stopped before the LP solving increased to 301 since 75 instances did not reach the desired fixing rate. The heuristic finds 146 solutions, 43 less than in the previous experiment. However, 31 of those had been constructed with the aid of an expensive sub-MIP. Overall, the number of sub-MIP calls is reduced from 55 to 14 with a 100% success rate now whenever the heuristic solves a sub-MIP. Since also the neighborhoods to investigate are smaller due to the fixing limit, the average LNS time is significantly reduced from more than 19 s to about half a second. This leads to a significantly smaller runtime of the heuristic compared to the previous experiment. On average over all instances, the heuristic needs about 0.2% of the root node processing time, which is a good value given a success rate of 29.4%.

Table 2 Statistics for the clique-driven fix-and-propagate heuristic with final fixing thresholds $\alpha = \beta = 65\%$ on the MMMC test set

Subset	Size	Sols	Root time	Heur time	F&P time	LP time	LNS time
All	496	146	81.80	0.13	0.11	0.01	0.01
Stopped early	301	–	76.93	0.09	0.09	–	–
Only LP solved	181	132	89.86	0.16	0.13	0.04	–
LP + LNS	14	14	82.21	0.55	0.06	0.03	0.46

6 The variable-bound-driven fix-and-propagate heuristic

In the variable-bound-driven fix-and-propagate heuristic, we implemented different rules for determining the variable fixings. All of them make use of a topological sorting of the variable bound graph. Recall that a topological sorting of an acyclic directed graph is an order of the nodes, such that for every arc (i, j), node i precedes node j in the order. Since the variable bound graph can contain cycles, we may need to break them by randomly removing one of the arcs in the cycle. We will use a topological sorting of this reduced graph to define the order in which variables are fixed. Note that cycles in the variable bound graph are uncommon and may already be removed during presolving. In SCIP, this is done by cycle detection in the variable bound graph [53] and the clique table analysis [40].

Note that each clique $\mathcal{C}$ represents a set of vbounds as well: for each pair of variables $x_i, x_j \in \mathcal{C}$, the two vbounds $x_i \leq 1 - x_j$ and $x_j \leq 1 - x_i$ are implied, which correspond to the arcs $(\mathrm{lb}(x_j), \mathrm{ub}(x_i))$ and $(\mathrm{lb}(x_i), \mathrm{ub}(x_j))$ in the variable bound graph, respectively. We take those implied vbounds into account when computing the topological sorting. Special care has to be taken to avoid unnecessarily high runtimes for the sorting process. This is done with a simple depth-first search, which has linear effort in the number of nodes and arcs. However, the number of edges represented by a single clique is quadratic in the number of variables contained in the clique. Therefore, we treat them implicitly and use an improvement suggested in a similar context by [8] so that we need to enter every clique at most twice. Thus, the variable-bound-driven fix-and-propagate heuristic also uses information stored in the clique table just like the clique-driven heuristic, but it only uses it to refine the topological sorting of the variable bound graph. The final fixing scheme summarized in Algorithm 3 is again a sub-algorithm of Algorithm 1 and called in each iteration of the fix-and-propagate phase to determine a variable to fix and the corresponding fixing value.

The fixing method starts with an initialization step (lines 2–4). The method *topological_sort* called for this purpose returns an array of nodes in topological order (with respect to the reduced graph). The method already sorts out disconnected nodes, nodes corresponding to continuous variables and nodes from which only nodes corresponding to continuous variables can be reached. Additionally, the set $\mathcal{U}$ of unprocessed nodes is initialized to all nodes in the order.

Then, the first unprocessed node in the topological order is selected. If it corresponds to an already fixed variable, e.g., by a previous iteration of the fixing phase, it is ignored and the next variable is selected (lines 9–11). Recall that each node v of the variable

---

Algorithm 3: `variable_bound_fixing`

---

input   : - MIP $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$
            - variable bound graph including clique table $\mathcal{G} = (\mathcal{V}, \mathcal{A})$
            - `strategy` $\in \{\texttt{aggr}, \texttt{cons}\}$ – force change or feasibility?
            - `obj` $\in \{\texttt{noobj}, \texttt{qual}, \texttt{feas}\}$ – how should the objective function be taken into account?

output : - index of binary variable x_k which should be fixed next
            - should x_k be fixed to its lower bound (otherwise: upper bound)?
            - result of the call: `continue` or `stop` fixing

---

1 **begin**
　　　// initialization
2　　**if** *1st call* **then**
　　　　// topological sorting; the order does not contain
　　　　　independent nodes, nodes which correspond to or only
　　　　　influence continuous variables
3　　　　$\pi \leftarrow \texttt{topological\_sort}(\mathcal{G})$;
4　　　　$\mathcal{U} \leftarrow \mathcal{V} \cap \pi$;
5　　$k \leftarrow 0$;
6　　**while** $k = 0 \wedge |\mathcal{U}| > 0$ **do**
　　　　// select first unprocessed node
7　　　　$k \leftarrow \min\{1 \le i \le |\pi| \mid \pi_i \in \mathcal{U}\}$;
8　　　　$\mathcal{U} \leftarrow \mathcal{U} \setminus \{\pi_k\}$;
　　　　// discard fixed variables
9　　　　**if** $\ell_{idx(\pi_k)} = u_{idx(\pi_k)}$ **then**
10　　　　　$k \leftarrow 0$;
11　　　　　**continue** ;

　　　　// fixing value
12　　　　**if** *strategy* $= \texttt{aggr}$ **then**
13　　　　　fixtolower $\leftarrow$ not lower(π_k);
14　　　　**else**
15　　　　　fixtolower $\leftarrow$ lower(π_k);

　　　　// consider objective function
16　　　　**if** *obj* $= \texttt{qual}$ **then**
　　　　　// do not fix to worse bound w.r.t. objective
17　　　　　**if** *fixtolower* $\neq (c_{idx(\pi_k)} \ge 0)$ **then**
18　　　　　　$k \leftarrow 0$;
19　　　　**else if** *obj* $= \texttt{feas}$ **then**
　　　　　// do not fix to better bound w.r.t. objective
20　　　　　**if** *fixtolower* $\neq (c_{idx(\pi_k)} \le 0)$ **then**
21　　　　　　$k \leftarrow 0$;

22　　**if** $k > 0$ **then**
23　　　　**return** $(idx(\pi_k), fixtolower, \texttt{continue})$;
24　　**else**
25　　　　**return** $(0, FALSE, \texttt{stop})$;

---

bound graph represents a bound of a variable. Tightening this bound causes some bound changes on other variables, as defined by all paths in the variable bound graph starting at node v. Consequently, the earlier a node is considered within the topological order, the more impact on other bounds we expect when tightening the corresponding bound.

The first variant by which the heuristic determines fixings aims at obtaining a large neighborhood by fixing variables such that only few additional restrictions are caused. This results in a neighborhood with a higher probability both for containing feasible solutions as well as high-quality solutions. To this end, this variant fixes the variable to the bound represented by the current node. This means that not the bound corresponding to the current node is tightened, but the opposite bound, which comes later in the topological order (if even) and thus causes fewer reductions on other bounds. The second variant uses an opposing argument: A large neighborhood is more expensive to process and finding any solutions in there might need more effort than in a smaller neighborhood with more fixed variables. Therefore, we fix the variable to the reverse bound, i.e., tighten the bound corresponding to the node in the variable bound graph. This forces a change of many other bounds of variables, a concept known to be rather effective in order to drive the solution to feasibility faster, cf. [57]. The parameter `strategy` is used to switch between the two variants in Algorithm 3, lines 12–15. Thereby, a value of `aggr` corresponds to the more aggressive second variant, while `cons` represents the more conservative first variant.

We obtain two variations for each of the previously mentioned variants by taking into account the objective function. For this, we need the notion of the best bound of a variable, which is the bound that leads to the best objective contribution of the variable, i.e., its lower bound if its objective value is non-negative, and its upper bound otherwise. The first variation applies fixings only if the variable is fixed to its best bound. Conversely, the second variation fixes a variable only if it is not fixed to its best bound. The motivation for the first variation is clearly to aim at obtaining high-quality solutions, it is enabled by setting `obj` to `qual` in Algorithm 3, lines 16–21, instead of the default value `noobj`. Variation 2 is based on the observation that typically, the constraints of the problem push variables away from their best bound, while fixing them to their worst bound might give a higher chance for a feasible solution in the end. It is triggered by setting `obj` to `feas`. Both variations may keep variables in the variable bound graph unfixed. If this is the case, the fixing algorithm returns `stop` as the `result` to signal that no further fixings will be performed by the current scheme (line 25).

Overall, this gives us six fixing schemes for the variable-bound-driven fix-and-propagate heuristic. We will investigate in the following how well they complement each other or if any of them is dominated by the others.

6.1 Computational analysis

Our computational experiments are performed in the same environment, with the same software, and on the same instance set as described in Sect. 5.1. Again, the updated

version of the variable-bound-driven fix-and-propagate heuristic is called once at the start of root node processing as motivated in Sect. 5.1.

Similar to the clique-driven fix-and-propagate heuristic, we performed computational experiments to define reasonable fixing thresholds. The success rate is similar to the previous experiment; however, the effort spent within a single call of a variant is higher. Note that this is caused mainly by a higher number of instances that lack enough relevant structure and we performed the first experiment without applying any fixing thresholds, as we did for the clique-driven heuristic. Again, fixing thresholds $\alpha = \beta = 65\%$ prove to be reasonable and will be used in the following. Given these thresholds, we only call the heuristic on instances where we expect enough fixings based on the size of the variable bound graph. For more details, we refer to the technical report [37]. The final fixing thresholds result in 1099 calls of a variant of the heuristic with a success rate of 67.6% and an average running time of 1.6% of the root node processing time. Compared to the clique-based heuristic, which has a higher success rate of 74.9 %, the average effort per call of a variant is almost identical. The different variants, however, allow to spend more effort to potentially find more or better solutions.

How well the variants complement each other is investigated in Table 3. For each of the six variants (defined by the first two columns which present the values of the two parameters `strategy` and `obj`), the table shows its success in finding primal solutions. We present four different statistics. Column "Sols" lists the total number of solutions found by the variant, while column "Best sols" displays the number of instances where the variant found a solution with best objective value among the variants. This includes cases where multiple variants found solutions with the same best objective value, while column "Single best sols" only includes instances where the solution was strictly better than the solutions found by all other variants. Finally, column "Single sols" summarizes the number of instances for which the respective variant was the only one able to construct a feasible primal solution.

The three variants (`cons`, `feas`), (`aggr`, `noobj`), and (`aggr`, `feas`), which find the highest number of solutions also find many best solutions. Variant (`aggr`, `qual`) is ranked worst with respect to the number of solutions found, but taking into account the objective during fixing proves beneficial for finding better solutions than the other variants: it is ranked second for the number of strictly best solutions. Overall, (`cons`,

Table 3 Comparison of the six variants of the variable-bound-driven fix-and-propagate heuristic with fixing thresholds $\alpha = \beta = 65\%$ on the MMMC test set (496 instances)

strategy	obj	Sols	Best sols	Single best sols	Single sols
cons	noobj	113	46	2	1
cons	qual	112	54	25	3
cons	feas	139	78	35	7
aggr	noobj	147	84	17	2
aggr	qual	90	59	26	6
aggr	feas	142	72	24	12

Table 4 Statistics for the variable-bound-driven fix-and-propagate heuristic with fixing thresholds $\alpha = \beta = 65\%$ on the MMMC test set

Subset	Size	Sols	Root time	Heur time	F&P time	LP time	LNS time
All	496	172	79.26	1.74	1.06	0.11	0.57
Stopped early	278	–	75.99	0.04	0.04	–	–
Only LP solved	165	122	81.70	3.19	2.97	0.21	–
LP + LNS	53	50	102.42	6.17	0.45	0.39	5.33

`noobj`) performs worst. Although finding the fourth-highest number of solutions, it is ranked last for all other criteria. For all but two instances, there is at least one other variant which finds a solution of equal or better quality. For all other variants, there are at least 17 instances where the variant finds the single best solution and often several instances where no other variant finds a solution at all. These results, together with the relatively small effort for the call of a single variant motivate the default settings for the heuristic. Variant (`cons`, `noobj`) is disabled by default, all five other variants are called sequentially in each call of the variable-bound-driven fix-and-propagate heuristic.

The performance of the final version of the variable-bound-driven fix-and-propagate heuristic is presented in Table 4. It lists the same kind of information as Table 1 for each call of the variable-bound-driven heuristic, which includes the sub-calls of the five variants enabled by default. Additionally, the heuristic is not run on instances for which the variable bound graph spans only such a small fraction of the variables that reaching the fixing rate is very unlikely. More precisely, the heuristic is only called if the number of variable bounds is at least 0.1α of the total variable number. As a consequence, the heuristic reaches the LP solving and rounding step for only 44% of the instances. Note that an instance is counted for having solved an LNS sub-MIP if at least one of the variants did so on this instance; otherwise, an instance is counted as having solved the LP if one of the variants did so. The former case happens for 53 instances, for 50 of those, a feasible primal solution can be constructed. Overall, the variable-bound-driven fix-and-propagate heuristic consumes about 2.2% of the root node processing time. The effort is negligible for instances where it is stopped before the LP solving, while for the remaining instances, it increases to 4.51%. This seems still reasonable given the success rate of 78.9% on this set of instances.

7 The variable-locks-driven fix-and-propagate heuristic

The variable-locks-driven fix-and-propagate heuristic works on a structure which is present in each MIP: the constraint matrix, and in particular, the variable locks, see Sect. 3.3. While cliques and the variable bound graph represent only a part of the original constraints, but also cover relations implicitly given in the problem and detected during presolving, the variable locks take into account all given constraints.

The heuristic is motivated by greedy heuristics for set covering problems: Starting with an all-zero solution, one selects one variable which is contained in the highest

number of constraints and fixes it to 1. By this, all these set covering constraints are fulfilled independently of the other variables' values. In subsequent steps, a variable is selected which is contained in the highest number of not-yet fulfilled constraints.

How the variable-locks-driven fix-and-propagate heuristic translates this approach to general MIP is shown in Algorithm 4. In each iteration of the fixing process, a "high-impact" binary variable is selected, where the impact of a variable is decided based on the sum of its up- and down-locks, which corresponds to the number of constraints it is part of, cf. line 5. Then, the given variable is fixed to the bound where it has the smaller number of locks, see lines 8 to 12. This aims at reaching feasibility fast and possibly ensuring that some constraints are already fulfilled after a few fixings, no matter how the values of the remaining variables in the constraint will be chosen within their updated bounds. If a variable has the same number of up- and down locks, we use a randomized approach to determine its fixing value. The variable is then fixed to 1 with a probability of 67%, see line 11. We chose this probability based on a previous experiment where it showed a good performance.

If a constraint is already fulfilled, its locks are disregarded (see lines 2–4), so that the impact and the fixing direction are always determined with respect to the not-yet fulfilled constraints only. Therefore, it may happen that all constraints are fulfilled already and none of the remaining variables has any locks left. In this case, we stop the fixing procedure and return that the LP should be solved directly in order to determine optimal values for the remaining variables, cf. lines 6–7.

Algorithm 4: `locks_fixing`

input : - MIP $\mathcal{P}(c, A, b, \ell, u, \mathcal{N}, \mathcal{I})$

output : - index of binary variable x_k which should be fixed next;
  or -1 if no further fixings should be performed
  - should x_k be fixed to it 0 (otherwise: 1)?
  - result of the call: `continue` or directly `solve LP`

1 **begin**
   // get variable locks
2   **for** $i \in \mathcal{B}$ **do**
3      $\zeta_i^+ \leftarrow |\{r \in [1, \ldots, m] : a_{ri} > 0 \wedge \max\{A_{r.}x | \ell \leq x \leq u\} > b_r\}|$
4      $\zeta_i^- \leftarrow |\{r \in [1, \ldots, m] : a_{ri} < 0 \wedge \max\{A_{r.}x | \ell \leq x \leq u\} > b_r\}|$

    // select variable with highest sum of up- and down-locks
5    $k \leftarrow \arg\max_{i \in \mathcal{B}}\{\zeta_i^+ + \zeta_i^-\}$

    // no variable with locks left, stop fixing phase
6   **if** $\zeta_k^+ = \zeta_k^- = 0$ **then**
7   **return** *(0, FALSE, solve LP)*

    // fix variable to bound where it has fewer locks
8   **if** $\zeta_k^+ > \zeta_k^-$ **then**
9   **return** *(k, TRUE, continue)*
10   **else if** $\zeta_k^+ = \zeta_k^-$ **then**
11   **return** *(k, TRUE, continue)* with probability 33 %
12   **return** *(k, FALSE, continue)*

7.1 Computational analysis

For the computational analysis of the variable-locks-driven fix-and-propagate heuristic, we used the same environment, the same software, and the same instance set as described in Sect. 5.1. As the previously presented heuristics, the heuristic is called once at the start of root node processing. Note that we are not using the exact activities to identify already fulfilled constraints. Computing them from scratch in each iteration would be too expensive; instead, we are using an updating mechanism which updates activities whenever the heuristic fixes a variable. The structure of SCIP, however, makes it hard to update them for propagated domain changes. Therefore, constraints may be detected to be fulfilled later than they could otherwise be, but the detection still works reasonably well while being sufficiently fast.

Our experiments concerning the fixing thresholds α and β showed that using a structure which covers the whole problem typically results in fixing a high number of variables or failing repeatedly while trying to do so. On 232 of the 286 instances for which at least an LP was solved, a fixing rate of 100% is obtained or all constraints could be satisfied. Only 18 instances have a fixing rate of $< 90\%$. For a detailed analysis, we refer to the technical report [37]. As default value for the fixing thresholds α and β, we choose 65%, also for consistency with the other heuristics.

The final statistics for the variable-locks-driven fix-and-propagate heuristic are summarized in Table 5, which again lists the same kind of information as Table 1. The heuristic finds 240 solutions, which results in a success rate of 48.4% over the complete test set and 88.2% for the subset of instances where the heuristic is not stopped early. It only solves a single LNS sub-MIP which is a direct consequence of the effectiveness of domain propagation and obtaining high fixing rates, making it more probable to be successful in rounding the LP solution. Overall, the heuristic is reasonably fast, accounting for about 0.5% of the average root node processing time.

8 Computational results

In the previous sections, we did a local analysis of the newly proposed heuristics. This means that we evaluated each heuristic individually, focussing on its success in terms of solutions being found. Additionally, we investigated their runtimes and which part of the heuristic algorithm consumed how much time. In this section, we investigate

Table 5 Statistics for the variable-locks-driven fix-and-propagate heuristic with final fixing thresholds $\alpha = \beta = 65\%$ on the MMMC test set

Subset	Calls	Sols	Root time	Heur time	F&P time	LP time	LNS time
All	496	240	78.15	0.42	0.22	0.20	0.00
Stopped early	224	–	81.04	0.25	0.25	–	–
Only LP solved	271	239	75.95	0.55	0.19	0.37	–
LP + LNS	1	1	24.70	1.88	1.15	0.05	0.69

Table 6 Pairwise comparison of the structure-driven fix-and-propagate heuristics, showing the number of instances where one heuristic found a better solution than the other

		worse solution					worse solution		
		clique	vbound	locks			clique	vbound	locks
better	clique	–	68	94	better	clique	–	43	71
	vbound	117	–	127		vbound	67	–	86
	locks	155	130	–		locks	38	20	–

(a) instances where at least one heuristic found a solution

(b) instances where both heuristics found a solution

the impact of the structure-driven fix-and-propagate heuristics on the overall solving behavior of SCIP and evaluate how well they can be combined. For this, we use SCIP version 4.0 with default settings except for the settings enabling the three investigated heuristics.

Before we come to the overall performance, let us shortly look at how well the heuristics complement each other. The variable-locks-driven fix-and-propagate heuristic finds a solution for 48.4% of the instances, while the clique- and the variable-bound-driven heuristic, which both depend on a more specific structure in the problem, are only able to generate solutions for 29.4% and 34.5% of the instances, respectively. Table 6 shows a pairwise comparison of the heuristics with respect to the objective value of solutions being constructed. On the left side, all instances are regarded where at least one of the two heuristics found a solution, not counting instances where both found a solution of equal objective value. Here, the variable-locks-driven heuristic performs best, followed by the variable-bound-driven one. On the right side the evaluation is restricted to instances where both heuristics found a solution; now the variable-bound-driven heuristic performs best with the variable-locks-driven heuristic being ranked last. Summing up, the variable-locks-driven heuristic performs very well with respect to the number of solutions found but does not take into account the objective function. The clique- and variable-bound-driven heuristics both consider the objective function and construct solutions with better objective values but are successful for fewer instances. Among the two, the latter is more successful, but also considerably more expensive due to running up to five fixing scheme variants. Based on this analysis, we decided to configure the heuristics as follows: the clique-driven heuristic is called first, then the variable-locks-driven one and finally the variable-bound-driven heuristic. We are running the faster heuristics first because a solution generated by them provides a cutoff bound that may speed-up the LP and sub-MIP solving of the variable-bound-driven heuristic.

For the overall performance evaluation, we ran SCIP once without any of the three structure-driven primal heuristics, once for each of the three heuristics with only this heuristic enabled and the other two disabled, and once with all three heuristics enabled. We used a cluster of 2.50 GHz Intel Xeon E5-2670 v2 CPUs with 128 GB main memory. Each job was run exclusively on one node with a time limit of 7200 s and a memory limit of 100 GB. In order to reduce the impact of performance variability (see [47,52]), we ran each instance six times with different random seeds (one of them being the default seed). We removed 27 instance/seed combinations that caused numerical

Table 7 Solution process statistics for SCIP with default settings and with additional structure-driven heuristics

Setting	Opt	Sols	First sol (s)	Primal int.	Time (s)
All instances (2967)					
nostructheur	1863	0	4.2	2080.3	431.7
onlyclique	1875	869	4.2	2025.0	429.0
onlyvbound	1877	1023	3.8	1979.1	428.9
onlylocks	1874	1443	4.0	2035.6	430.0
allstructheur	1880	1730	3.5	1857.2	421.2
Hard instances (874)					
nostructheur	791	0	7.3	3011.3	686.9
onlyclique	803	187	7.1	2994.7	667.6
onlyvbound	805	231	6.4	2908.0	661.0
onlylocks	802	405	6.6	2755.4	671.0
allstructheur	808	479	5.7	2596.3	630.2

troubles in one of the experiments, leaving us with 2967 instance/seed combinations. In a slight abuse of notation, we will refer to each instance/seed combination as an individual instance in the following. The results of the experiments were evaluated with the *Interactive Performance Evaluation Tools* (IPET) [44] and are summarized in Table 7.

The table is divided into two parts. Each part lists solving process statistics for the five settings, each represented by one row, on a different (sub-)set of instances. The first column denotes the setting, followed by columns listing the number of instances solved to optimality within the time limit of two hours and the number of solutions found by the structure-driven fix-and-propagate heuristics. Column four lists the time in seconds until the first solution was found for the instances, averaged by the shifted geometric mean [2] with a shift of 1 s. Column "Primal int." shows the shifted geometric mean with a shift of 100 of the primal integrals[1] for the instances. The last column presents the shifted geometric mean of the running time in seconds with a shift of 10 s.

The first part of the table considers all 2967 instances. On this set, each of the structure-driven fix-and-propagate heuristics helps to increase the number of instances solved to optimality within the time limit, while enabling all of them leads to the best results with 17 more instances being solved than without any of the heuristics. When running all three heuristics, an initial solution is constructed for 58.3% of the instances. Each of the heuristics individually reduces the time needed to find a first feasible solution by 1.7% (clique-driven heuristic) to 10.3% (variable-bound-driven heuristic). Used together, they even accomplish a reduction of 16.4%. With respect to the primal integral, each heuristic alone accounts for a reduction by 2.2% to 4.9%, by enabling all three heuristics, a reduction of 10.7% is obtained. But not only these measures tailored to primal heuristics are improved: the most important measure, the

---

[1] We compute the primal integral [15] of instance i as $P(i) = \int_{t=0}^{t_{max}} \gamma_i(t)\, dt$ with $t_{max} = 7200$ seconds and $\gamma_i(t)$ the primal gap at time t.

solving time to optimality, is also slightly improved by running the heuristics. Even though about one-third of the instances in the test set do not change their running time because they time out with all settings, we observe a speed-up of 2.4% with all three heuristics enabled. More interesting in this regard is the second part of Table 7 which refers to the subset of "hard" instances. This excludes instances that all settings solved within 100 s. For these instances, the potential for improvement is smaller while the overhead for running the heuristic has a larger impact. The hard instances, on the other hand, are the ones where algorithmic improvements are particularly needed and where we expected a larger impact of the heuristics. We additionally exclude all instances that none of the settings could solve since those dampen the actual improvement. The number of additionally solved instances stays unchanged compared to the complete set, as is implied by the definition of the subset. The success rates of clique- and variable-bound-driven heuristics are both reduced by about 8% as compared to the complete test set. The variable-locks-driven fix-and-propagate heuristic is almost as successful as before, while running all heuristics still finds a solution for 54.8% of the hard instances. When running all heuristics, the time to the first solution is reduced by 21.2% and the average primal integral by 13.8%. Due to focusing on the most interesting instances and omitting unsolvable and very easy ones, we can observe a solving time reduction by 8.3% now when activating all heuristics.

These are impressive numbers for primal heuristics in mixed-integer programming. The improvement is not caused by the solutions found by the heuristics alone, however. There is also a side-effect which impacts performance: the generation of conflict constraints [1,63]. They capture the essence of infeasible assignments detected during the fix-and-propagate phase and help to guide the subsequent search. To assess this effect, we further split the hard instances into the 479 instances where at least one of the heuristics found a solution on the one side and the remaining 395 on the other side. For the first set, the positive effects of running all three structure-driven heuristics are strengthened. The time to the first solution is reduced by 45.9% and the average primal integral by 25.8%. The average solving time goes down by 12.3% while 11 more instances are solved. On the 395 instances where no solution is constructed, the time to the first solution is increased by 0.5% and the average primal integral by 2.3%. This is due to the overhead caused by the heuristics which slows down the initial root processing of the main MIP solve. In the long run, however, conflict constraints generated by the seemingly unsuccessful heuristic calls improve the solving time by 3.1% and even help to solve 6 more instances. An additional computational experiment in which we disabled the creation of conflict constraints showed a reduction in the solving time improvement by about one third, while the impact on the average time to a first solution and average primal integral is considerably smaller (about 5% and 20%, respectively). This shows that the main task of our newly proposed primal heuristics, namely generating primal feasible solutions, is still the main reason for the improvements we observed.

9 Conclusions and outlook

In this paper, we presented three primal heuristics which are based on global structures available within MIP solvers. Those structures are the clique table, the variable bound graph, and the variable locks based on the constraint matrix. The heuristics use these structures to define a sequence of variable fixings applied in a fix-and-propagate approach. The LP relaxation of the resulting sub-problem is then solved and rounded. If the rounded LP solution is not feasible, the sub-problem is solved in an LNS fashion. In our approach, domain propagation is not only used as a tool to avoid infeasible fixings but rather are the fixing order and fixing values decided based on their effect on the domain propagation step. The global structures provide the tools to predict this effect by representing a part of the domain reductions that can be deduced from a variable fixing.

We performed a detailed analysis of the three heuristics to derive appropriate default settings. Our final computational experiments indicate that all three heuristics complement each other in the academic MIP solver SCIP. When applying all of them at the beginning of the branch-and-bound search, they are able to generate a solution for almost 60% of the instances in standard MIP benchmark sets. This reduces the shifted geometric means of both the time to the first solution as well as the primal integral significantly. The structure-driven fix-and-propagate heuristics prove to perform particularly well on hard instances, where they decrease the solving time by more than 12% when successful. But even when they do not find a feasible solution, they are still able to provide a small improvement by 3% on average due to other effects, most importantly, conflict constraints generated for unsuccessful calls. Therefore, the updated versions of all three heuristics are part of SCIP since release 5.0 and enabled by default.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optim. **4**(1), 4–20 (2007)
2. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
3. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optim. **4**(1), 77–86 (2007)
4. Achterberg, T., Berthold, T., Hendel, G.: Rounding and propagation heuristics for mixed integer programming. In: Klatte, D., Lüthi, H.-J., Schmedders, K. (eds.) Operations Research Proceedings 2011, pp. 71–76. Springer, Berlin (2012)
5. Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. Technical Report 16-44, ZIB, Takustr. 7, 14195 Berlin (2016)
6. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. **34**(4), 1–12 (2006)
7. Achterberg, T., Raack, C.: The MCF-separator: detecting and exploiting multi-commodity flow structures in MIPs. Math. Program. Comput. **2**(2), 125–165 (2010)
8. Achterberg, T., Wunderling, R.: Mixed integer programming: analyzing 12 years of progress. In: Facets of Combinatorial Optimization, pp. 449–481. Springer (2013)
9. Andrade, C.E., Ahmed, S., Nemhauser, G.L., Shao, Y.: A hybrid primal heuristic for finding feasible solutions to mixed integer programs. Eur. J. Oper. Res. **263**(1), 62–71 (2017)

10. Atamtürk, A., Nemhauser, G.L., Savelsbergh, M.W.: Conflict graphs in solving integer programming problems. Eur. J. Oper. Res. **121**(1), 40–55 (2000)
11. Atamtürk, A., Nemhauser, G.L., Savelsbergh, M.W.: The mixed vertex packing problem. Math. Program. **89**(1), 35–53 (2000)
12. Bergman, D., Cire, A.A., van Hoeve, W.-J., Yunes, T.: BDD-based heuristics for binary optimization. J. Heuristics **20**(2), 211–234 (2014)
13. Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optim. **4**(1), 63–76 (2007)
14. Berthold, T.: Primal heuristics for mixed integer programs. Diploma thesis, Technische Universität Berlin (2006)
15. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013)
16. Berthold, T.: Heuristic algorithms in global MINLP solvers. Ph.D. thesis, Technische Universität Berlin (2014)
17. Berthold, T.: RENS—the optimal rounding. Math. Program. Comput. **6**(1), 33–54 (2014)
18. Berthold, T.: Improving the performance of MIP and MINLP solvers by integrated heuristics. In: Dörner, K.F., Ljubic, I., Pflug, G., Tragler, G. (eds.) Operations Research Proceedings 2015, pp. 19–24. Springer, Cham (2017)
19. Berthold, T., Feydy, T., Stuckey, P.J.: Rapid learning for binary programs. In: Lodi, A., Milano, M., Toth, P. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Lecture Notes in Computer Science, pp. 51–55. Springer, Berlin (2010)
20. Berthold, T., Gleixner, A.M.: Undercover—a primal heuristic for MINLP based on sub-MIPs generated by set covering. In: Bonami, P., Liberti, L., Miller, A.J., Sartenaer, A. (eds.), Proceedings of the EWMINLP, pp. 103–112 (2010)
21. Berthold, T., Perregaard, M., Mészáros, C.: Four good reasons to use an interior point solver within a MIP solver. Technical Report 17-42, ZIB, Takustr. 7, 14195 Berlin (2017)
22. Bixby, R.E.: A brief history of linear and mixed-integer programming computation. Documenta Math., pp. 107–121 (2012)
23. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima **58**, 12–15 (1998)
24. Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., Wunderling, R.: MIP: Theory and practice—closing the gap. In: Powell, M.J.D., Scholtes, S. (eds.) Systems Modelling and Optimization: Methods, Theory, and Applications, pp. 19–49. Kluwer Academic Publisher, Dordrecht (2000)
25. COR@L. MIP Instances (2014). http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/. Accessed 02 July 2018
26. Dakin, R.J.: A tree-search algorithm for mixed integer programming problems. Comput. J. **8**(3), 250–255 (1965)
27. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Program. **102**(1), 71–90 (2004)
28. Dey, S., Iroume, A., Molinaro, M., Salvagnin, D.: Improving the randomization step in feasibility pump. arXiv preprint arXiv:1609.08121 (2016)
29. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. **104**(1), 91–104 (2005)
30. Fischetti, M., Lodi, A.: Local branching. Math. Program. **98**(1–3), 23–47 (2003)
31. Fischetti, M., Lodi, A.: Repairing MIP infeasibility through local branching. Comput. Oper. Res. **35**(5), 1436–1445 (2008). Special Issue: Algorithms and Computational Methods in Feasibility and Infeasibility
32. Fischetti, M., Lodi, A.: Heuristics in mixed integer programming. In: Cochran, J.J., Cox, L.A., Keskinocak, P., Kharoufeh, J.P., Smith, J.C. (eds.) Wiley Encyclopedia of Operations Research and Management Science. Wiley, New York (2010). Online publication
33. Fischetti, M., Monaci, M.: Proximity search for 0–1 mixed-integer convex programming. J. Heuristics **20**(6), 709–731 (2014)
34. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Math. Program. Comput. **1**, 201–222 (2009)
35. Fügenschuh, A., Martin, A.: Computational integer programming and cutting planes. In: Aardal, K., Nemhauser, G.L., Weismantel, R. (eds.) Discret. Optim., volume 12 of Handbooks in Operations Research and Management Science, chapter 2, pp. 69–122. Elsevier, Amsterdam (2005)
36. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-based primal heuristics for mixed integer programming. In: Fujisawa, K., Shinano, Y., Waki, H (eds.) Optimization in the Real World, vol. 13, pp. 37 – 53. Springer Tokyo (2015)

37. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-driven fix-and-propagate heuristics for mixed integer programming. Technical Report 17-56, ZIB, Takustr. 7, 14195 Berlin (2017)
38. Gardi, F.: Toward a mathematical programming solver based on local search. Habilitation thesis, Université Pierre et Marie Curie (2013)
39. Ghosh, S.: DINS, a MIP improvement heuristic. In: Fischetti, M., Williamson, D.P. (eds.) Proceedings of 12th International IPCO Conference on Integer Programming and Combinatorial Optimization, vol. 4513 of LNCS, pp. 310–323. Springer, Berlin (2007)
40. Gleixner, A., Eifler, L., Gally, T., Gamrath, G., Gemander, P., Gottwald, R.L., Hendel, G., Hojny, C., Koch, T., Miltenberger, M., r, B.M., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schlösser, F., Serrano, F., Shinano, Y., Viernickel, J.M., Vigerske, S., Weninger, D., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 5.0. Technical Report 17-61, ZIB, Takustr. 7, 14195 Berlin (2017)
41. Guastaroba, G., Savelsbergh, M., Speranza, M.: Adaptive kernel search: a heuristic for solving mixed integer linear programs. Eur. J. Oper. Res **263**(3), 789–804 (2017)
42. Guzelsoy, M., Nemhauser, G., Savelsbergh, M.: Restrict-and-relax search for 0–1 mixed-integer programs. EURO J. Comput. Optim. **1**, 1–18 (2013)
43. Hansen, P., Mladenović, N., Urošević, D.: Variable neighborhood search and local branching. Comput. Oper. Res. **33**(10), 3034–3045 (2006)
44. Hendel, G.: IPET interactive performance evaluation tools. https://github.com/GregorCH/ipet. Accessed 02 July 2018
45. Johnson, E.L., Padberg, M.W.: Degree-two inequalities, clique facets, and biperfect graphs. N-Holl. Math. Stud. **66**, 169–187 (1982)
46. Koc, U., Mehrotra, S.: Generation of feasible integer solutions on a massively parallel computer using the feasibility pump. Oper. Res. Lett. **45**, 652–658 (2017)
47. 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. Math. Program. Comput. **3**(2), 103–163 (2011)
48. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**(3), 497–520 (1960)
49. Lazić, J.: Variable and single neighbourhood diving for MIP feasibility. Yugosl. J. Oper. Res. **26**(2), 131–157 (2016)
50. Lodi, A.: Mixed integer programming computation. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958–2008, pp. 619–645. Springer, Berlin (2010)
51. Lodi, A.: The heuristic (dark) side of MIP solvers. In: Talbi, E.-G. (ed.) Hybrid Metaheuristics, Volume 434 of Studies in Computational Intelligence, pp. 273–284. Springer, Berlin (2013)
52. Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. In: Theory Driven by Influential Applications, chapter 1, pp. 1–12. INFORMS (2013)
53. Maher, S.J., Fischer, T., Gally, T., Gamrath, G., Gleixner, A., Gottwald, R.L., Hendel, G., Koch, T., Lübbecke, M.E., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Weninger, D., Witt, J.T., Witzig, J.: The SCIP Optimization Suite 4.0. Technical Report 17-12, ZIB, Takustr. 7, 14195 Berlin (2017)
54. Marchand, H., Wolsey, L.A.: Aggregation and mixed integer rounding to solve MIPs. Oper. Res. **49**(3), 363–371 (2001)
55. Margot, F.: Symmetry in integer linear programming. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art, pp. 647–686. Springer, Berlin (2010)
56. Munguía, L.-M., Ahmed, S., Bader, D.A., Nemhauser, G.L., Shao, Y.: Alternating criteria search: a parallel large neighborhood search algorithm for mixed integer programs. Comput. Optim. Appl. **69**(1), 1–24 (2017)
57. Pryor, J., Chinneck, J.W.: Faster integer-feasibility in mixed-integer linear programs by branching to force change. Comput. Oper. Res. **38**(8), 1143–1152 (2011)
58. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J. Comput. **19**(4), 534–541 (2007)
59. Salvagnin, D.: Detecting and exploiting permutation structures in MIPs. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming. Lecture Notes in Computer Science, vol. 8451, pp. 29–44. Springer, Berlin (2014)

60. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA J. Comput. **6**, 445–454 (1994)
61. Wallace, C.: ZI round, a MIP rounding heuristic. J. Heuristics **16**(5), 715–722 (2010)
62. Winkler, M.: Presolving for pseudo-Boolean optimization problems. Diploma thesis, Technische Universität Berlin (2014)
63. Witzig, J., Berthold, T., Heinz, S.: Experiments with conflict analysis in mixed integer programming. In: Salvagnin, D., Lombardi, M. (eds.) Integration of AI and OR Techniques in Constraint Programming, pp. 211–220. Springer, Cham (2017)
64. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. Ph.D. thesis, Technische Universität Berlin (1996)