Mathematical Programming Computation
https://doi.org/10.1007/s12532-024-00269-5

FULL LENGTH PAPER

A fix-propagate-repair heuristic for mixed integer programming

Domenico Salvagnin[1] · **Roberto Roberti**[1] · **Matteo Fischetti**[1]

Received: 15 December 2022 / Accepted: 14 September 2024

Abstract
We describe a diving heuristic framework based on constraint propagation for mixed integer linear programs. The proposed approach is an extension of the common fix-and-propagate scheme, with the addition of solution repairing after each step. The repair logic is loosely based on the WalkSAT strategy for boolean satisfiability. Different strategies for variable ranking and value selection, as well as other options, yield different diving heuristics. The overall method is relatively inexpensive, as it is basically LP-free: the full linear programming relaxation is solved only at the beginning (and only for the ranking strategies that make use of it), while additional, typically much smaller, LPs are only used to compute values for the continuous variables (if any), once at the bottom of a dive. While individual strategies are not very robust in finding feasible solutions on a heterogeneous testbed, a portfolio approach proved quite effective. In particular, it could consistently find feasible solutions in 189 out of 240 instances from the public MIPLIB 2017 benchmark testbed, in a matter of a few seconds of runtime. The framework has also been implemented inside the commercial MIP solver Xpress and shown to give a small performance improvement in time to optimality on a large internal heterogeneous testbed.

Keywords Mixed integer programming · Heuristics · Solution repair · Constraint propagation

Mathematics Subject Classification 90C10 · 90C11 · 90C57

✉ Domenico Salvagnin
domenico.salvagnin@unipd.it

Roberto Roberti
roberto.roberti@unipd.it

Matteo Fischetti
matteo.fischetti@unipd.it

[1] Department of Information Engineering (DEI), University of Padova, Padua, Italy

Published online: 07 October 2024

1 Introduction

In this paper, we consider mixed integer linear programming (MIP) problems of the form:

$$\begin{aligned} \min\ & c^\intercal x \\ L \leq\ & Ax \leq U \\ l \leq\ & x \leq u \\ x_j \in\ & \mathbb{Z}\ \forall j \in J \end{aligned}$$

where $J \subseteq I$ and I is the set of variable indices. Without loss of generality, all constraints can be assumed in the ranged form above, by allowing the row lower bound L_i to be $-\infty$ or the row upper bound U_i to be $+\infty$; equations can be obtained by setting $L_i = U_i$. We also assume all variable bounds to be finite: this is in general a restriction for the MIP paradigm, unless bounds of exponential size are allowed, see for example [31]. The assumption that all variable bounds are finite greatly simplifies both exposition and implementation, in particular for constraint propagation, see Sect. 2, and it is an acceptable restriction for a primal heuristic. However, we note that the algorithms presented in the paper can naturally be extended to deal with infinite bounds. Being MIP an NP-hard problem, there is a long and established literature on designing general-purpose primal heuristics, including [8, 12, 13, 15, 16, 20, 21, 33, 34] and the literature overviews in [1, 3, 5, 18], to either help exact methods like branch-and-cut, or to be run as standalone methods.

Primal heuristics for MIP can be broadly categorised depending on the tools that they are built upon. At one end of the spectrum, we have so-called LP-free heuristics: those are based on simple concepts like rounding and constraint propagation, and, as the name implies, do not assume the availability of a linear programming (LP) solver. As such, they are mostly geared at finding a first feasible solution very early in the solution process, before the first LP is even solved. Examples of LP-free heuristics are *shift-and-propagate* [8], *rapid learning* [7], and the structure-based heuristics in [20]. Then, we have LP-based heuristics, that allow for the solution of linear programming (or, in general, convex) relaxations. Examples of LP-based heuristics are rounding heuristics like *ZI round* [41], diving heuristics [32], and the *feasibility-pump* family [15]. At the other end of the spectrum we have primal heuristics based on the *subMIP* paradigm, i.e., heuristics that are based on the solution of one or more integer subproblems: while those MIPs are typically of smaller size w.r.t. the original problem, they are still relatively expensive w.r.t. to the previously described methods. Examples of subMIP heuristics are, among others, *RINS* [12], *local branching* [16], and *polishing* [34] (also see [24] for an overview of subMIP heuristics). Typically, but not necessarily, LP-free and LP-based heuristics are so-called *constructive* heuristics, i.e., their purpose is to find an initial feasible solution for a given MIP, while subMIP heuristics are *improving* heuristic, i.e., their aim is to improve the quality of an already feasible solution, via some form of large neighborhood search solved with a black box MIP solver. There are exceptions, of course: for example, the RENS [6] and zero-objective [2] subMIPs are constructive heuristics. Finally, the work in [17] uses local branching to (try to)

reduce the infeasibility of an infeasible solution, thus implementing a (more expensive) approach for solution repair than the one presented in this paper.

This paper describes an extended version of framework we developed to participate in the MIP 2022 Computational Competition, and that ranked second there. The competition called for novel general-purpose primal heuristics for mixed-integer linear optimization problems. According to the competition rules, participants were encouraged to develop LP-free heuristics, but were still allowed to solve auxiliary convex optimization problems; subMIPs, on the other hand, were forbidden. We sticked to those restrictions in our development and computational testing. We also note that there has been a renewed interest in fast and LP-free primal heuristics in recent years, partly because of the competition itself, see for example the *feasibility jump* [28] and *Scylla* [30], but also independent of it, like the *Local-ILP* combinatorial local search in [27].

Our starting point is the (folklore) fix-and-propagate heuristic [2, 8, 20, 30]. The idea behind fix-and-propagate is to iteratively *fix* variables according to some ranking of the variables, and *propagate* the linear constraints of the problem after each such fixing before picking the next variable to fix. In general, some very limited form of backtracking (e.g., 1-level backtracking) is also allowed to prevent the dive to be aborted too quickly. Of course, the actual heuristic depends on the strategy used to select the next variable to fix and to which value to fix it.

A weakness of fix-and-propagate is its inability to recover from past mistakes: 1-level backtracking only catches the most recent *wrong turn* but is unable to fix older mistakes. In addition, the success of the method greatly depends on the variable and value selection strategies. An alternative approach to constructive heuristics is the so-called *solution repair* approach. The idea is to start with a complete, yet infeasible, solution to the problem, and then to iteratively change (*shift*) the value of a single variable in the hope of reducing the total amount of infeasibility, until a feasible solution is eventually found. Examples of the solution repair approach are the *shift-and-propagate* heuristic [8] and, in the context of boolean satisfiability, the WalkSAT algorithm [40]. The WalkSAT algorithm, in particular, has been extensively studied by the SAT community, both theoretically [36] and computationally [25, 39]. While we found that applying the WalkSAT approach to MIP is in general not very effective, some of its key ideas are used in our proposed framework as well.

The outline of the paper is as follows. In Sect. 2, we introduce basic concepts and notation, while in Sect. 3 we describe our fix-propagate-repair framework. In Sect. 4, we describe the different variable/value strategies we tried, and in Sect. 5 we describe our solution repair approach. Extensive computational results are given in Sect. 6. Future developments are discussed in Sect. 7.

2 MIP notation and concepts

In this section we introduce the basic concepts and notation used to describe constraint propagation and solution repair. Constraint propagation is a form of inference that consists in removing values from the domain of one of more variables if it can be shown that those cannot be part of any feasible solution given the problem constraints.

In the integer programming community, this often goes under the name of bound strengthening [29, 35], and consists in tightening the bounds of the variables based on the linear constraints of the problem. For a single linear constraint, the logic of bound tightening is based on the so-called minimum ($r^{\min}$) and maximum ($r^{\max}$) *activities*. Let us consider the $\geq$ side of a linear constraint (similar arguments can be made for the $\leq$ side):

$$\sum_{j \in P} a_j x_j + \sum_{j \in N} a_j x_j \geq L$$

where P and N denote the set of variables appearing in the constraint with positive and negative coefficients, respectively. Then, under the assumption that all variable lower and upper bounds are finite, we can compute the activities of the constraint as:

$$r^{\min} = \sum_{j \in P} a_j l_j + \sum_{j \in N} a_j u_j$$
$$r^{\max} = \sum_{j \in P} a_j u_j + \sum_{j \in N} a_j l_j$$

Note that the activities provide lower and upper bounds on the value of the linear expression of a constraint given the current variable bounds. Once those activities are computed, we can compute lower bounds for the variables in P as:

$$\bar{l}_j = u_j - \frac{r^{\max} - L}{a_j}$$

and upper bounds for the variables in N as:

$$\bar{u}_j = l_j - \frac{r^{\max} - L}{a_j}$$

Note that for integer constrained variables we can round down (resp. up) the new upper (resp. lower) bounds. The minimum activity plays a similar role for the $\leq$ side of a constraint. Moreover, note that the activities above can also be used to detect when a linear constraint becomes infeasible ($r^{\max} < L$ or $r^{\min} > U$) or redundant ($r^{\min} \geq L$ and $r^{\max} \leq U$). Finally, note that the logic can be extended to deal with infinite bounds: the intuition stays the same, but we need to keep track of how many variables contribute with an infinite amount to a given activity, and we can derive a tightened bound only if the corresponding activity is finite, see [23, 29] for details.

A set of linear constraints can be propagated iteratively: each constraint is given a chance to tighten bounds at the very beginning, and then it is called again if the domain of a variable that appears in it is changed, until no more reductions can be found (a so-called *fixpoint* has been reached). The complexity of the scheme depends on the domain of the variables: it is polynomial for binary variables (each binary variable can be tightened at most once before fixpoint), but exponential for general integer variables

and might not even converge in a finite number of steps for continuous variables, see for example [19]. For these reasons, practical implementations use limits on how many times the domain of a single variable can be changed within a propagation call or on the minimum changed that is allowed. We refer to [38] for a general overview of constraint propagation systems.

When dealing with binary variables, it is often convenient to introduce the concept of *literal*, which is either a binary variable itself x_j or its negation $1 - x_j$, with the obvious mapping that a literal is considered true iff the corresponding expression evaluates to 1. A very common class of constraints that can be described in terms of literals is the one containing the so-called *clique* constraints, i..e, linear constraints of the form:

$$\sum_{j \in P} x_j + \sum_{j \in N} -x_j \leq 1 - |N|$$

where P (resp., N) denotes the set of positive (resp., negative) literals in the clique. Intuitively, a clique constraint is satisfied if at most one literal is true. We also consider *equality* cliques, i.e., cliques that require exactly one literal to be true. Given an arbitrary set of binary variables, we can define its clique cover C as an ordered partition of the set into disjoint subsets, each of which is covered by a clique constraint in the problem. The order of the cliques in the partition will be important later on when considering variable strategies. Notice that a clique cover always exists, as in the worst case any binary variable can be considered as part of a (trivial) clique of size one. In practice, due to the large use of binary variables in MIP modeling, clique constraints are very common in MIP instances, so a non-trivial clique cover involving a significant portion of the binary variables exists in most cases. Clique constraints in the model can easily be identified by a linear scan over the constraint matrix and put into a dedicated data structure (the so-called *clique table*), so that we can efficiently iterate over the cliques and query which cliques a given variable appears in. We note that a more advanced implementation could extract cliques from other constraints (for example, from knapsack constraints), or obtain additional cliques via probing [35]. Clique covers will be used extensively in our variable strategies.

Another concept that is relevant for at least one strategy is variable locks [1]. Given a MIP with constraints in $\geq$ form, the number of *down locks* (resp. *up locks*) of a given variable x_j is given by the number of linear constraints where the variable appears with a positive (resp. negative) coefficient. An analogous definition applies to linear constraints in $\leq$ form. Note that a constraint with finite row bounds L_i and U_i always contributes to both the number of up locks and down locks of any variable appearing in it. Intuitively, variable locks just count how many constraints could become infeasible if changing the value of a variable in each direction. In the following, we will denote the up and down locks of a given variable x_j with the symbols σ_j^+ and σ_j^-, respectively.

Finally, some strategies make use of a point in the interior of the LP-feasible region, a so-called *corepoint*. In line with previous works [9, 10], we use the analytic center x^{ac} of the LP relaxation as our corepoint. Given a MIP with m linear constraints, all

in $\geq$ form with right-hand-side b, the analytic center x^{ac} is defined as:

$$x^{ac} = \arg\max \left\{ \sum_{j \in I} \log(x_j - l_j) + \sum_{j \in I} \log(u_j - x_j) + \sum_{k=1}^{m} \log(a^k x - b_k) \right\}$$

The analytic center can be efficiently computed in practice by interior point methods: in our implementation, we just solve the initial LP relaxation without objective function with the barrier algorithm without the final crossover step.

3 Fix-propagate-repair framework

The main idea behind our fix-propagate-repair framework is to augment the fix-and-propagate scheme with a repair step, whose purpose is to recover the feasibility of the current partial assignment without backtracking. In particular, we allow for changes not just to the most recent variable fixed, as 1-level backtracking would do, but also to previous ones. In preliminary computational experiments, this *incremental* repair proved to be more effective than a pure WalkSAT approach that tries to repair a complete solution. In practice, our framework can be seen (and is implemented) as a limited depth-first search strategy, where at each node we can decide whether to do constraint propagation, whether to allow for solution repair, and whether to backtrack in case the infeasibility is not resolved. To implement depth-first search (DFS) non recursively and to support efficient constraint propagation and repair, we make use of a few data structures. In particular, we have a *propagation engine* object E which is in charge of maintaining the current domain of the variables, initializing and incrementally updating minimum and maximum activities for all constraints whenever a domain change occurs (for whatever reason), and to provide backtracking functionality. Since we are performing DFS, backtracking can be implemented with a so-called *trail* [37], i.e., a stack where we record in LIFO order the changes that we apply to the variable domain, together with their previous value. With the trail, we can easily support partial undo by popping changes from the stack (in reverse order, and up to the required pointer) and applying the reverse change. We assume the trail to be part of E, and that we can query a pointer to the current top of the trail via a function `TrailP`(E). We also assume that the current partial assignment is fully encoded by the current domain, which is also part of E, and that we can obtain it via the function `Domain`(E). Then we maintain a stack Q of open nodes: each node is described by a branching bound change (empty for the root node) and by a pointer to the last bound change on the trail when the node was created.

A generic pseudocode for the proposed framework is given in Fig. 1. The algorithm starts at line 1 by creating and initializing a propagation engine E from the problem P. Then we create an empty node stack and push an empty fixing (together with the current trail pointer), to serve as root node. Then we perform a standard DFS search where, at each iteration, we pop a node from the stack (line 5), we process it (lines 8 to 13) and, if needed, branch by creating new nodes (lines 14 to 22). At each node, depending on the parametrization, we can perform constraint propagation, try a repair

```
input  : A MIP instance P, limits and parameters propagate, repair,
         backtrackOnInfeas
output : A feasible solution if found, None otherwise
E ← InitEngine(P)
Q ← empty stack
Push(Q, {∅, TrailP(E)})                    // push root node to stack
while Q not empty and limits not reached:
    fixing, tp ← Pop(Q)                    // pop a node from stack
    Backtrack(E, tp)                       // backtrack to parent node
    infeas ← Apply(fixing, E)              // apply branching fixing
    // Node processing
    if not infeas and propagate :
        infeas ← Propagate(P, E)
    if infeas and repair:
        infeas ← Repair(P, E)
    if infeas and backtrackOnInfeas:
        continue
    branches ← Branch(P, E)                // branch according to strategy
    if branches is empty:
        if infeas:
            continue
        else:
            return Domain(E)               // feasible solution found
    else:
        foreach b in branches do
            Push(Q, {b, TrailP(E)})        // push new nodes to stack
return None
```

Fig. 1 Fix-propagate-repair scheme

step, and backtrack on an infeasible node. Different values for the parameters can yield quite different algorithms. For example, a regular fix-and-propagate search can be obtained by enabling propagation and backtrack, and disabling repair. On the other hand, a simple incremental repair strategy can be obtained by enabling repair and disabling propagation and backtrack. Note that, in the scheme above, the problem itself is never modified, and all the state is captured in the propagation engine E and stack Q.

Because of its DFS nature, the algorithm itself is complete, in the sense that it will return a feasible solution if there is one, if limits are set large enough. Of course, in practice the algorithm is called with very strict limits: in our runs, the node limit was set to be equal to the number of variables in the problem to be solved plus one, and the search stops when the very first feasible solution is found.

As to parametrizations, we used the following settings:

- dfs: propagation enabled, repair disabled, and backtrack on infeasibility; this is a regular fix-and-propagate scheme.
- dfsrep: same as `dfs`, but with repair enabled; notice that, in this case, the algorithm might perform redundant work, as the subtrees in the search are no longer necessarily disjoint.

dive: propagation disabled, repair enabled, and no backtrack on infeasibility; this is an incremental repair strategy that constructs a complete solution in a single long dive.

diveprop: same as `dive`, but with propagation enabled.

In the following section, we describe the variable/value strategies that we designed and tested. Details about the repair process itself are deferred to Sect. 5.

4 Strategies

Once a parametrization has been set, we still need to specify a variable/value strategy in order to obtain a fully specified method. Clearly, there are many different possibilities. Strategies can be either *static*, i.e., computed upfront "once and for all" before running the heuristic, or *dynamic*, where the ranking is updated at each node taking the current domain into account. Clearly, a static strategy is simpler to implement and faster to run, as no update is necessary after each fixing/propagation, but it could require more nodes than a more sophisticated dynamic approach. In addition, we can use different criteria depending on the main target of the heuristic: for example, if the emphasis is on feasibility, we might discard objective-based information (such as, e.g., reduced costs) and stick to feasibility-based measures, such as variable locks. Finally, we can distinguish between *LP-free* and *LP-dependent* strategies: for example, a strategy ranking variables by their fractionality in an LP solution is LP-dependent, while one just looking at the objective coefficients is LP-free. We note that variable and value strategies affect only the implementation of the function `Branch` in the pseudocode of Fig. 1.

4.1 Variable strategies

In our computational study, we implemented only static variable strategies, and tried both LP-free and LP-dependent criteria. However, we did not experiment with sophisticated objective-based methods (such as reduced costs), as the emphasis of the present work is on finding a first feasible solution. We argue that solution quality is better obtained by other means, in particular by subMIP-based local searches [24], like for example *RINS* [12] or *local-branching* [16]. For ease of future reference, the variable strategies we tested are listed in Table 1, with a brief description.

Basic strategies. The most trivial strategy is `LR`: it just uses the order of the variables as they appear in the formulation, without any further computation or logic. All other strategies take, at least, variable types in account, and their common first step is to bucket sort variables into binary, general integer and continuous variables, in this order. Notice that bucket sort is a stable sorting algorithm and thus, without additional second-level criteria, variables still appear in formulation order within each bucket. `type` is the outcome of this first step. Then we have strategies that introduce additional criteria *within* each bucket: `random` just does a random shuffle, while `locks` sorts the variable by non-decreasing value of $\max(\sigma_j^+, \sigma_j^-)$, where up and down locks are computed statically upfront.

Table 1 Variable strategies tested in our implementation

Name	Description
LR	Left-to-right as they appear in the formulation
type	Grouped by type (binary, integer, continuous)
random	Random shuffle within each type
locks	Non-increasing variable locks within each type
typecl	As `type`, but use clique cover to sort binary variables
cliques/cliques2	Based on clique cover and relaxation solution

Clique covers. The remaining three strategies, `typecl`, `cliques` and `cliques2`, are, to the best of our knowledge, novel and they are all based on the concept of a *clique cover*, as introduced in Sect. 2. The intuition is to compute a clique cover over the binary variables, and use that to change their ranking. As a result, those strategies can only differ in the ranking of binary variables, with general integer and continuous being unaffected. In particular, if no binary variable is present, they are all equivalent to `type`.

In general, for a given instance, there exist many different clique covers: since constructing one that optimizes a given criterion (for example, minimizing the number of subsets in the partition) is computationally hard and practically expensive, we resort to simple greedy heuristics, each yielding a different strategy. The first greedy algorithm that we describe is the one used by the strategy `typecl`, and works as follows. We first process equality cliques (if any): this is done by just iterating once over the equality cliques, in the order in which they appear in the model, and adding a clique to the cover if and only if it is disjoint w.r.t. to all the ones that have already been added, so that we can keep track of the equality status. Then, we try to cover the remaining binary variables by:

- counting how many binary variables are covered by each clique;
- assigning each binary variable to the largest clique covering it;
- counting how many binary variables are covered by each of the selected cliques;
- doing some local adjustments (e.g., switch a variable to a larger selected clique);
- finally sorting the selected cliques by size.

The ordered list of cliques obtained in this way implicitly defines an order among groups of binary variables. Within each clique, variable are then sorted according to the order in which they appear in the clique itself. Binary variables not covered by any clique are moved to the end of the binary bucket, again in formulation order. This strategy is LP-free. The same clique cover is also used in strategy `cliques`, where however we exploit the analytic center x^{ac} in order to sort variables within each clique in the cover. In details, we weigh the literals in a clique using their value in x^{ac}, and then use those weights to sample from a *weighted discrete distribution*. The intuition is that we interpret the values in the analytic center as probabilities, and we base a *randomized sorting* on those. A pseudocode for the sorting of variables within each clique in strategy `cliques` is given in Fig. 2.

```
input  : variable bounds l, u, clique cover C, analytic center x^ac,
output : sorted list of binary variables S
S ← [ ]
foreach clique cl in C do
    vars ← [ ]
    foreach literal lit in cl do
        if l_j = u_j:
            continue
        j ← Var(lit)
        Append(vars, j)
        if IsPos(lit):
            w_j ← x^ac_j
        else:
            w_j ← 1 − x^ac_j
    foreach j in vars do
        w_j ← log(Rand(0,1) / w_j)
    Sort(vars, w)
    foreach j in vars do
        Append(S, j)
return S
```

Fig. 2 Pseudocode for sorting variables within each clique for strategy `cliques`. The code assumes the primitives `Var`(lit) returning the variable associated with a given literal, and `IsPos`(lit) returning whether is literal is positive or not. It also assumes that we can draw pseudorandom numbers from the uniform distribution in the range [0, 1] with the function `Rand`(0, 1), and that we can sort a list v of elements in non-decreasing w.r.t. a set of weights w with the function `Sort`(v, w)

Finally, in strategy `cliques2`, we construct a clique cover dynamically using both the clique table and a reference LP solution, in this case, a zero-objective vertex. The method is quite simple: we just loop over the cliques in the problem, skipping fixed variables and non-tight cliques (w.r.t. to the reference LP solution), pick the most positive literal in the clique and then the remaining uncovered binary variables (again, in the order they appear in the clique). A pseudocode for the sorting of variables within each clique in strategy `cliques2` is given in Fig. 3.

4.2 Value strategies

As far as value strategies are concerned, we also have several options. Those we implemented are listed in Table 2, again with a brief description.

Let us give a few more details about the value strategies. As for value selection, by *against objective* (resp., *toward objective*) we mean picking the bound that makes the objective worse (resp., better): if the variable does not appear in the objective, we always pick the lower bound. In `loosedyn`, we prefer the direction over which the variable has fewer locks, but we compute those locks dynamically, using the current minimum and maximum activities for each constraint, as explained in Sect. 2. Thus, constraints that have already become redundant do not contribute to the lock counters. Finally, in LP-based strategies, we take a reference LP solution x^{LP} and use the fractional part $f_j = x_j^{LP} - \lfloor x_j^{LP} \rfloor$ of the LP value of a variable p_j as a probability to

```
input  : variable bounds l, u, clique table T, LP solution x^LP,
output : sorted list of binary variables S
S ← [ ]
foreach clique cl in T do
    sum ← 0
    bestVar ← None
    bestValue ← 0
    foreach literal lit in cl do
        j ← Var(lit)
        if IsPos(lit):
            if l_j = 1:
                skip clique
            v ← x^LP_j
            sum ← sum + v
            if v > bestValue and u_j = 1:
                bestVar ← j
                bestValue ← v
        else:
            if u_j = 0:
                skip clique
            v ← 1 − x^LP_j
            sum ← sum + v
            if v > bestValue and l_j = 0:
                bestVar ← j
                bestValue ← v
    if bestVar ≠ None and sum = 1:
        Append(S, bestVar)
        foreach literal lit in cl do
            j ← Var(lit)
            if j ≠ bestVar:
                Append(S, j)
return S
```

Fig. 3 Pseudocode for sorting variables within each clique for strategy `cliques2`. The code assumes the primitives `Var`(lit) returning the variable associated with a given literal, and `IsPos`(lit) returning whether is literal is positive or not

Table 2 Value strategies tested in our implementation

Name	Description
up	Always prefer the upper bound
random	Randomly between lower and upper bound
goodobj	Fix toward the objective
badobj	Fix against the objective
loosedyn	Compute dynamic locks based on current activities
LP-based	Use fractional part of LP value as probability to pick upper bound

Table 3 Overall variable/value strategies

Name	Variable	Value
random	typecl	random
random2	random	random
badobj	type	badobj
badobjcl	typecl	badobj
goodobj	type	goodobj
goodobjcl	typecl	goodobj
locks	LR	loosedyn
locks2	locks	loosedyn
cliques	cliques	up
cliques2	cliques2	up
zerocore	typecl	zerocore
zerolp	typecl	zerolp
core	typecl	core
lp	typecl	lp

pick the current value rounded up as preferred value. In details, the preferred value v_j of variable x_j is selected as:

$$v_j = \begin{cases} \lceil x_j^{LP} \rceil & \text{with probability } f_j \\ \lfloor x_j^{LP} \rfloor & \text{otherwise} \end{cases}$$

The formula above applies not only to binary variables, but also to general integer ones. In addition, since the reference solution x^{LP} might no longer be feasible for the current domain, the preferred value v_j is always clipped to the current domain $[l_j, u_j]$. Again, the logic of value selection is simplified by the assumption that all bounds are finite.

As we have two independent choices depending on which LP solution to use as a reference (namely: whether to use the zero or original objective and whether to use a interior point or a vertex solution), in the end we have four different LP-based value strategies, that we call `zerocore`, `zerolp`, `core` and `lp`, with the obvious meaning.

All strategies but `loosedyn` are static, in the sense that they compute a preferred value upfront. However, such a value is in any case dynamically clipped to the domain of the variable at the time we branch on it.

Combining all variable/value strategies would give a total of 63 different combinations: however, after some preliminary tests, we decided to select only 14 combinations for further testing, as the other combinations either performed worse or were not significantly different from the chosen ones. Those strategies are listed in Table 3.

5 Solution repair

Our repair strategy is based on the classical WalkSAT [40] approach for boolean satisfiability problems. The algorithm starts with a complete random assignment to the boolean variables. Then, until a satisfiable assignment is found or computational limits are hit, a boolean variable is flipped according to a very specific logic, which can be interpreted as a careful mix of *greedy* and *random walk* behaviour:

- Pick an unsatisfied clause C uniformly at random;
- If some boolean variables in C can be flipped without breaking any currently satisfied clause, pick one randomly;
- Otherwise, pick a random variable in C with probability p, and a variable in C which breaks the minimal number of clauses with probability $1 - p$, where p is the so-called *noise* parameter.

Interestingly, only the *damage* done by a flip is taken into account when computing its score, while the number of clauses that becomes satisfied by the flip is ignored.

In order to apply WalkSAT to the MIP case, even in the most straightforward sense, we need to generalize its logic to non-binary variables and to general linear constraints, as opposed to plain clauses. For non-binary variables, the natural extension of a *flip* is that of a *shift*: we are allowed to shift the value of a fixed variable (binary or not) within its original domain. Of course, this opens some degrees of freedom: in the binary case, once a variable is chosen, there is only one possibility, while in the non-binary case we might have a very large domain. As to linear constraints, computing a violation measure is still doable, but we now have a choice on which one to use: while for clauses cumulative violation and violation count coincide, as a violated clause always has a violation of exactly one, for linear constraints the two measures are different. We also note that while flipping a binary in a violated clause is guaranteed to make that clause feasible, the same does not hold for a linear contraint: for example, shifting the value of a non-binary variable might increase violation in any direction if the constraint is an equality.

In our implementation, we use total violation as a measure, and compute the shifted value (in the non-binary case) as the minimal shift that makes the constraint feasible. If the constraint cannot be made feasible, we still shift the variable (within its domain) so as to reduce the violation as much as possible. In detail, given the current complete assignment x^* and a violated linear constraint $a^\intercal x \geq L$, we compute the best shift s_j for each non-binary variable x_j as:

$$s_j = \frac{L - a^\intercal x^*}{a_j}$$

If the variable x_j is non-continuous, we round down (resp. up) the shift s_j if its value is positive (resp. negative), in order to maintain the integrality of the current assignment, and finally clip the shift such that the new (potential) value $x_j^* + s_j$ is within the original bounds $[l_j, u_j]$.

In preliminary experiments, we tested a version of WalkSAT generalized to work on MIPs. While its behaviour was in line with public-domain implementations when run

on MIP models encoding SAT problems, the results were quite poor on more general MIPs, as for example the instances from the MIP 2022 Computational Competition. We noticed that the method could quickly reduce the amount of violation of the initial random solution at the beginning, but it had then troubles in actually bringing the violation to zero. This is what prompted our strategy of applying solution repair not on complete assignments, but as a repair procedure within the fix-and-propagate search.

In order to apply WalkSAT within a search scheme, we need to extend its logic further, from complete assignments to partial assignments. The partial assignment at the current node in the DFS is encoded by the current domain $[\bar{l}, \bar{u}]$. From this domain we compute, for each constraint $L_i \leq a_i^\mathsf{T} x \leq U_i$, the minimum and maximum activities $r_i^{\min}$ and $r_i^{\max}$, and define the violation V_i of the constraint as:

$$V_i = \max\{(L_i - r_i^{\max})^+, (r_i^{\min} - U_i)^+\}$$

Note that activities are needed for constraint propagation as well, a fact that is exploited by our implementation: those quantities are incrementally kept up-to-date by both propagation and repair steps.

Given a violated constraint, the logic for computing the set of candidates for shifting also needs to be slightly modified. When dealing with complete assignments, all variables are by definition fixed. With partial assignments, we might have to shift any variable that had its domain reduced: in other words, in general we do not shift just values, but intervals. The reason for this need is easily shown by the following example. Suppose we have a problem with a binary variable x and a continuous variable y with domain [0, 10], and the constraints model the implications $x = 0 \rightarrow y \leq 3$ and $x = 1 \rightarrow y \geq 6$. After fixing $x = 0$, constraint propagation reduces the domain of y to [0, 3], but let us suppose that $x = 0$ triggers an infeasibility in some other parts of the model, so that solution repair needs to flip x to 1: there would be no way to obtain a feasible partial assignment if the repair process would be forbidden to shift the domain of y as well[1]. More formally, given the current domain $[\bar{l}, \bar{u}]$ and a violated constraint $L_i \leq a_i^\mathsf{T} x \leq U_i$, for each non-binary variable x_j in it we compute the best shift s_j as:

$$s_j = \begin{cases} \frac{U_i - r_i^{\min}}{a_j} & \text{if } r_i^{\min} > L_i \\ \frac{L_i - r_i^{\max}}{a_j} & \text{otherwise} \end{cases}$$

Again, if the variable x_j is non-continuous, we round down (resp. up) the shift s_j if its value is positive (resp. negative), and finally clip the shift such that the shifted domain $[\bar{l}_j + s_j, \bar{u}_j + s_j]$ is within the original bounds $[l_j, u_j]$. A variable x_j is a candidate for shift only if $s_j \neq 0$ and the violation of the constraint is reduced by applying the shift.

[1] Note that we do *not* allow domain enlargement as a repair move, as it would lead to trivial repair actions where fixings are just undone.

```
   input  : A MIP instance P, noise parameters p, current state E
   output : updated state E
 1 bestViol ← TotalViolation(E)            // initial violation
 2 bestPointer ← TrailP(E)
 3 while ∃ violated row and limits not reached:
 4     i ← PickRandom(ViolatedRows(E))
 5     bestDamage ← +∞
 6     cand ← [ ]
 7     foreach j in Support(P, i) do
 8         s_j ← ComputeShift(P, E, i, j)
 9         if s_j = 0:
10             continue
11         if EvalDeltaViol(E, i, j, s_j) ≥ 0:
12             continue
13         bestDamage ← min{damage, EvalDamage(E, j, s_j)}
14         Append(cand, j)
15     if |cand| = 0:
16         continue
17     if bestDamage > 0 and Rand(0, 1) < p:
18         j* ← PickRandom(cand)
19     else:
20         cand = Filter(cand, bestDamage)
21         j* ← PickRandom(cand)
22     ApplyShift(E, j*, s_j*)
23     currViol ← TotalViolation(E)
24     if currViol < bestViol:
25         bestViol ← currViol
26         bestPointer ← TrailP(E)
27 Backtrack(E, bestPointer)              // backtrack to best state
28 return E
```

Fig. 4 `RepairWalk` scheme. The code assumes the availability of utility functions to pick an element uniformly at random from a list (`PickRandom`($list$)), and to filter a list of variables according to a given upper bound u on damage (`Filter`($list, u$))

A pseudocode describing the overall repair logic based on WalkSAT can be found in Fig. 4. In the pseudocode, we assume that the propagation engine E keeps tracks of which constraints (rows) are violated, and that we have functions to get the list of violated rows (`ViolatedRows`(E)), to get the cumulative violation (`TotalViolation`(E)), to evaluate the effect on the violation of row i of a shift of value s_j for variable x_j (`EvalDeltaViol`(E, i, j, s_j)), and to evaluate the damage of a shift (`EvalDamage`(E, j, s_j)). Function `ComputeShift`($P, E, i, j,$) computes the best shift s_j for the variable x_j w.r.t. row i, according to the logic just described. The function `RepairWalk` can be used as is in Fig. 1 as repair procedure (line 11).

The code starts at line 1 by storing the current violation and a pointer to the current state. Then, at each iteration of the main loop, we pick a violated linear constraint i uniformly at random (line 4) and compute the list of candidates `cand`, i.e., the list of variables x_j appearing in the selected constraint for which the shift s_j is non-zero and that can reduce the violation V_i. For each candidate we also evaluate its damage (line 13), i.e., the total increase in violation over all constraints for which violation

actually increased (as in the original WalkSAT, we ignore improvements in violation) as a result of the shift, and keep the best damage. If the candidate list is not empty, then we proceed with the usual WalkSAT logic (lines 17–22): if there are variables that can be shifted with zero damage, we pick one at random among those; otherwise, we pick a random variable (among the candidates) with probability p, and choose among the candidates with minimal damage with probability $1 - p$, where p is again the noise parameter. We also keep track of the best state that we encountered in our walk (lines 23–26), as this is the state that we will backtrack to and return at the end of the function.

In our implementation, the noise parameter is set to $p = 0.75$ and we use a small iteration limit of 200 for each call, as the repair function is potentially called very frequently within DFS.

5.1 Repair search

The repair scheme described so far works rather well in practice, but it is still unsatisfactory for several reasons. First of all, the repair process does not make use of constraint propagation: this is quite obvious, as it is not possible to do constraint propagation from an infeasible state, but it would be nice to at least take into account the implications of a given repair move. Also, because the repair moves are simple shifts without propagation, we might need multiple moves in order to implement simple changes like, for example, swapping two binary variables in a clique constraint: because of the random walk nature of the process, it is not even guaranteed that we would easily find the correct sequence of simple flips equivalent to such a “complex" move. Finally, the method is easily trapped into cycles: again, the random walk nature guarantees that we will eventually escape them, but that might take too much time, in particular given the strict resource limits of the repair routine. We experimented with adding a short tabu list and soft restarts to the logic, but their effect was limited.

In order to overcome the issues above, we propose to organize the repair process itself as a search scheme, instead of a random walk. The idea is to have an auxiliary DFS search, not in the space of partial assignments, but in the space of move sequences. Whenever a shift is chosen as a repair move, we turn it into a repair disjunction, so that we can undo it on backtrack and try the opposite move in a systematic manner. Then, we can do constraint propagation on this auxiliary tree and obtain the implications of the repair moves done on the path to the current repair node. We are still not allowed to do constraint propagation on the current (still infeasible) partial assignment, but at least we get the implications of our moves. If we detect infeasibility, we can backtrack in the repair space and avoid wasting time in a hopeless subtree. Note that the auxiliary tree deals with two propagation engines at the same time: one that encodes the current partial solution that we want to repair (and the repair changes that we have applied so far) and a repair engine, which is initialized from the global domain, and that is used to derive the implications of the repair moves.

This approach has several advantages (and some disadvantages). On the positive side, the systematic search over moves takes care of avoiding cycles. Also, we can exploit constraint propagation and have a higher chance of doing “complex" moves,

```
input  : A MIP instance P, noise parameters p, current state E
output : updated state E
R ← InitEngine(P)
bestViol ← TotalViolation(E)                    // initial violation
Q ← empty stack
Push(Q, {∅, TrailP(E), TrailP(R), bestViol})
  // push root to stack
while Q not empty and limits not reached:
    move, tp, tpr, viol ← Pop(Q)                // pop a node from stack
    Backtrack(E, tp)                            // backtrack to parent node
    Backtrack(R, tpr) infeas =← Apply(fixing, R)
     // apply branching to R
    if not infeas:
        infeas ← Propagate(P, R)
                                                // propagate move
    if infeas:
        continue
    SyncChanges(E, R, tpr, TrailP(R))
    viol ← TotalViolation(E)                    // current violation
    if viol == 0:
        return E
    bestViol ← min{viol, bestViol}
    if not enough progress:
        BacktrackBestOpen(Q, E, R)
    move ← FindRepairMove(P, E, p)              // find repair move
    if move is None:
        continue
    else:
        branches ← MoveToDisjunction(E, R, move)
        foreach b in branches do
            Push(Q, {b, TrailP(E), TrailP(R), viol})
BacktrackBestOpen(Q, E, R)
return E
```

Fig. 5 RepairSearch scheme

like a swap of binary variables: if we flip a binary to one in a clique constraint, constraint propagation will take care of fixing the other ones to zero, thus automatically flipping the old variable to one (if any). On the negative side, we now have all the shortcomings of regular DFS, like the inability to escape a wrong subtree without exploring it all. We (partially) sidestep this issue by allowing for jumps in the tree: if we detect that we are not making enough progress in the current subtree, we backtrack directly to the most promising open node, at the cost of giving up on completeness (this is not a concern in practice, as we run this repair search with very strict limits anyway). Finally, organizing the repair process as a tree search further complicates the algorithm: we now need to figure out how to turn a repair move into a repair disjunction, and we need to describe how to synchronize the domains of the two engines.

A pseudocode for the proposed search-based repair process is given in Fig. 5. The search is still based on a stack of open nodes, but this time each node stores two trail pointers (one for the current partial solution, encoded into engine E, and one for the

repair engine R) instead of one, and also the overall violation of the node, so that we can backtrack to the best open node in the stack when jumping. The algorithm starts at line 1 by creating a new propagation engine R, initialized from the global domain of P, for the purpose of propagating the implications of repair moves. At each iteration of the search, we pop a node from the stack (line 6) and reset both engines to the state of the parent node. Then we apply the branching changes to R and propagate: if this results in an infeasibility we drop the node (the repair moves are themselves infeasible), otherwise we apply the new range of changes, specified by a pair of trail pointers, from R to E (function `SyncChanges` at line 13). The logic for synchronizing changes from the domain D_r of a variable in R to the domain D of the same variable in E is, as usual, much simpler for binary variables than for non-binary ones. In the binary case, a variable x_j can only be fixed to a given value: if x_j is still unfixed in the current node (this can happen if x_j is implied by a repair move but not by the original partial assignment), then we just fix it to the same value. Otherwise, we either leave it as is (the two domains already agree on x_j) or we flip it. For non-binary variables, in general we need to compare two intervals. Two cases can arise:

1. the two intervals have a non-empty intersection: in this case we can tighten the bounds in D so that the new interval is the intersection;
2. the two intervals are disjoint: in this case we fix the variable to the endpoint of D_r closer to D.

A numerical example can clarify the logic. Let us say that a non-binary variable x_j has domain $[1, 4]$ in D and $[2, 5]$ in D_r: then its new domain in D will be $[2, 4]$ (its domain in D_r being unchanged). Suppose instead that the intervals are $[1, 3]$ and $[4, 5]$, again in D and D_r respectively: then x_j will be fixed to 4 in D. Notice that we always apply changes from D_r to D, never the other way around, and that the rationale is to do the minimal amount of changes to achieve the condition $D \subseteq D_r$. Once changes are synchronized, we compute the new total violation of E: if zero, then repair succeeded and we can stop, returning a feasible state (line 16). Otherwise, we either jump to the best open node (if not enough progress has been made) or we go on and find a new repair move (line 20). The logic for finding a new repair move is the same as a single iteration of the `RepairWalk` main loop. If we can find a move, we turn it into into a repair disjunction. Again, this is trivial for binary variables: a repair move fixes $x_j = b_j$ on the preferred branch, and the other side of the disjunction is simply $x_j = 1 - b_j$. In the non-binary case, a repair move always guarantees that the shifted interval $[a, b]$ in D is contained in the interval $[c, d]$ in D_r. We compute the gaps to the left and to the right w.r.t. to $[c, d]$, i.e., $l = a - c$ and $r = d - b$, and the disjunction is then as follows: if $l \leq r$, we impose $x_j \leq b \vee x_j \geq b$, otherwise $x_j \leq a \vee x_j \geq a$. Once we have a disjunction, we push the corresponding nodes on the stack. Finally, after the search is stopped, we backtrack to the best open node and return the state E associated with it.

6 Computational results

We implemented our framework in C++ and used IBM ILOG CPLEX 12.10 [26] as LP solver and MIP presolver. All codes were run on a cluster of 24 identical machines, each equipped with an Intel Xeon CPU E3-1220 V2 CPU running at 3.10 GHz, and 16 GB of RAM, and take full advantage of multi-threading. Each method was run on the 240 instances from the MIPLIB 2017 [22] benchmark set, using different random seeds to limit the effect of performance variability [11]. Each method was run on each instance-seed combination with a time limit of 10 min, as in the MIP 2022 Computational Competition. As far as implementation details are concerned, we mention that:

- The same MIP presolve is applied at the beginning of each run, in order to get a smaller model and tighter bounds on all variables. In addition, if after presolve some variables still have some infinite bound, we impose an artificial bounding box of $[-100000, +100000]$: as already mentioned, this is in general not a valid constraint, but it is acceptable for heuristic purposes.
- Constraint propagation and solution repair work on the same data structures and perform incremental activity updates whenever possible.
- The overall effort spent in constraint propagation (and solution repair) is subject to a deterministic work limit: in particular, we allow a number of matrix accesses of at most 100 times the number of nonzeros in the presolved model. This is to avoid time consuming runs where constraint propagation is too expensive.
- Whenever a feasible solution is found, before adding it to the solution pool, we apply a simple greedy 1-opt step to try to improve its objective. This is important, e.g., for set covering models, where the trivial solution with all variables set to 1 is often found first.

6.1 Preliminary experiments

A first preliminary experiment consisted in running each variable/value strategy in the regular *fix-and-propagate* scheme without any form of solution repair, i.e., the `dfs` parametrization. The purpose of this experiment is to evaluate the effectiveness of the known approach and provide some insight on which variable/value strategies are most promising. For these preliminary tests, we tried 3 different random seeds for each instance (for a total of 720 instance-seed pairs) and let each method perform at most 100 trial dives, in each case stopping at the first solution found. Cumulative results are given in Table 4, where we report the number of solutions found (out of 720), the average primal gap w.r.t. the optimal solution, and the shifted geometric mean [1] of runtime (with a shift of 1 s).

According to the table, there is a relatively large variability in success rate between the different strategies: the worst method (`goodobj`) can find a feasible solution only in 334 runs, while the best strategy (`zerolp`) in 481. Similarly, the average runtime can also vary a lot between strategies, with the best method being again (`zerolp`) with 2.58s, and the slowest being `lp` with 3.66s. As expected, there is far less variability on the average solution quality, which is quite poor regardless of the method. It is

Table 4 Preliminary strategy study on `dfs`

Strategy	#found	Primal gap	Time (s)
random	409	0.75	2.78
random2	362	0.79	3.34
badobj	387	0.79	2.62
badobjcl	393	0.78	2.64
goodobj	334	0.79	3.03
goodobjcl	370	0.77	2.74
cliques	393	0.79	3.49
cliques2	408	0.80	2.88
locks	378	0.78	2.70
locks2	381	0.79	2.81
zerocore	432	0.79	3.07
zerolp	481	0.76	2.58
core	409	0.73	3.55
lp	422	0.72	3.66

also worth noting that while a pure random approach (`random2`) is among the worst methods, allowing us to argue that we are not just looking at noise, still a mildly randomized approach like `random` (where variables are ranked according to the type and greedy clique cover, and the fixing value is picked at random) is one of the best methods. Looking at the detailed results,[2] we can also notice that, while there is a subset of 103 *easy* models where a solution is found by pretty much any strategy, and a subset of 58 *hard* models where no strategy ever finds a solution, for the remaining models there is no clear dominance among strategies. In particular, on some models only few strategies are able to consistently find a feasible solution: for example, `locks2` is the only strategy able to find feasible solutions for instance `rail01`, while `cliques`, though not being very effective on average, is the only strategy able to find solutions for instances `peg-solitaire-a3` and `unitcal_7`. Similarly, while the average runtime is in the same order of magnitude for all strategies, on a single model the difference can be huge: for example, on instance `tbfp-network` the fastest strategy finds a feasible solution in 0.5s, while the slowest does not find one in 400 s. Finally, we note that the *virtual best* among those strategies would be able to find a feasible solution on 547 instance-seed pairs, again an indication of the different behaviour, and lack of dominance, among strategies.

6.2 Effect of solution repair

We now evaluate the effect of solution repair and report in Table 5 the results, on the same models, of all four different parametrizations. For readability, we dropped the three strategies `random2`, `goodobj` and `goodobjcl`, which clearly ranked worst

[2] All the data is available from the authors upon request.

Table 5 Preliminary strategy study

Type	Strategy	#found	Primal gap	Time (s)
dfs	random	409	0.75	2.78
	badobj	387	0.79	2.62
	badobjcl	393	0.78	2.64
	cliques	393	0.79	3.49
	cliques2	408	0.80	2.88
	locks	378	0.78	2.70
	locks2	381	0.79	2.81
	zerocore	432	0.79	3.07
	zerolp	481	0.76	2.58
	core	409	0.73	3.55
	lp	422	0.72	3.66
dfsrep	random	503	0.71	2.64
	badobj	520	0.73	2.44
	badobjcl	512	0.72	2.56
	cliques	499	0.74	3.30
	cliques2	523	0.73	2.73
	locks	521	0.70	2.29
	locks2	515	0.71	2.53
	zerocore	527	0.74	2.86
	zerolp	536	0.73	2.68
	core	505	0.68	3.46
	lp	519	0.66	3.67
dive	random	372	0.78	3.77
	badobj	435	0.78	3.11
	badobjcl	386	0.79	3.70
	cliques	370	0.79	4.84
	cliques2	438	0.77	3.23
	locks	450	0.75	2.70
	locks2	454	0.74	2.65
	zerocore	433	0.78	3.67
	zerolp	461	0.77	3.10
	core	436	0.71	3.94
	lp	444	0.69	4.07
diveprop	random	504	0.70	2.65
	badobj	524	0.72	2.30
	badobjcl	496	0.72	2.65
	cliques	490	0.74	3.44
	cliques2	519	0.73	2.71

Table 5 continued

Type	Strategy	#found	Primal gap	Time (s)
	locks	523	0.69	2.27
	locks2	503	0.71	2.49
	zerocore	525	0.74	2.86
	zerolp	542	0.73	2.63
	core	513	0.68	3.34
	lp	524	0.66	3.55

in the previous experiment. A graphical representation of the same data is available in Fig. 6.

According to the table, solution repair significantly improves the success rate of the different strategies: `dfsrep` consistently finds more solutions than `dfs`, with an improved average primal gap, and with a similar runtime. Interestingly, `dive`, that only applies solution repair without any constraint propagation, is competitive with the original `dfs` as far as success rate and primal gap are concerned, albeit at the cost of being more expensive to run. `diveprop` is again superior to `dive`, and pretty much on par with `dfsrep`. We conclude from those experiments that solution repair is a beneficial addition to fix-and-propagate, and that the most successful methods, `dfsrep` and `diveprop`, actually use both constraint propagation and repair. In addition, it seems that the combination of constraint propagation and solution repair reduces the variability between different strategies, at least as far as success rate is concerned. Despite those improvements, that reduce from 58 to 36 the number of models where we cannot find any feasible solution, it is still true that no single strategy dominates any other, and again some solutions are only found by very specific combinations. This is confirmed by the success rate of the virtual best solver which, at 606, is still significantly higher than any method.

6.3 Portfolio approach

The results of the preliminary experiments, plus the observation that all methods are inherently sequential, motivate a portfolio approach, in which different parametrizations/strategies are run in parallel until one finds a feasible solution and the whole process stops. The benefits are two-fold: first of all, we can exploit parallel hardware, which is standard nowadays; second, given the difference in performance of the different methods on a given instance, we can obtain superlinear speedups with respect to a process that runs each method sequentially.

The portfolio approach we implemented is quite simple: we partitioned the different methods in three different classes, depending on whether they are LP-free, depend on a zero-objective LP solution, or an optimal LP solution. Then we run the methods (in parallel) in a given class and move to the next only if we still have not found a feasible solution. It is important to note that, in general, there are many more potential methods in a class than there are CPU cores available, so we cannot just naïvely run all methods in a class. This would be quite inefficient for at least two reasons:

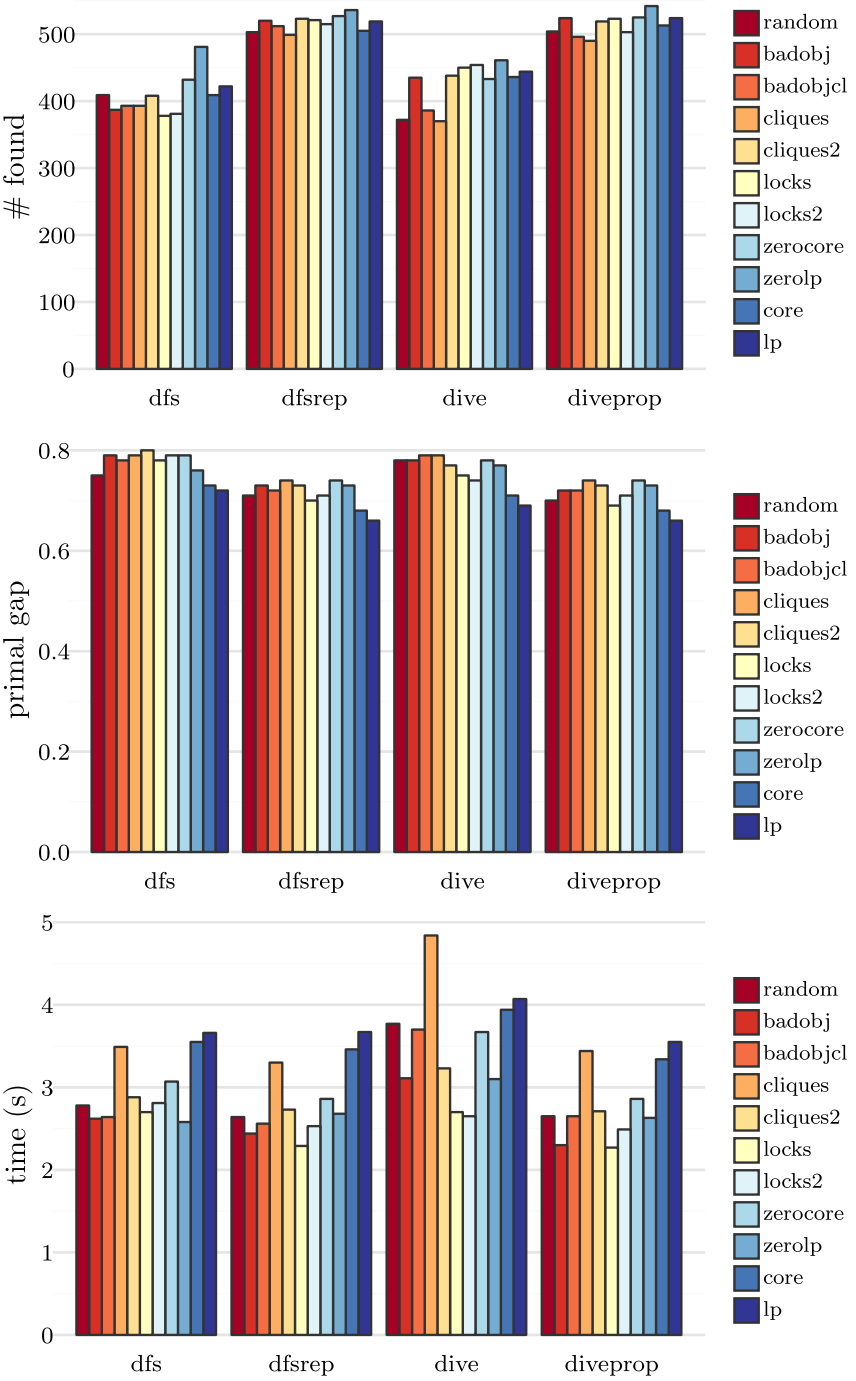

Fig. 6 Comparison between different strategies and parametrizations

1. We risk wasting time running all the methods in a class, potentially not effective on the given model, when maybe a method from a subsequent class could easily find a solution;
2. Although no dominance exists between different strategies, running a few is in general sufficient to *cover* all the instances for which a solution can be found with the method in a class.

In our implementation, we used the detailed results of the preliminary experiments to manually select a subset of methods to run in each class. Our manual selection can probably be improved by using a more principled approach,[3] but we are quite satisfied with its current performance and we did not find it worth to invest additional effort on it. The final detailed logic is described below:

- As already mentioned, we first run the instance through the MIP presolve of CPLEX. Then we perform a first round of constraint propagation, until a fix-point is reached. After that, we fix all trivially-roundable variables (if any) to the corresponding bound.
- Then, we run the following LP-free strategies: `dfs-badobjcl`, `dfs-locks2`, `dive-locks2`, `dfsrep-locks`, `dfsrep-badobjcl` and `diveprop-random`. If we have a feasible solution at this point, we stop.
- We temporarily remove the objective function from the problem and compute a zero-objective corepoint using the barrier algorithm without crossover. This LP solve is also done with some strict deterministic limits, in order to avoid overly expensive LPs.
- At this point, we can run some of the methods that rely on a zero-objective corepoint solution: `dfs-zerocore`, `dive-zerocore`, `diveprop-zerocore` and, if we detect a predominant clique structure, `dfs-cliques`. Again, if we have a feasible solution at this point, we stop.
- We compute with the simplex method a zero-objective vertex solution and try strategies `dfs-zerolp`, `diveprop-zerolp` and `diveprop-cliques2`.
- If still without a solution, we solve the LP relaxation of the problem with the original objective and the concurrent LP solver of CPLEX. Finally, we run the strategies `dfs-lp`, `dive-lp` and `diveprop-lp`.

Note that, in the scheme above, and differently from the preliminary experiments, only a single trial dive is performed by each method.

The aggregated results of our portfolio approach are reported in Table 6, this time over 5 different random seeds. The line corresponding to method `default` gives average figures for the portfolio approach described above. Its success rate is slightly above 80%$(969/(240 \cdot 5))$, up from the 75%$(542/(240 \cdot 3))$ of the best method from the preliminary experiments, with a primal gap matching the best (0.66) and with a significantly smaller runtime (1.24s). The virtual best solver in this case would have found a feasible solution in 1010 cases, which is quite close to the portfolio score.

The remaining part of Table 6 gives the results that we would obtain by stopping the portfolio approach after each stage. It confirms that LP-free methods alone are

[3] Selecting the "best" subset of methods to be run, given the data on a testset, can be cast itself as an optimization problem.

Table 6 Aggregated portfolio results

Method	#found	pgap	Time (s)
lpfree	898	0.69	0.72
zerocore	945	0.67	0.98
zerolp	964	0.66	1.14
default	969	0.66	1.24

Table 7 Repair search vs. repair walk in portfolio approach

Method	#found	pgap	Time (s)
repair-search	969	0.66	1.24
repair-walk	945	0.67	1.09

Table 8 Repair search vs. repair walk contingency table

		Repair-search Success	Not found
repair-walk	Success	927	18
	Not found	42	208

able to find a feasible solution in 898 cases, and do so quite quickly: for reference, the average time for just MIP presolving the problem (something that all of our methods do, and that is accounted for in the timings) is 0.62s. Each additional stage gives a smaller (but non-negligible) contribution, at the expense of increased running times. We note that just preprocessing and solving the first LP relaxation with a concurrent solver would take, on our machines, 1.04s: so our overall portfolio approach is just a little more expensive, on average, than solving the initial LP.

6.4 Impact of repair search

We can evaluate the effectiveness of the repair search described in Sect. 5.1 against the simpler walk-based approach. In Table 7, we report average figures for our portfolio approach using repair search (our default) and using repair walk (`repair-walk`). Using the more sophisticated repair scheme gives a higher success rate (969 solutions founds vs. 945), which in turn gives a slightly better average primal gap. At the same time, it is also marginally more expensive (1.24s vs 1.09s).

In Table 8, we report a standard contingency table to evaluate the effectiveness in finding feasible solutions between the two methods: `repair-search` can find a feasible solution on 42 instance-seed pairs where `repair-walk` cannot, while the opposite happens only in 18 cases. According to a standard McNemar's test, the difference is statistically significant.

Aggregating success rate results by instance over the set of 5 seeds, we have that `repair-search` is a *consistent win*, i.e., the difference between the number of feasible solutions it can find and those found by `repair-walk` is at least 3 on 8 instances, while the opposite happens only on 3 instances.

Table 9 Effect of fix-propagate-repair inside a commercial solver, on the whole testset (upper part) and on the subset of affected models only (lower part)

		Without FPR				With FPR			
Class	N	Solved	Time	Nodes	PDI	Solved	Time	Nodes	PDI
all	4085	3874	181.75	2509	47.08	3875	0.996	0.993	0.995
[0, 3600}	3876	3874	153.68	1898	43.07	3875	0.996	0.992	0.995
[10, 3600}	3755	3753	166.49	2072	45.58	3754	0.996	0.992	0.995
[100, 3600}	2311	2309	346.95	3873	77.79	2310	0.990	0.990	0.994
all	116	104	196.33	1378	90.34	105	0.869	0.769	0.856
[0, 3600}	106	104	147.51	1673	65.88	105	0.855	0.747	0.850
[10, 3600}	106	104	147.51	1673	65.88	105	0.855	0.747	0.850
[100, 3600}	62	60	348.68	2708	144.15	61	0.766	0.678	0.800

6.5 Impact inside a commercial MIP solver

The fix-propagate-repair framework described in the previous sections has been successfully implemented by the first author within the commercial and state-of-the-art MIP solver FICO Xpress [14]. The version inside Xpress is a completely new, yet very accurate, reimplementation of the original academic code. The only major difference is that constraint propagation routines were not reimplemented from scratch, but we rather reused the facilities already available inside the solver. While the implementation is generally equivalent, the way the framework is used inside the solver is quite different w.r.t. its usage as a standalone heuristic, for several reasons. First of all, inside the solver fix-propagate-repair has to compete with the (many) other primal heuristic already implemented there, so the main target is not the absolute success rate, but rather the relative improvement. Finally, and most importantly, the main performance criteria for evaluation of a code change are not just the success rate, but also the improvements on time to proven optimality and in primal-dual integral (PDI) [4]. After extensive tuning, we eventually settled for running the different strategies sequentially (not in parallel, like in the portfolio approach), but in a separate so-called *background* root task, i.e., as an auxiliary computation that is performed in parallel to the main root cutloop if there are idle threads available. The final evaluation was done on the Xpress internal testset, which is a mix of 817 instances coming from publicly available and commercial sources. Tests have been run with 5 different random seeds on a cluster of identical machines, each equipped with a double-socket Intel Xeon Gold 5218 CPU running at 2.8GHz and 128GB of RAM. Each model was run on one socket, with 20 threads and a time limit of 1 h.

Aggregated results are given in Table 9. The format of the table is as follows. For each method under comparison, we report number of solved instances, running time, nodes and PDI. For the reference method we give absolute numbers for all measures, while for the other one we report the ratios (with the exception of the number of solved instances) w.r.t. the reference method. Each row of the table corresponds to a (sub)set of the models in a given class. The first row (`all`) shows the results for

the entire testbed, while the other rows report results for the instances in bracketed subsets. Each bracketed subset is of the form $[t, 3600\}$, and contains all instances that could be solved by at least one method, and where the slowest formulation took at least t seconds to solve the model or timed out. The number of instances in each class is reported in the column N. The bracketing convention is used to show how the speed of different methods changes as the difficulty of the models increase. Time, node and PDI results use a shifted geometric mean [1] with a shift of 10. The upper part of the table reports results on the full testset, while the lower part on the set of affected models only.

Overall, the effect of the new heuristic is (small but) positive. On the harder bracket we see a reduction in time to optimality and PDI of 1%, and no degradation on the other brackets. The small magnitude of the improvement is justified by the very conservative settings used in the implementation, as confirmed also by the relatively small number of affected models (116 / 4085, less than 3%). At the same time, on the affected models the effect is clearly positive on all counts. For those reasons, the heuristic will be available and activated by default in the next FICO Xpress release.

7 Conclusions and future research

We devised a hybrid diving strategy that alternates between constraint propagation and solution repair. While similar to some previous heuristics in the literature, like shift-and-propagate and WalkSAT, the proposed combination is novel and quite effective in finding feasible solutions on a heterogeneous testset like the MIPLIB 2017 benchmark set. A preliminary version of the approach ranked second in the MIP 2022 Computational Competition. In addition, the framework was implemented inside the commercial solver FICO Xpress, where it gave a small improvement in time to optimality and primal-dual integral on the large and heterogeneous testset used for internal solver development. The framework itself is very generic and there are many ideas still left to be tried and evaluated, in particular as far as variable-value strategies are concerned.

Funding Open access funding provided by Università degli Studi di Padova within the CRUI-CARE Agreement. The last two authors were supported by Ministero dell'Università e della Ricerca (MUR), PRIN 2022 project 20229ZWC97 - "Revise and Enhance: Win-win INsights for home Delivery services" (REWIND).

Data availability All the instances used in this study are taken from the public repository at the address https://miplib.zib.de, with the exception of Section 6.5, where a private collection was used to test the performance impact inside a commercial MIP solver.

Code availability The source code is available from the corresponding author upon request.

Declaration

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, Technische Universität Berlin (2007)
2. Achterberg, T., Wunderling, R.: Mixed integer programming: analyzing 12 years of progress. In: Facets of combinatorial optimization, pp. 449–481 (2013)
3. Berthold, T.: Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin (2006)
4. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013). https://doi.org/10.1016/j.orl.2013.08.007
5. Berthold, T.: Heuristic algorithms in global MINLP solvers. Ph.D. thesis, Technische Universität Berlin (2014)
6. Berthold, T.: RENS-The optimal rounding. Math. Program. Comput. **6**, 33–54 (2014)
7. Berthold, T., Feydy, T., Stuckey, P.J.: Rapid learning for binary programs. In: CPAIOR 2010 Proceedings, **6140**, 51–55 (2010)
8. Berthold, T., Hendel, G.: Shift-and-propagate. J. Heuristics **21**(1), 73–106 (2015)
9. Berthold, T., Perregaard, M., Meszaros, C.: Four good reasons to use an interior point solver within a MIP solver. In: Operations Research Proceedings, pp. 159–164 (2018)
10. Bonami, P., Salvagnin, D., Tramontani, A.: Implementing automatic Benders decomposition in a modern MIP solver. In: IPCO 2020 Proceedings, Springer, pp. 78–90 (2020)
11. Danna, E.: Performance variability in mixed integer programming. MIP 2008 workshop in New York City (2008). http://coral.ie.lehigh.edu/~jeff/mip-2008/program.pdf
12. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Program. **102**(1), 71–90 (2005)
13. Eckstein, J., Nediak, M.: Pivot, cut, and dive: a heuristic for 0–1 mixed integer programming. J. Heuristics **13**(5), 471–503 (2007)
14. FICO: FICO Xpress reference manual (2024)
15. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. **104**(1), 91–104 (2005)
16. Fischetti, M., Lodi, A.: Local branching. Math. Program. **98**(1–3), 23–47 (2003)
17. Fischetti, M., Lodi, A.: Repairing MIP infeasibility through local branching. Comput. Oper. Res. **35**(5), 1436–1445 (2008)
18. Fischetti, M., Lodi, A.: Heuristics in mixed integer programming. Wiley, Hoboken (2011). https://doi.org/10.1002/9780470400531.eorms0376
19. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Math. Program. Comput. **1**(2–3), 201–222 (2009)
20. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-driven fix-and-propagate heuristics for mixed integer programming. Math. Program. Comput. **11**, 675–702 (2019)
21. Ghosh, S.: DINS, a MIP improvement heuristic. In: Fischetti, M., Williamson, D.P. (eds.) Integer programming and combinatorial optimization, 12th International IPCO Conference, Ithaca, NY, USA, June 25–27, 2007, proceedings, lecture notes in computer science, vol. 4513, pp. 310–323. Springer (2007)
22. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T., Salvagnin, D., Shinano, Y.: MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. Math. Program. Comput. **13**, 443–490 (2021)
23. Gondzio, J.: Presolve analysis of linear programs prior to applying an interior point method. INFORMS J. Comput. **9**, 73–91 (1997)
24. Hendel, G.: Adaptive large neighborhood search for mixed integer programming. Math. Program. Comput. **14**(1), 185–221 (2022)
25. Hoos, H.H., Stützle, T.: Local search algorithms for SAT: an empirical evaluation. J. Autom. Reason. **24**(4), 421–481 (2000)

26. IBM: ILOG CPLEX 12.10 User's Manual (2020)
27. Lin, P., Cai, S., Zou, M., Lin, J.: New characterizations and efficient local search for general integer linear programming (2023). https://arxiv.org/abs/2305.00188
28. Luteberget, B., Sartor, G.: Feasibility Jump: an LP-free Lagrangian MIP heuristic. Math. Program. Comput. **15**, 365–388 (2023)
29. Maros, I.: Computational Techniques of the Simplex Method. Kluwer Academic Publishers (2002)
30. Mexi, G., Besançon, M., Bolusani, S., Chmiela, A., Hoen, A., Gleixner, A.: Scylla: a matrix-free fix-propagate-and-project heuristic for mixed-integer optimization (2023). https://arxiv.org/abs/2307.03466
31. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: algorithms and Complexity. Dover (1982)
32. Patel, J., Chinneck, J.W.: Active-constraint variable ordering for faster feasibility of mixed integer linear programs. Math. Program. **110**, 445–474 (2007)
33. 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)
34. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J. Comput. **19**(4), 534–541 (2007)
35. Savelsbergh, M.W.P.: Preprocessing and probing for mixed integer programming problems. ORSA J. Comput. **6**, 445–454 (1994)
36. Schöning, U.: A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: 40th annual symposium on foundations of computer science, pp. 410–414. IEEE Press (1999)
37. Schulte, C.: Comparing trailing and copying for constraint programming. In: ICLP, pp. 275–289 (1999)
38. Schulte, C.: Programming constraint services. Ph.D. thesis, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany (2000)
39. Seitz, S., Alava, M., Orponen, P.: Focused local search for random 3-satisfiability. J. Stat. Mech.: Theory Exp. (2005). https://doi.org/10.1088/1742-5468/2005/06/P06006
40. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proceedings of the twelfth AAAI national conference on artificial intelligence, pp. 337–343. AAAI Press (1994)
41. Wallace, C.: ZI round, a MIP rounding heuristic. J. Heuristics **16**(5), 715–722 (2010)