A fix-propagate-repair heuristic for Mixed Integer Programming

Domenico Salvagnin, Roberto Roberti, and Matteo Fischetti

Department of Information Engineering (DEI), University of Padova
{domenico.salvagnin,roberto.roberti,matteo.fischetti}@unipd.it

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 heterogenous 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.

1 Introduction

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

$$\begin{aligned} \min cx \\ Ax \leq b \\ l \leq x \leq u \\ x_j \in \mathbb{Z}\ \forall j \in J \end{aligned}$$

where $J \subseteq I$ and I is the set of variable indices. Note that we assume all bounds to be finite and, without loss of generality, all linear constraints in $\leq$ form. Being MIP an NP-hard problem, there is a long and established literature on designing general-purpose primal heuristics [2,1], to either help exact methods like branch-and-cut, or to be run as standalone methods.

In this paper, we consider so-called constructive heuristics, i.e., primal heuristics whose purpose is to find an initial feasible solution for a given MIP. In addition, we do not assume to always have a linear programming (LP) solution available, and we do not allow for the solution of subMIPs, no matter how small.

Our starting point is the well-known fix-and-propagate diving heuristic. 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 these approaches are the *shift-and-propagate* heuristic [3] and, in the context of boolean satisfiability, the WalkSAT algorithm [14]. The WalkSAT algorithm, in particular, has been extensively studied by the SAT community, both theoretically [12] and computationally [13,9]. 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 Section 2, we describe our fix-propagate-repair framework. In Section 3, we describe the different variable/value strategies we tried, while in Section 4 we describe our solution repair approach. Extensive computational results on the MIPLIB 2017 benchmark set are given in Section 5. Future developments are discussed in Section 6.

The heuristic described in this paper is an extended version of the *fix-propagate-repair* heuristic that ranked second in the MIP 2022 Computational Competition.

2 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 of trying 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. A generic pseudocode for the proposed framework is given in Figure 1.

```
input  : A MIP instance P, limits and parameters propagate, repair,
         backtrackOnInfeas
output : A feasible solution if found, None otherwise
1  $Q = \{\emptyset\}$                          // push root node to queue
2  while $Q$ not empty and limits not reached:
3      fixing = Pop ($Q$)                  // pop a fixing from queue
4      infeas = Apply (fixing, P)        // apply fixing to current problem
       // Node processing
5      if not infeas and propagate:
6          infeas = PropagateConstraints (P, fixing)
7      if infeas and repair:
8          infeas = RepairWalk (P)
9      if infeas and backtrackOnInfeas:
10         Backtrack
11     branches = Branch (P)             // branch according to strategy
12     if branches is empty:
13         if infeas:
14             Backtrack
15         else:
16             return current domain      // feasible solution found
17     else:
18         $Q = Q \cup branches$                 // added branches to queue
19 return None
```

Fig. 1: Fix-propagate-repair scheme

The algorithm starts at line 1 by pushing an empty fixing to the queue, to serve as root node. Then we perform a standard DFS search where, at each iteration, we pop a node (represented by a variable fixing) from the queue, we process it and, if needed, branch by creating new nodes. At each node, depending on the parametrization, we can perform constraint propagation, try a repair 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 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 far as parametrization is concerned, we used the following presets:

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 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 big 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 Section 4.

3 Strategies

Once a parametrization has ben 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, i.e., the number of constraints that become more infeasible if a variable is rounded up or down. 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.

In our computational study, we implemented both static and dynamic strategies, and tried both LP-free and LP-dependent criteria. On the other hand, 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 [8], like for example *RINS* [5] or *local-branching* [6].

The variable strategies we tested are listed in Table 1, with a brief description.

While some of those strategies are quite trivial and well known (like `type` and `random`), others are less obvious and, to the best of our knowledge, novel. In particular, three strategies (`typecl`, `cliques` and `cliques2`) are based on the concept of a *clique cover*, which we now describe in detail. Formally, a clique cover C is a partition of the (binary) variables of a problem into disjoint subsets, each of which is covered by a clique constraint, i.e., at most one variable in each subset can be set to one in a feasible solution. 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

name	description
LR	left-to-right as they appear in the formulation
type	grouped by type (binaries, integer, continuous)
typecl	as `type`, but use clique cover to sort binaries
random	random shuffle within each type
locks	decreasing variable locks within each type
cliques	based on clique cover and relaxation solution
cliques2	alternative based on clique cover and relaxation solution

Table 1: Variable strategies tested in our implementation.

non-trivial clique cover involving a significant portion of the binary variables exists in most cases.

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 too expensive, we resort to simple greedy heuristics, that we now describe. First of all, all the clique constraints in the model are extracted 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 allow complemented variables in a clique; in other words, a clique is not a set of variables but rather a set of literals (variables or their complementation). This extraction phase is, in our current implementation, very simple: we just scan the linear constraints in the problem and identify those that are pure clique constraints (again, allowing complemented variables), i.e., are 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. A more advanced implementation could extract cliques from other constraints (for example, from knapsack constraints), or obtain additional cliques via probing [11]. Once the clique table is set up, we can start a greedy algorithm to construct a clique cover. We first process equality cliques (i.e., clique constraints where exactly one literal must be set to one): this is obtained by just iterating once over the equality cliques, in the order as they appear in model, and add a clique 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 binaries by:

– counting how many binaries are covered by each clique;
– assigning each binary to the largest clique covering it;
– counting how many binaries 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.

Notice that this method is LP-free, in the sense that the clique partition does not depend on a LP solution. This clique cover is used in strategy `typecl`, where variables within each clique are simply ordered as they appear in the formulation. The same clique cover is also used in strategy `cliques`, where however we exploit a reference LP solution in order to sort variables within each clique in the cover. In details, we weigh the literals in a clique using their value in a zero-objective interior point LP solution (obtained by running the barrier algorithm without crossover on the model without objective) and then use those weights to sample from a *weighted discrete random distribution*. The intuition is that we interpret the values in the reference LP solutions as probabilities as we want to have a *randomized sort* based on those. 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 binaries (again, in the order they appear in the clique).

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.

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 2: Value strategies tested in our implementation.

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 domain to compute updated minimum and maximum activities for each constraint. 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 and use the fractional part of the LP value of a variable as a probability to pick the current upper bound as preferred value: notice that this is correct not only for binary variables but also for general integer ones. Since we have two independent choices depending on which LP solution to use as reference (namely: whether to use the zero or original objective and whether to use a corepoint 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.

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. Those strategies are listed in Table 3.

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

Table 3: Overall variable/value strategies.

4 Solution Repair

Our repair strategy is based on the classical WalkSAT [14] approach for a 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 *shift*: we are allowed to shift the value of a fixed variable (binary or not) within its original domain. Of course, this is less straightforward in general: in the binary case, once a variable is chosen, there is only one possibility, while in the general integer case we might have a very large domain (and the situation is even worse for continuous variables). As for 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 violation count as a measure, we skip the variables in a constraint that do not reduce violation by shifting, 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 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. What we noticed is that the method could quickly reduce the infeasibility of the initial random solution down to a medium level of violation, 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, i.e., the current values for lower and upper bounds on all variables. From this domain we compute, for each constraint $a_i^T x \leq b_i$, the so-called minimum (m_i) and maximum (M_i) activities, i.e., the minimum and maximum values the left hand side $a_i^T x$ of the constraint can achieve by only considering the bounds on the variables. Note that the very same quantities are needed for constraint propagation as well, a fact that is exploited by our implementation: activities are incrementally kept up-to-date by both propagation and repair steps. By comparing the activity range $[m_i, M_i]$ with the constraint right hand side b_i, we can detect whether a constraint is currently violated and by how much. Thus, the violation of a given constraint is the distance between the interval $[m_i, M_i]$ and b_i.

As far as repair moves are concerned, we notice that we might need to shift not only fixed variables, but potentially 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 explained: if a non-binary variable has its domain reduced (but not fixed) as implication of constraint propagation of, say, fixing a binary variable, the repair process needs to be able to undo this tightening if the implying binary is flipped. Given that we do not want to allow domain enlargement as a repair move (as it would lead to trivial repair actions where fixings are just undone), we must allow for shifts on non-fixed variables.

With those details sorted out, the basic WalkSAT-inspired repair step is described as follows. At each step:

– We pick a violated linear constraint uniformly at random.
– We build a list of candidate variables that can reduce the violation of the current constraint if shifted. The list also includes non-binary variables, although the logic is a bit different: while for binaries we just check whether a flip would reduce violation by using a sign argument, for non-binary variables we compute the minimal shift that would make the constraint satisfied, round it if the variable is general integer, and then clip it so that the variable is still within its global bounds. If the resulting shift is nonzero, then the variable is added to the list of candidates.
– For each candidate, we compute the *damage* of the shift, i.e., we sum up the increases in violation over all constraints for which violation actually increases (as in the original WalkSAT, we ignore improvements in violation).
– Finally, we apply the usual WalkSAT logic: 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 $1 - p$.

The noise parameter is set to $p = 0.75$ in our implementation. As we call the repair function very frequently, we use a small iteration limit of 200 for each call. Finally, we extended this basic logic with two simple tricks: we maintain a short tabu list of the last 3 shifts in order to avoid short cycles (whose probability is quite large when there are very few violated constraints), and we soft restart from the best state every 10 shifts (this is to avoid the repair process to diverge to states from which it is very time consuming to recover).

4.1 Repair search

The repair scheme described so far works rather well in practice, but it is unsatisfactory from 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 examples, swapping two binaries 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 taking into account that the repair routine is potentially called over and over at different nodes of the DFS search, and thus has strict limits. Adding the short tabu list and soft restarts helped in reducing those shortcomings, but their effect is 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 an 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. Notice that if we detect infeasibility, we can backtrack in the repair space and avoid wasting time in a hopeless subtree.

This approach has several advantages (and some disadvantages). On the positive side, we no longer need tabu lists: systematic search takes care of avoiding cycles. Also, we can exploit constraint propagation and have a higher chance of doing "complex" moves, like a swap of binaries: 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 always 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 domain of partial solution we are trying to repair with the current domain in the repair tree. In the rest of the section, we will provide details about those steps.

Let us start with synchronization. At the beginning of the repair process, the root node of the repair tree is initialized with the global domain. Then, as we apply repair moves and their implications, the domain gets tightened: clearly those changes need to be applied in some way to the domain describing the current (infeasible) node in the main tree. Therefore, we need to apply changes from the repair domain, say D_r, to the current infeasible domain D. As usual, the logic is much simpler for binary variables than for non-binaries. 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 shift the interval of D until it touches the interval in D_r, and then fix the variable to the common endpoint.

A simple 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 3 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 $D \subseteq D_r$.

Finally, let us describe the logic to turn a repair move 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 is always a shift s and the shifted interval $[a, b]$ in D is always contained in the interval $[c, d]$ in D_r, by construction. 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 $x_j \leq a \vee x_j \geq a$.

5 Computational results

We implemented our framework in C++ and used IBM ILOG CPLEX 12.10 [10] 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 [7] benchmark set, using different random seeds to limit the effect of performance variability [4]. Each method was run on each instance-seed combination with a time limit of 10 minutes, 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 $[-100000, +100000]$: this is in general not a valid constraint, but it is acceptable for heuristic purposes, and allows the constraint propagation routines to always assume finite bounds.

– 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 where all variables are set to 1 is often found first.

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 reported 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 second).

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

Table 4: Preliminary strategy study on `dfs`.

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 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[1], 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 the 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 400s. 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.

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 in the previous experiment. A graphical representation of the same data is available in Figure 2.

According to the table, solution repair significantly improves the success rate of the different strategies: `repdfs` 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 `repdfs`. We conclude from those experiments that solution repair is a beneficial addition to fix-and-propagate, and that the most successful methods, `repdfs` 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.

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

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
	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

Table 5: Preliminary strategy study.

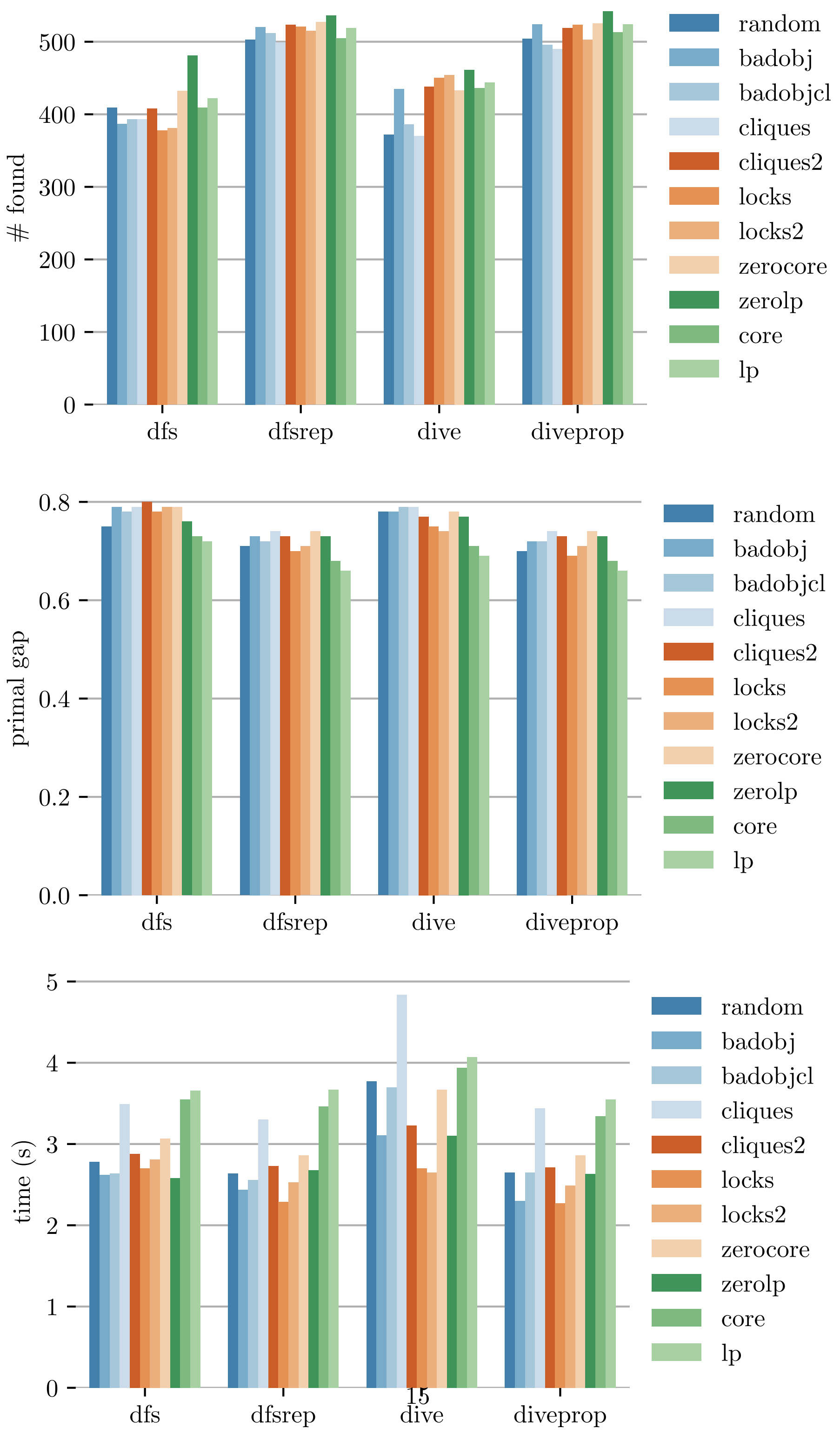

Fig. 2: Comparison between different strategies and parametrizations.

This is confirmed by the success rate of the virtual best solver which, at 606, is still significantly higher than any method.

5.1 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 super-linear 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:

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[2], but we are quite satisfied with its current performance and we did 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 fixpoint 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`, `dfsref-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.

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

– 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 · 5), up from the 75%(542/240 · 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.

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 6: Aggregated portfolio results.

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 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. On the other hand, it is worth noting that the current tuning is geared for a standalone usage of the method: if implemented inside a MIP solver to complement regular branch-and-cut, we would probably go for a different setting, where LP-free methods are executed concurrently to the first

LP, while the other methods are only executed after the root relaxation is solved (as, at that point, we would have LP information for free).

5.2 Impact of repair search

Finally, we can evaluate the effectiveness of the repair search described in Section 4.1 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).

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

Table 7: Repair search vs. repair walk in portfolio approach.

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.

		repair-search	
		success	not found
repair-walk	success	927	18
	not found	42	208

Table 8: Repair search vs. repair walk contingency table.

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.

6 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 seems to be quite effective in finding feasible solutions on a heterogeneous testset like the MIPLIB 2017 benchmark set, and a preliminary version of the approach ranked second in the MIP 2022 Computational Competition. 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.

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
2. Berthold, T.: Primal Heuristics for Mixed Integer Programs. Master's thesis, Technische Universität Berlin (2006)
3. Berthold, T., Hendel, G.: Shift-and-propagate. Journal of Heuristics **21**(1), 73–106 (2015)
4. Danna, E.: Performance variability in mixed integer programming. MIP 2008 workshop in New York City. http://coral.ie.lehigh.edu/~jeff/mip-2008/program.pdf (2008)
5. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve MIP solutions. Mathematical Programming **102**(1), 71–90 (2005)
6. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1–3), 23–47 (2003)
7. 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. Mathematical Programming Computation **13**, 443–490 (2021)
8. Hendel, G.: Adaptive large neighborhood search for mixed integer programming. Mathematical Programming Computation **14**(1), 185–221 (2022)
9. Hoos, H.H., Stützle, T.: Local search algorithms for SAT: An empirical evaluation. Journal of Automated Reasoning **24**(4), 421–481 (2000)
10. IBM: ILOG CPLEX 12.10 User's Manual (2020)
11. Savelsbergh, M.W.P.: Preprocessing and probing for mixed integer programming problems. ORSA Journal on Computing **6**, 445–454 (1994)
12. Schöning, U.: A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: FOCS. pp. 410–414. IEEE Press (1999)
13. Seitz, S., Alava, M., Orponen, P.: Focused local search for random 3-satisfiability. CoRR (2005)
14. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: AAAI. pp. 337–343 (1994)