Local search with adaptive weighting as a MIP primal heuristic

Bjørnar Luteberget, Giorgio Sartor, Volker Hoffmann, and Oddvar Kloster

SINTEF Digital, Oslo, Norway
`bjornar.luteberget@sintef.no`

Abstract. This report describes our entry to the MIP 2022 Computational Competition[1]. We present a local search algorithm that optimizes a weighted sum of currently unsatisfied constraints, and the original problem objective. The algorithm finds feasible solutions to 13 of 19 of the competition's public instances. Performance and quality results for the competition's public instances are given in Table 1 on page 7.

1 Introduction

This report describes an LP-free primal heuristic for quickly and efficiently finding solutions to mixed-integer linear programming (MIP) problems. The method is essentially incomplete: it does not necessarily produce solutions to all feasible problems, the solutions it produces are not in general optimal, and it cannot detect infeasibility. Still, algorithms that quickly produce feasible solutions can be valuable as a subroutine of a complete branch-and-bound MIP solver, either because the user might be content with possibly sub-optimal solutions that are produced as quickly as possible, or because the branch-and-bound search itself can become faster when a (good-quality) feasible solution is known (see, for example, [1, 2, 4]).

Many of the MIP instances provided so far for the competition (the public instances) contain a lot variables and constraints, but they are mostly sparse. The instances also have predominantly integer variables with small domains (especially after presolving), and many of them are binary (0-1). Motivated by these features, we have investigated techniques from constraint programming and local search on finite domains.

Local search. We have developed a local search algorithm that starts from any (potentially infeasible) solution, and measures how far away the current incumbent solution is from satisfying the constraints (i.e., from feasibility). We maintain for each variable a small set of potential new values (which are different from the current incumbent) and we associate to each of these values a score

[1] A previous version of this report included also a SAT-based algorithm, in combination with the local search. The SAT-based algorithm was removed after conferring with the competition organizers to comply with the competition's rule stating that no external solvers with branching were allowed.

that is proportional to the potential reduction in the total constraints violation. At each iteration, we assign a new value to a promising variable and we update the scores of the variables that appear in the same constraints. If no value exists that improves the current sum of constraint violations, we have reached a local minimum. We increase the weight of the currently unsatisfied constraints and update the scores of the values of the variables involved. After the first feasible solution has been found, the original objective of the MIP starts to contribute as well in the score of the potential values of each variable. This algorithm is based on the pseudo-boolean optimization algorithm presented in [5], but it has been adapted to support generic mixed-integer programs by considering more potential values that a variable can change to. In the binary case, there is only one value for a variable that is different from the current solution's value, but for general integer and continuous variables, there can be any number of values to choose. We handle this by allowing only three choices for a variable x_i:

- Up: move the variable towards its upper bound by the step size z_i.

$$x_i \leftarrow \min(x_i + z_i, u_i)$$

  If the variable is already at the upper bound, this choice is not valid. The step size z_i is defined for each variable x_i separately, and it is initially set to the size of the domain $z_i = u_i - l_i$. When the value has changed back and forth between the same values more than twice, we decrease the step size by setting $z_i \leftarrow z_i/2$.
- Down: similarly to Up, move the variable towards its lower bound:

$$x_i \leftarrow \max(x_i - z_i, l_i)$$

- Jump: the jump value is the value that most reduces the total constraint violation (see Section 2).

For any solution and any variable, at least one of Up or Down will be applicable, and Jump will always be applicable. See section 3 for details.

We have combined the local search with MIP presolving and linear programming (LP) solving. We use the Gurobi MIP solver for presolving instances. Depending on parameters, we also remove continuous variables from the problem by replacing them with their lower bounds (in less-than constraints), and then after finding an integer solution we substitute the integers into the original MIP instance and solve the resulting LP instance with Gurobi (similarly to the approach in [3]).

2 The Jump

The Jump value used in the local search algorithm was motivated by the *best shift* of the `Shift-and-Propagate` heuristic [3], but extended to consider partial feasibility improvements. In [3], given an (infeasible) incumbent solution $\tilde{x}$, the best shift for a variable x_i is the value $\tilde{s}_i$ such that $\tilde{s}_i + \tilde{x}_i$ reduces the largest number of violated constraints (assuming the rest of the incumbent stays the same). The intrinsic problem of considering only when constraints switch between being violated and satisfied is the loss of information associated with constraints such as:

$$x_1 + \cdots + x_n \geq b, \qquad x_i \in \{0,1\},\ n, b > 1,\ \tilde{x}_i = 0, \tag{1}$$

where shifting any of the variables by +1 would not be enough to satisfy the constraint. An alternative measure that prevents this drawback is simply the one that considers how far away is each constraint from being satisfied (see, for example, [5]). In other words, we consider a measure that is proportional to the total constraint violation. For example, the constraint in (1) would contribute b when $\tilde{x}_i = 0$, and shifting any of the variables by +1 would reduce its contribution to $b - 1$. Therefore, the jump value for a variable x_i is defined as the value (different from the current incumbent) that most reduces the total constraint violation.

Note that the violation function of each constraint is convex, which means that the sum of these functions is also convex. Therefore, to find the jump value of a variable one could simply scan through its values, starting from the lower bound, and stopping as soon as the slope becomes non-negative or we reach the upper bound.

Although the jump value for a variable x_i depends on the incumbent values of all the variables that appear in the same constraints as x_i, we update the jump value for a variable only when we change its incumbent value. Preventing this propagation effect is important to keep the computational complexity low. In other words, while the jump value is not necessarily the *best* jump value at any point in time, we trade exactness for performance.

Example. Consider the pair of constraints $x_1 + x_2 = 2$ and $x_2 + x_3 \geq 3$, where $x_1, x_2, x_3 \in \{0, 1, 2, 3, 4\}$ and the current incumbent is $\tilde{x}_1 = 2, \tilde{x}_2 = 0, \tilde{x}_3 = 0$. Figure 1 shows the violation function for variable x_2 in each constraint, while Figure 2 shows the sum of those violation functions. In this case, the jump value could be either 2 or 3, since they are the values that most reduce the total constraint violation. The algorithm breaks these ties by simply choosing the first value.

3 Local search algorithm

The local search maintains a current solution $\vec{x}$, which can be infeasible. We define the infeasibility cost $C^*(\vec{x})$ as the sum of infeasibility cost over each constraint i, i.e. $C^*(\vec{x}) = \sum_i w_i c_i(\vec{x})$, where w_i are a weight for each constraint initially set to 1. For less-than constraints $\sum a_{ij} x_j \leq b_i$ we have

$$c_i(\vec{x}) = \max\Big(0,\ b_i - \sum a_{ij} \vec{x}(j)\Big)$$

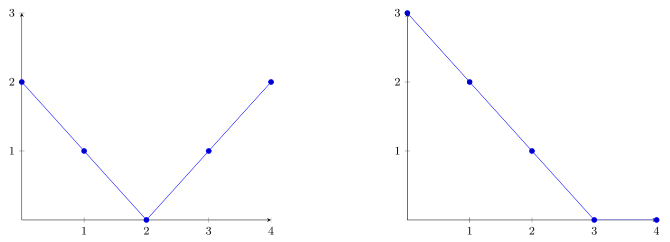

Fig. 1. Constraint violation function for variable x_2 in each constraint.

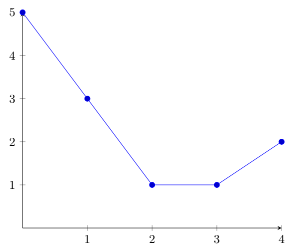

Fig. 2. The sum of the constraint violation functions for variable x_2.

The infeasibility cost is combined with the original MIP objective $\sum_j o_j x_j$, weighted by a factor q, so the objective of the local search is:

$$C(\vec{x}) = C^*(\vec{x}) + q \sum_j o_j x_j .$$

Initially, $q = 0$, so we are only searching for feasible solutions. We define a score for every *choice* $\rho \in X \times D$, a choice being the combination of a variable $x_i \in X$ and a direction $d_i \in D$, where directions are either of $D = \{\textsc{Up}, \textsc{Down}, \textsc{Jump}\}$ described above. The choice defines a potential new value $v(x_i, d_i)$. The score $s(\vec{x}, \rho)$ of the choice $\rho = (x_i, d_i)$ at the current solution $\vec{x}$ is defined by the difference in the cost achieved by assigning $v(x_i, d_i)$ to the j^{th} component of $\vec{x}$:

$$s(\vec{x}, \rho) = C(\vec{x}_{[x_i \leftarrow v(x_i, d_i)]}) - C(\vec{x})$$

The score of all choices is maintained by the algorithm, along with a set that contains all variables with a positive score. At every iteration, a selection of choices is sampled from the set of positive scores and the best one is chosen. We found that maintaining the ranking of all variables to find the highest scoring one was slowing the algorithm down, and we did not find that increasing the sample size has much impact on the performance. We use a sample size of 100.

If a choice (x_i, d_i) with positive score exists, we update the current solution $\vec{x}$ to have $\vec{x}(i) = v(x_i, d_i)$. We also update x_i's Jump value and the scores for all choices of the neighboring variables (variables that appear in the same constraints as x_i). If no choice with positive score exists, we are currently in a local minimum. In this case, we increase the weights of the constraints that

Algorithm 1 Local search with adaptive weights

```
    Input a MIP instance $A\vec{x} \leq \vec{b}$, with objective $\sum_j o_j x_j$, starting solution $\vec{x}$,
        a random seed, and relax_continuous
    Output: one or more feasible solutions to the MIP, or NULL.
 1: best_feasibility ← $\infty$, best_objective ← $\infty$, best_x ← NULL
 2: last_improvement ← 0
 3: if relax_continuous then
 4:     $(A, \vec{b}) \leftarrow$ RelaxContinuous($A$, $\vec{b}$)
 5: while CurrentTime() − last_improvement < limit do
 6:     if number of unsatisfied constraints < best_feasibility then
 7:         best_feasibility ← number of unsatisfied constraints
 8:         last_improvement ← CurrentTime(),
 9:     if all constraints satisfied and $\sum_j o_j x_j$ < best_objective then
10:         best_x ← $\vec{x}$
11:         best_objective ← $\sum_j o_j x_j$
12:         last_improvement ← CurrentTime(),
13:     if any choice $(x_i, d_i)$ has positive score then
14:         $x_i \leftarrow v(x_i, d_i)$
15:     else
16:         for unsatisfied constraints $i$ do
17:             $w_i \leftarrow w_i + 1$
18:         if all constraints are satisfied then
19:             $q \leftarrow q + 1$
20:         $x_i \leftarrow v(x_i, d_i)$ if any choice has positive score (otherwise a random choice).
21: return best_x
```

are currently unsatisfied and update the scores of the affected variables ($w_i \leftarrow w_i + 1$). This approach is known in the literature as a *guided local search*, and it is a way to avoid getting stuck in local minima by discovering which constraints are harder to satisfy and should be prioritized.

Every time we discover a feasible solution and we are in a local minimum, we update the weight of the original objective of the MIP ($q \leftarrow q+1$), hopefully steering the search towards solutions with better objective value.

Before starting the algorithm, if `relax_continuous` is set to true, we relax the continuous variables' terms by replacing each occurrence $a_i x_i$ of a continuous variable x_i by the lower bounds of $a_i x_i$, i.e. $a_i l_i$ if $a_i > 0$ and $a_i u_i$ if $a_i < 0$. To be deterministic, the CurrentTime function does not actually measure time, but measures the amount of work done by the algorithm.

The computational complexity of each iteration is dominated by the updating of the scores:

- Making the choice of variable and direction takes constant time because it consists of sampling a constant amount of variable scores.
- Updating the constraint weights w_i requires iterating over all currently unsatisfied constraints (in the worst case, all constraints) and over all variables appearing in those constraints. In total, the worst case scales with the number of non-zero coefficients in the problem.

– Updating the objective weight q scales with the number of non-zero objective coefficients.
– Setting the value of a variable requires updating the jump value and the scores of all the neighboring variables. This requires iterating over all the constraint the that the variable appears in and all the variables appearing in those constraints. In total, the worst case scales with the number of non-zero coefficients in the problem.

4 Competition entry solver

We run 8 threads in parallel for diversification, each with a different set of parameters. The parameters are the random seed for the local search, whether to relax continuous variables or not, and whether to presolve or not, see Algorithm 2. We suspect that some modifications made by the Gurobi presolve algorithm are geared towards branch-and-bound LP methods, for example relaxing of integrality requirements. These assumptions might not hold for our LP-free solver, and we see that usually the presolve is very beneficial, while sometimes it is not. Going forward, we will investigate whether bounds propagation only will give the same benefits as Gurobi's presolve without the unwanted effects.

Algorithm 2 MIP heuristic solver with local search, presolving and LP solving

```
     Input a MIP instance $A\vec{x} \leq \vec{b}$
     Output: a feasible solution to the MIP, or NULL.
 1: for thread_id $\in [1, $n_threads$]$ do
 2:     encoding_aggressiveness $\leftarrow$ thread_id/n_threads
 3:     seed $\leftarrow$ thread_id
 4:     relax_continuous $\leftarrow$ thread_id%2 == 0
 5:     presolve $\leftarrow$ thread_id%2 == 1
 6:     if presolve then
 7:         $A,\vec{b} \leftarrow$ PRESOLVE($A,\vec{b}$)
 8:     $\vec{x} \leftarrow$ LOCALSEARCH($A,\vec{b}$,start=$\vec{x}$, seed, relax_continuous)
 9:     if relax_continuous then
10:         $\vec{x} \leftarrow$ SOLVELP($A,\vec{b}$,integer_values=$\vec{x}$)
11:     OUTPUT($\vec{x}$)                                   ▷ Ignored if infeasible.
```

5 Results

Table 1 shows the results[2] of our Rust implementation running on an AMD Threadripper 3990x CPU. Presolve time is included in the running time numbers. Additionally, we tested the same MIPLIB instances used in [3], our solver is able to find feasible solutions for 114/168 instances.

Instance	First sol. time (s)	Last sol. time (s)	Termination time (s)	First sol. objective	Last sol. objective
`academictimetablesmall.mps`	1.07	1.2	15.39	10120	3586
`comp07-2idx.mps`	0.04	0.04	10.03	1569	1569
`cryptanalysiskb128n5obj16.mps`	-	-	22.11	-	-
`eil33-2.mps`	0.02	0.02	9.33	1642.57	1642.57
`highschool1-aigio.mps`	2.75	2.88	12.45	14924	12956
`mcsched.mps`	0.14	0.28	6.88	476402	474281
`neos-1354092.mps`	0.9	0.9	10.94	400591	400591
`neos-3024952-loue.mps`	-	-	17.30	-	-
`neos-3555904-turama.mps`	2.83	2.83	27.78	-34.7	-34.7
`neos-4532248-waihi.mps`	0.95	9.31	13.94	643.2	581.76
`neos-4722843-widden.mps`	-	-	19.35	-	-
`ns1760995.mps`	1.86	21.97	28.82	-140.45	-150.81
`ns1952667.mps`	-	-	14.59	-	-
`peg-solitaire-a3.mps`	-	-	14.24	-	-
`qap10.mps`	0.22	0.25	12.69	534	440
`rail01.mps`	-	-	19.20	-	-
`rococoC10-001000.mps`	21.27	21.27	27.77	42727	42727
`seymour.mps`	0.01	0.29	5.27	491	475
`supportcase10.mps`	1.05	1.05	8.78	17	17

Table 1. Results for the MIP2022 competition's 19 public instances using the local search in Algorithm 1. The first column contains the instance name. The next three columns describe the time the algorithm takes: the time until the first solution was found, the time until the last solution was found, and the time until the algorithm terminated. The last two columns show the objective value of the first and the last solutions. Cells without numbers indicate that the solver did not find any feasible solutions for this problem instance.

[2] A previous version of this report included the results of an algorithm that combined local search with a SAT-based method. The SAT-based method has been removed and this table shows the results of the local search only.

References

1. Tobias Achterberg. *Constraint integer programming*. PhD thesis, Berlin Institute of Technology, 2007.
2. Timo Berthold. Primal heuristics for mixed integer programs. Diplomarbeit, Zuse Institute Berlin (ZIB), 2006.
3. Timo Berthold and Gregor Hendel. Shift-and-propagate. *J. Heuristics*, 21(1):73–106, 2015.
4. Matteo Fischetti, Giorgio Sartor, and Arrigo Zanette. Mip-and-refine matheuristic for smart grid energy management. *International Transactions in Operational Research*, 22(1):49–59, 2015.
5. Zhendong Lei, Shaowei Cai, Chuan Luo, and Holger H. Hoos. Efficient local search for pseudo boolean optimization. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 332–348. Springer, 2021.