Feasibility Jump: an LP-free Lagrangian MIP heuristic

Bjørnar Luteberget Giorgio Sartor

November 2022

Abstract

We present Feasibility Jump (FJ), a primal heuristic for mixed-integer linear programs (MIP) using stochastic guided local search over a Lagrangian relaxation. 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. It does, however, very quickly produce feasible solutions to many hard MIP problem instances. Starting from any variable assignment, Feasibility Jump repeatedly selects a variable and sets its value to minimize a weighted sum of constraint violations. These weights (which correspond to the Lagrangian multipliers) are adjusted for constraints that remain violated in local minima. Contrary to many other primal heuristics, Feasibility Jump does not require a solution of the continuous relaxation, which can be time-consuming for some problems. We compare FJ against FICO Xpress Solver 8.14 and we show that this heuristic is effective on a range of problems from the MIPLIB 2017 benchmark set [17], significantly improving the average time to first feasible solution. We also show that providing these quick solutions to Xpress produces a modest reduction in the average time to optimal solution in the same benchmark set.

Our entry based on FJ to the MIP 2022 Computational Competition (which challenged participants to write LP-free MIP heuristics) won 1st place. Moreover, an implementation of Feasbility Jump now runs by default on FICO Xpress Solver 9.0, where similar results to the ones presented here could be observed [21].

1 Introduction

In mathematical programming, heuristic algorithms have always been an essential tool to quickly find good feasible solutions. Not only they are used as standalone specialized algorithms, but they are also tightly integrated in all state-of-the-art MIP solvers. The introduction of primal heuristics was one of the four key ideas (together with cutting planes, branching strategies, and preprocessing) that helped MIP solvers become so effective in the last two decades [20]. But as MIP solvers got better, users wanted to solve increasingly hard optimization problems, making sure the task of finding proven optimal solutions (or even just feasible solutions) would still be challenging. This is particularly true for mixed-integer linear programs, since it is, in general, not only NP-hard to find an optimal solution, but also NP-hard to find a feasible one [26]. This work will focus on the development of a general purpose primal heuristic, that is a heuristic algorithm for finding feasible solutions to generic MIP problems, allowing it to be used as a standalone heuristic or to be integrated into a generic MIP solver.

Most of the general purpose primal heuristics employed by existing MIP solvers are applied only after solving the root node, such as *local branching* [12], *pivot and shift* [5], *feasibility* *pump* [11], *RINS* [10], and *RENS* [7]. Some more recent heuristics that exploit information obtained from the LP relaxation of a MIP problem include conflict-driven diving heuristics [25] and even ML-based heuristics, such as [24] and [23].

But very few heuristics have been developed so as to be applied before the root node (i.e., pre-root heuristics), and they can generally be subdivided into *propagation methods* or *relaxation methods*. Propagation methods mostly involve constructive heuristics, where iterative greedy decisions on the value of variables are propagated to the rest of the problem. Berthold and Hendel's [8] *shift-and-propagate* heuristic follows exactly this approach. It alternately fixes one variable at a time to a promising value, and propagates this partial assignment to the rest of the problem. The order in which the variables are fixed is decided from the beginning and depends on the number of violated rows in the initial assignment. The promising value of a certain variable, called *best-shift*, is the one that minimizes the total number of violated constraints. More recently, a similar structure-driven approach was proposed by Gamrath et al. [16], where the order in which the variables are fixed is based on information extracted either from the clique table or the variable bound graph, both of which are usually computed in the presolve phase of state-of-the-art MIP solvers.

Relaxation methods follow instead a very different approach, usually employing local search methods in a relaxed version of the problem. This is also the approach recently proposed by Lei et al. [19] in the context of pseudo-Boolean optimization, that is when all variables have binary domains (also called 0-1 integer programs). The idea is to consider the Lagrangian relaxation of a pseudo-Boolean problem and iteratively flip the value of the variable that most reduces the total weighted constraint violation. The weighting of the constraints (which corresponds to the Lagrangian multipliers) is updated whenever a local minimum is reached, that is, when there is no variable that improves the current weighted total constraint violation and the current assignment is still infeasible. Once a feasible solution has been found, the original objective function of the problem can be also taken into account. Despite the large amount of recent literature about Lagrangian methods for MIP problems, most of the proposed heuristics are very problem specific, and "an improved Lagrangian technology would be a useful tool in the bag of tricks available for solving difficult optimization problems" [15].

In this paper we describe Feasibility Jump (FJ), a general-purpose pre-root primal heuristic for mixed-integer linear programming problems that belongs to the category of relaxation methods. It can be used either as a standalone heuristic or tightly integrated into a more sophisticated MIP solver. It extends the Lagrangian approach described in [19] to deal with both general integer and continuous variables. In particular, when considering a variable separately and assuming all other variables are fixed, we can efficiently find a new optimal value for the variable. We say that values of promising variables *jump* towards assignments with smaller constraint violations. The jump values are computed by extending and improving the concept of *best-shift* described in [8]. These values are updated in a lazy fashion, only after a variable *jumps* and only for that variable.

Being an incomplete algorithm, Feasibility Jump does not have any guarantee in terms of finding a feasible solution or proving infeasibility, for example. The trade off is speed. With a recent laptop and on sparse problems (almost independently of their size), FJ can hit 1 million jumps per second. Algorithms that quickly produce feasible solutions can be extremely 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, [3, 6, 14]).

The algorithm was originally developed for the MIP 2022 Workshop's Computational Competition [1], where it won 1st place. It competed on a set of (hidden) MIP instances as a standalone heuristic. In this paper, we go one step further by integrating FJ with Xpress, and showing how it can improve both the time to first feasible solution and the time to optimal solution on instances from the MIPLIB 2017 benchmark set [17]. A C++ open-source implementation of Feasibility Jump together with the Xpress integration are available at `https://github.com/sintef/feasibilityjump`.

This paper provides the following contributions:

- a fast and effective primal heuristic for MIP problems;
- a high-performance open-source C++ implementation;
- results from a tight integration with the FICO Xpress solver.

2 Feasibility Jump

A mixed-integer linear program (MIP) is an optimization problem of the form

$$\begin{array}{rlr} \text{minimize} & \sum_{j\in N} c_j x_j & \\ \text{subject to} & \sum_{j\in N} a_{ij}x_j \le b_i & i \in M, \\ & l_j \le x_j \le u_j & j \in N, \\ & x_j \in \mathbb{Z} & j \in I, \end{array} \tag{1}$$

where $N = \{1, \ldots, n\}$, $M = \{1, \ldots, m\}$, $x \in \mathbb{R}^n$, l_j and u_j are variable bounds, and $I \subseteq N$ are indices of variables that are constrained to take only integer values. Any specific vector $\bar{x} \in \mathbb{R}^n$ is an *assignment* to the variables. An assignment that satisfies the linear constraints, bounds, and integrality, is a *feasible solution.* A feasible solution that minimizes the objective is an *optimal solution.*

Local search algorithms are heuristic optimization algorithms that work by considering a feasible solution $\bar{x}$ and examining a set of other neighboring solutions $\mathcal{N}(\bar{x})$ (i.e., solutions that are close to $\bar{x}$ according to some predefined distance measure). If it finds a new feasible solution $\bar{x}' \in \mathcal{N}(\bar{x})$ with a better objective value, then it sets $\bar{x} \leftarrow \bar{x}'$, and the process repeats as long as such an improving solution can be found. When no such $\bar{x}'$ exists, the process has reached a *local minimum.* It is not possible, in general, to know if the local minimum is also a global minimum. In practice, local search algorithms work well for many optimization problems even though their theoretical guarantees tend to be weak [2].

The objective of primal heuristics is to find feasible solutions to problems such as (1). In the context of a local search algorithm, this requires relaxing some of the constraints, so that it becomes possible to start from a solution that is feasible for all but the relaxed constraints. Depending on which constraints get relaxed, different algorithms or techniques may emerge. For example, the well-known *Feasibility Pump* (for which we owe the inspiration of our heuristic's name) is a local search algorithm that relaxes the integrality constraints of 1 and tries to move towards solutions in which these constraints are less and less violated. Our approach is different. We relax all but the variable bounds and the integrality constraints, penalizing the relaxed ones (when violated) in the objective function. This technique is usually known as *Lagrangian relaxation.*

In the next sections, we describe all the pieces that contribute to Feasibility Jump. We first introduce a Lagrangian relaxation of the MIP problem (Section 2.1) and we explain how to compute promising values that variables can use to heuristically "jump" towards local minima of the corresponding Lagrangian function (Section 2.2). We then describe how these values can be used to efficiently define new neighbourhoods (Section 2.3) and how to traverse them (Section 2.4). Finally, Section 2.5 summarizes the entire algorithm.

2.1 Relaxing the linear constraints

Based on the well-known Lagrangian relaxation of a MIP problem, we define our relaxed MIP problem as:

$$\begin{array}{rll} \text{minimize} & O^q(x) + F^w(x) & \\ \text{subject to} & l_j \le x_j \le u_j & j \in N, \\ & x_j \in \mathbb{Z} & j \in I, \end{array} \tag{2}$$

where $O^q(x) = q \sum_{j \in N} c_j x_j$ is the original objective weighted by q, and $F^w(x)$ is the total infeasibility penalty of an incumbent solution x, defined as the sum of the infeasibility penalties computed for each linear constraint

$$F^w(x) = \sum_{i \in M} w_i f_i(x), \tag{3}$$

where w_i is a weight associated to each constraint and the infeasibility penalty $f_i(\cdot)$, $i \in M$, for a single linear constraint $\sum_{j \in N} a_{ij} x_j \le b_i$ is

$$f_i(x) = \max \left\{ 0, \sum_{j \in N} a_{ij} x_j - b_i \right\}. \tag{4}$$

The factor q is introduced to adjust the relative importance of the feasibility versus the original MIP objective. If we set $q = 0$, the relaxed problem's objective becomes exclusively to achieve feasibility in the original problem. We use the max function in $f_i(\cdot)$ (as opposed to the classical Lagrangian full penalty $\sum_{j \in N} a_{ij} x_j - b_i$) because we are more interested in solutions that live at the edge of the feasible region, rather than at its center. On one hand, this could yield feasible solutions with better objective value and, on the other hand, it can be beneficial for feasibility heuristics based on local search (see, for example, [11]).

The first surveys on Lagrangian techniques for discrete optimization started appearing already in the 1970s [22], but the basic idea did not change since then: minimize the Lagrangian function $O^q(x) + F^w(x)$ for fixed q and w, produce modified weights $\bar{q}$ and $\bar{w}$ (usually by increasing their value for violated constraints and reducing it for satisfied ones), and repeat. Unfortunately (but not unexpectedly), no theoretical guarantees exist for the this method to converge to an integer feasible solution [9].

As we will see in the following sections, Feasibility Jump follows a very similar framework. However, we do not want to minimize the Lagrangian function exactly, but rather look for a local minimum in neighbourhoods where we are allowed to only change the value of one variable at a time. The next section describes how to compute a promising value to which each single variable could jump to.

2.2 The *jump* value

Given the current variable assignment $\bar{x}$ and considering a single variable x_j, we would like to find the value that solves (2) when all variables x_k, $k \in N$, $k \neq j$, are fixed to their current value $\bar{x}_k$. In essence, this is the value of x_j that minimizes the total constraint violation (plus the original objective penalty) given that all other variables are fixed to the current incumbent. But there is a caveat. We want this value to be different from the current $\bar{x}_j$. This is common in local search methods, where one would want for each variable a non-empty neighbourhood. However, if x_j is not restricted to take only integer values, then one could not simply add the additional constraint $x_j \neq \bar{x}_j$.

Notice that when all variables but x_j are fixed, then each $f_i(x_j)$ in (4) measures the constraint violation of an expression of the form $a_{ij}x_j \leq d_i$, where $d_i = b_i - \sum_{k \neq j} a_{ik}\bar{x}_k$. If x_j is integer and d_i is fractional, then it only makes sense to consider $a_{ij}x_j \leq \lfloor d_i \rfloor$ if $a_{ij} > 0$, or $a_{ij}x_j \leq \lceil d_i \rceil$ if $a_{ij} < 0$ [18].

In general, given a current variable assignment $\bar{x}$ and a variable x_j, for each constraint violation function $f_i(x_j |\ x_k = \bar{x}_k,\ k \neq j)$ in which $a_{ij} \neq 0$, we define the critical value, $t_{ij}(\bar{x}_{k \neq j})$, as follows:

$$t_{ij}(\bar{x}_{k\neq j}) = \begin{cases} \lfloor r \rfloor & a_{ij} > 0 \\ \\ \lceil r \rceil & a_{ij} < 0 \end{cases} \qquad r = \frac{1}{a_{ij}}(b_i - \sum_{k \neq j} a_{ik}\bar{x}_k),$$

where $\bar{x}_{k\neq j}$ is short for $x_k = \bar{x}_k$, $k \neq j$. In other words, the critical value $t_{ij}(\bar{x}_{k\neq j})$ is the greatest (resp. smallest, if the coefficient is negative) value that variable x_j can take before constraints i becomes violated, when all the other variables are fixed to $\bar{x}$. For greater (resp. smaller) values of x_j, the penalty associated with the violation of constraint i increases proportionally to its corresponding weight w_i. For values of x_j smaller (resp. greater) than $t_{ij}(\bar{x}_{k\neq j})$, the penalty is zero. For a given $\bar{x}$, this defines a piecewise-linear convex function $g_{ij}(t|\bar{x}_{k\neq j})$ such that:

$$g_{ij}(t|\bar{x}_{k\neq j}) = \begin{cases} \max\left\{0, w_i\big(t - t_{ij}(\bar{x}_{k\neq j})\big)\right\} & a_{ij} > 0 \\ \\ \max\left\{0, -w_i\big(t - t_{ij}(\bar{x}_{k\neq j})\big)\right\} & a_{ij} < 0, \end{cases} \tag{5}$$

where $t \in \mathbb{R}$ and $j \in N$. For ease of explanation, we ignored the original objective function in these considerations. This can be taken into account simply by considering the additional constraint $c^t x < \infty$ with a weight equal to q. In the rest of the section, we assume the objective function to be part of the set of constraint indices M.

We are now ready to define the promising value each variable is allowed to jump to.

Definition 1 *Given a feasible solution $\bar{x}$ for problem* (2)*, we define the* jump *value of variable x_j as the* feasible *value of x_j (different from $\bar{x}_j$) that minimizes the sum of the constraint violation penalties,*

$$G_j(t|\bar{x}_{k\neq j}) = \sum_{i \in M:\, a_{ij} \neq 0} g_{ij}(t|\bar{x}_{k\neq j}).$$

Then we have:

$$\texttt{Jump}_j(\bar{x}_{k\neq j}) = \underset{t \in [l_j, u_j], t \neq \bar{x}_j}{\arg\min}\ G_j(t|\bar{x}_{k\neq j}). \tag{6}$$

By looking at (5), it is easy to see that the *jump* value of a variable will be exactly equal to one of the values in the set of critical values, plus its lower and upper bounds:

$$T_j(\bar{x}) = l_j \cup u_j \cup \bigcup_{i \in M} t_{ij}(\bar{x}_{k \neq j}).$$

Therefore, it is always possible to find a t that is different than $\bar{x}_j$, even in the continuous case. A simple and efficient algorithm to compute the *jump* value of variable x_j given the current incumbent $\bar{x}$ is described in Algorithm (1). We start by computing $T_j(\bar{x}_{k\neq j})$ and sorting its values in ascending order. Then, for every $t \in T_j(\bar{x}_{k\neq j})$, we keep track of the slope changes in $G_j(t|\bar{x}_{k\neq j})$ while storing the best *feasible* value t that is different from $\bar{x}_j$. It is easy to see that the initial slope corresponds to the sum of all negative slopes (see also Example 1), and once the slope becomes equal to 0 or the upper bound u_j is reached, then we found the optimal solution of $\min_{t\in[l_j,u_j],\, t\neq \bar{x}_j} G_j(t|\bar{x}_{k\neq j})$.

Algorithm 1 Jump value

```
Input Problem (1), incumbent solution $\bar{x}$, and variable index $j$
Output: The jump value.
1:  $\Delta \leftarrow [(l_j, 0), (u_j, 0)]$                ▷ Initialize a list (value,slope) pairs
2:  slope $\leftarrow 0$                                     ▷ Initialize the cumulative slope
3:  for $i \in M$ where $a_{ij} \neq 0$ do
4:      $\Delta$.insert$\big((t_{ij}(\bar{x}_{k\neq j}), w_i)\big)$   ▷ Add critical value and slope change
5:      if $a_{ij} < 0$ then                             ▷ If coeff. is negative, we start in a negative slope
6:          slope $\leftarrow$ slope $- w_i$
7:  $t^* \leftarrow -\infty$                                  ▷ Initialize the best value
8:  for $(t, w) \in$ sorted$(\Delta)$ do
9:      slope $\leftarrow$ slope $+ w$                       ▷ Update the slope
10:     if $l_j \leq t \leq u_j$ then
11:         $t^* \leftarrow t$                               ▷ Update the best value only when feasible
12:     if slope $\geq 0$ then                              ▷ Stop when the slope becomes non-negative
13:         break
14: return $t^*$
```

The following example demonstrates the computation of the *jump* value in a simple MIP problem.

Example 1 *Consider a pure feasibility problem (i.e., $q = 0$ in (2)) and consider the pair of constraints:*

$$x_1 + x_2 = 2$$
$$x_2 + x_3 \geq 3,$$

where $x_1, x_2, x_3 \in \mathbb{Z}^+$, the current incumbent is $\bar{x}_1 = 2, \bar{x}_2 = 0, \bar{x}_3 = 0$, and $w_1, w_2 = 1$. Figure 1 shows the constraint violation functions $g_{i,2}(t|\bar{x}_{k\neq j})$ and corresponding critical values for the first (Figure 1a) and second (Figure 1b) constraints, while Figure 1c shows the sum of those functions and the jump value. In this case, $T_2(\bar{x}_{k\neq j}) = \{0, 2, 3, +\infty\}$, and Algorithm 1 loops through these values starting from $l_2 = 0$ with a slope equal to -2. When hitting the critical value at $x_2 = 2$, the slope changes to 0, and since $\bar{x}_2 \neq 2$, then 2 is the jump *value.*

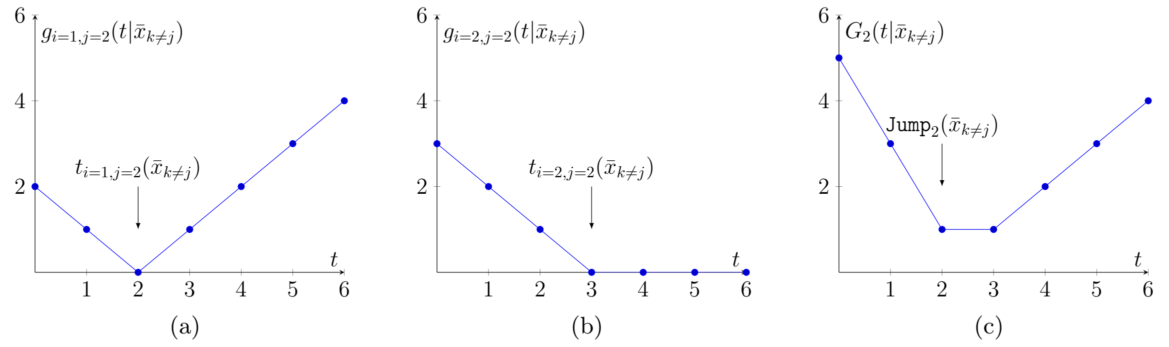

Figure 1: The constraint violation functions for variable x_2 (Figure (a) and (b)), and their sum (Figure (c)).

The *jump* value was motivated by the *best shift* of the `Shift-and-Propagate` heuristic [8]. In [8], given an incumbent solution $\bar{x}$, the best-shift for a variable x_j is the value ψ_j such that $\bar{x}_j + \psi_j$ reduces the largest number of violated constraints (assuming the rest of the incumbent stays the same). This is in contrast with the infeasibility penalties described in (4), where even partial reductions of constraint violations are considered. The intrinsic problem of considering only when constraints switch between being violated and satisfied is the loss of information associated with, for example, the following combination of constraint and incumbent solution:

$$x_1 + \cdots + x_n \geq b, \quad x_1, \ldots, x_n \in \{0, 1\},\ n \geq 2,\ b > 1,\ \bar{x}_1, \ldots, \bar{x}_n = 0.$$

Since in the current incumbent all variables have value zero, changing the value of any of the variables individually by +1 would not be enough to satisfy this constraint, and there would no incentive for the algorithm to do so.

In the next section, we show how can we construct neighbourhoods based on the *jump* value.

2.3 Lazy adaptive neighborhoods

Feasibility Jump looks for local minima of (2) by traversing neighbourhoods in which only one variable at a time can change its value. For a particular incumbent solution $\bar{x}$, we define a neighborhood $\mathcal{N}(\bar{x})$ for problem (2) by simply associating a pair (v_j, s_j) to each variable x_j, where v_j is a new value the variable could jump to and s_j represents its corresponding score.

This score is computed as the difference between the current total constraint violation penalty $G_j(\bar{x}_j|\bar{x}_{k\neq j})$ and the same penalty obtained by changing the current value of x_j from $\bar{x}_j$ to v_j:

$$s_j = G_j(\bar{x}_j|\bar{x}_{k\neq j}) - G_j(v_j|\bar{x}_{k\neq j}). \tag{7}$$

The value v_j is computed based on the *jump* value, but is not necessarily always equal to the *jump* value at all times. Ideally, since in the previous section we went through the trouble of computing the value that minimizes $G_j(t|\bar{x}_{k\neq j})$, we would like to always have $v_j = \texttt{Jump}_j(\bar{x}_{k\neq j})$. But since the *jump* values depend on the current incumbent $\bar{x}$, we would need to recompute all of them every time we change $\bar{x}$, which means every time we make a single variable jump to a new value. This can easily become too computationally expensive.

We consider instead "lazy" neighbourhood updates. The idea is to initialize the very first neighbourhood with the *jump* values, $v_j = \texttt{Jump}_j(\bar{x}_{k \neq j})$ for all $j \in N$. Then, every time a variable x_j performs a jump, we update *only* its value $v_j = \texttt{Jump}_j(\bar{x}_{k \neq j})$, keeping the remaining $v_k, k \neq j$ intact, but updating all the scores s_j, $j \in N$. This means that in any neighbourhood except the first one, only the value v_j of the variable that performed a jump in the previous neighbourhood is guaranteed to be equal to its corresponding *jump* value, as defined in (1). Note that only the scores of the variables that share the same constraints as the previous "jumping" variable need to be updated, and this can be done with a simple single pass.

A positive score $s_j > 0$ means that assigning the new value v_j to x_j will reduce the total constraint violation of the current incumbent solution. Having defined our neighborhood $\mathcal{N}$ and how to update it, we can hope that repeatedly choosing new assignments for variables with positive scores will eventually lead us to a feasible solution, or even to an optimal solution (when $q > 0$). There are no guarantees, however, that the neighborhood will always contain such an improving assignment, i.e., that there exist a variable with a positive score. When this happens, we say that the local search is stuck in a local minimum. To escape such a local minimum, we can add a new layer of heuristics that will try to perturb the current assignment or search parameters so that local search will not return to the same local minimum but instead find a new and potentially better one. The next section describes exactly this.

2.4 How to guide the search

The specific meta-heuristic we use in the Feasibility Jump is known as guided local search (see [4] for a detailed survey). Guided local search works by modifying the objective function of the local search whenever the search is stuck in a local minimum. In our Lagrangian relaxation (2), we introduced the weight parameters w_i, $i \in M$, to be able to adjust the "importance" of each constraint individually. Since these parameters influence the scores s_j, $j \in N$, we hope that we can use them to guide the search out of local minima and towards a feasible (or optimal) solution.

The heuristic we use for updating these weights is based on the fact that whenever we reach a local minimum, we expect most of the constraints in our original MIP (1) to be satisfied by the current assignment. By increasing the weights of the few remaining violated constraints, we hope that subsequent solutions will be less likely to violate them, since the corresponding penalties (5) will be higher. In general, a MIP problem may contain constraints that are easy to satisfy, and other constraints (or combinations of constraints) that are difficult to satisfy. By increasing the weight of the latter we focus the search on the constraints that are the hardest to satisfy.

We update the weights w_i, $i \in M$, at every local minimum by setting:

$$w_i \leftarrow \Delta W(w, \bar{x}, i),$$

where ΔW is some weight update function that depends on the current local minimum $\bar{x}$. The simplest weight update is to increase the weight of any violated constraints by a constant amount:

$$\Delta W^+(w, \bar{x}, i) = \begin{cases} w_i & \sum_{j \in \mathcal{N}} a_{ij}\bar{x}_j \leq b_i \\ w_i + 1 & \text{otherwise.} \end{cases}$$

This is in fact the update function that we have used. We did also experiment with multiplicative updates, i.e.,

$$\Delta W^*(w, \bar{x}, i) = \begin{cases} w_i & \sum_{j \in \mathcal{N}} a_{ij} \bar{x}_j \leq b_i \\ \lambda w_i & \text{otherwise,} \end{cases}$$

which also requires some implementation tricks to avoid saturating the floating point number representation, but we found that it had no significant effect on the algorithm's performance.

If we reach a local minimum where no constraints are violated, then we have found a feasible solution, and ΔW does not change any weights. We can now modify the parameter q of (2) to increase the weight of the original MIP objective and hope that this leads to a different feasible solution that has a better objective value in the original MIP (1).

We update q in every local minimum by setting:

$$q \leftarrow \Delta Q(q, \bar{x}),$$

for some objective weight update function ΔQ. The simplest such update is simply to increase its value by a constant amount whenever all linear constraints are satisfied.

$$\Delta Q^+(q, \bar{x}) = \begin{cases} q + 1 & \sum_{j \in \mathcal{N}} a_{ij} \bar{x}_j \leq b_i,\ i \in M \\ q & \text{otherwise} \end{cases}$$

While different and more sophisticated update functions could be derived, we found the current results satisfying.

2.5 The algorithm

In this section we combine the ideas described in the previous sections and present our LP-free Langrangian MIP heuristic. Feasibility Jump performs a highly-efficient guided local search by computing promising neighbourhoods based on the jump values, and updating them lazily. The steps of the algorithms are shown in Algorithm 2. The algorithm starts from any (potentially infeasible) assignment, and measures how far away the current incumbent assignment is from satisfying the constraints (i.e., from feasibility). We maintain for each variable a promising new value (which is different from the current incumbent value), and we associate to this value a score that is proportional to the potential reduction in the total constraints violation. At each iteration (line 3-29), we assign a new value to a promising variable (line 27) and we perform two updates: we compute the jump value for the current 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 (line 13), 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 (line 15-16). After the first feasible solution has been found, we augment the function that computes the score of each variable by taking into account the objective function of the original MIP (line 18-21). This helps steering the search towards feasible solutions with better objective value.

To further reduce the computational effort of the heuristic, we maintain a set of variable indices with a positive score, and we choose the variable index with the highest score among a small random sample of them. We found that maintaining the ranking of all variables to find the highest scoring one had a modest negative impact in the performance (i.e., number of "jumps per second"), while providing almost zero benefits. Moreover, a bit of randomness

Algorithm 2 Feasibility Jump

```
    Input relaxed MIP problem (2), initial assignment $\tilde{x}$
    Output: a feasible solution or NULL
 1: best_feas $\leftarrow \infty$, best_obj $\leftarrow \infty$, $x^* \leftarrow$ NULL    ▷ Initialize best feasible solution
 2: $\bar{x} \leftarrow \tilde{x}$    ▷ Initialize incumbent
 3: $q = 0$, $w_i = 1$, $i \in M$    ▷ Initialize weights
 4: $v_j = \texttt{Jump}_j(\bar{x}_{k \neq j})$, $j \in N$    ▷ Initialize promising values
 5: $s_j = G_j(\bar{x}_j | \bar{x}_{k \neq j}) - G_j(v_j | \bar{x}_{k \neq j})$, $j \in N$    ▷ Initialize scores
 6: $P = \{j \in N : s_j > 0\}$    ▷ Initialize set of indices with positive score
 7: while ShouldTerminate() is false do
 8:     if $F^w(\bar{x}) <$ best_feas then
 9:         best_feas $\leftarrow F(\bar{x})$    ▷ Update the best violation penalty
10:     if $F^w(\bar{x}) = 0$ and $O^q(\bar{x}) <$ best_obj then
11:         $x^* \leftarrow \bar{x}$    ▷ Update the best feasible solution
12:         best_obj $\leftarrow O^q(\bar{x})$    ▷ Update the best weighted objective
13:     if $P = \emptyset$ then    ▷ Whether we reached a local minimum
14:         if $F^w(\bar{x}) = 0$ then
15:             $q \leftarrow q + 1$    ▷ Put more emphasis on the original objective
16:             Update all scores $s_j : c_j \neq 0, j \in N$
17:         else
18:             $U \leftarrow \emptyset$
19:             for $i \in N : f_i(\bar{x}) > 0$ do
20:                 $w_i \leftarrow w_i + 1$    ▷ Put more emphasis on satisfying this constraint
21:                 $U \leftarrow U \cup i$
22:             Update all scores $s_j : a_{ij} \neq 0, i \in U, j \in N$
23:         $j^* =$ random choice in $N$    ▷ Random move
24:     else
25:         $P^* =$ randomly choose up to 100 indices from $P$
26:         $j^* = \arg\max_{j \in P^*} s_j$    ▷ Best move among a random set of good ones
27:     $\bar{x}_{j^*} \leftarrow v_{j^*}$    ▷ Make the move
28:     $v_{j^*} = \texttt{Jump}_{j^*}(\bar{x}_{k \neq j^*})$    ▷ Recompute a new promising value
29:     Update all scores $s_j$ of the neighboring variables
30: return $x^*$
```

can sometimes be beneficial when dealing with MIP problems (see, for example [13]). We use a sample size of $\min\{n, 100\}$.

As there is no natural time at which to stop the algorithm, we use an estimate of the computation effort expended by the algorithm. One of the easiest ways to do this is simply to sum the size of the bounds of every for-loop that runs. The advantage of using such an effort estimate over measuring wall-clock time, is that the heuristic runs deterministically, which simplifies debugging, adds reliability, and is typically a property that MIP solvers offer. The algorithm terminates after some amount of effort has been expended since the last improvement made, that is since the last time $F^w(\bar{x})$ or $O^q(\bar{x})$ have been reduced. The value of the threshold has currently been tuned based on the MIPLIB 2017 benchmark set [17]. Whenever FJ runs within a MIP solver, one could also simply decide to stop the algorithm after any feasible solution has been found (either by FJ or by other components of the MIP solver).

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

- 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.

3 Implementation

We have developed a C++ reference implementation of Feasibilty Jump that is not dependent on any other MIP solver. This solver is available online[1] under an open source license. To use it, one creates a new `FeasibilityJumpSolver` object and adds variables by calling the `addVar` method. Constraints are added by calling the `addConstraint` method. The `solve` function takes an initial assignment and a callback function parameter that the solver will call periodically with an `FJStatus` object, containing the effort spent so far and any new solutions found and their objective value. The callback function's return value decides whether to continue or abort the heuristic.

```
// Method signatures
class FeasibilityJumpSolver {
  FeasibilityJumpSolver(int seed=0, int verbosity=0);
  int addVar(VarType type, double lb, double ub, double coeff);
  int addConstraint(RowType sense, double rhs, int numCoeffs, int *rowVarIdxs,
    double *rowCoeffs);
  int solve(double *x, function<int(FJStatus)> callback);
};
```

The C++ implementation is generic in the type of neighborhoods it uses, with the Jump value being one of many possible. We have also experimented with other neighborhoods, such as the nearest integer values (i.e., $\bar{x}_j \leftarrow \bar{x}_j \pm 1$), but found that they had only marginal impact on the solver's performance. All uses of the neighborhood throughout the algorithm are performed through the method `forEachMove`, where all moves for a variable are enumerated. The neighborhood can be modified by simply editing this function, which has the following signature:

```
template <typename F> void forEachMove(int32_t varIdx, F f);
```

[1]`https://github.com/SINTEF/feasibilityjump`

Our reference implementation can also be used as a starting point for other experiments with MIP local search, such as modifying the weight update functions $\Delta W, \Delta Q$ or testing other meta-heuristics.

Feasibility Jump can be used both a standalone heuristic or integrated within an existing exact MIP solver. In the latter case, one would be interested not only in improving the average time to first feasible solution, but also the average time to optimal solution. Current state-of-the-art MIP solvers, such as FICO Xpress, are already so effective that improving the average solving time of just few percent would be a great achievement. We used the interface presented above also to integrate the heuristic with Xpress. But since we only have access to Xpress' external interface, it requires copying all the constraints from the Xpress problem instance representation to the heuristic's representation. Except for duplicating the constraint coefficients (also called *the matrix*), Feasibility Jump requires very little memory, proportional the sum of the number of variables and the number of constraints. Duplicating the constraint coefficients would not be required if the heuristic was implemented using the MIP solver's internal interfaces.

To integrate FJ with Xpress, we start by loading the MIP problem into Xpress. Then, we copy all the variables and constraints from Xpress to FJ and start FJ on a background thread. After starting FJ (i.e., on the non-presolved instance), we presolve the problem using Xpress and copy all the variables and constraints of the presolved problem into a new FJ instance, which is launched on another background thread. We then call the main `MIPoptimize` function of Xpress with a `checktime` callback function, which Xpress will periodically call. In this callback function we check if any solutions have been found by any of the two FJ threads, and, if so, copy them to Xpress by using the `addMIPsolution` method.

Our C++ reference implementation is around 800 lines of code, with Xpress integration adding an additional 500 lines. The version of Feasibility Jump that participated in the computational competition of the 2022 Mixed Integer Programming Workshop was written in Rust and contained a few additional tunable parameters and additional move types. These additional features only allowed finding feasible solutions to a few more instances from the MIPLIB 2017 Benchmark collection, and we found that the increase in complexity was not worth-while for the presentation in this paper (and its accompanying source code), nor for integration in a more complete branch-and-bound MIP solver system. The Rust competition implementation is available on request.

4 Computational results

In this section, we provide an extensive set of results to assess the performance and effectiveness of Feasibility Jump. All tests were executed on single threads of an AMD Threadripper 3990x CPU running at 2.9GHz with 128GB of memory. The instances we used for testing belong to the MIPLIB 2017 benchmark set [17], a widely-used set of 240 mixed-integer problems of various size and difficulty.

We first look at average time it takes FJ to perform an iteration, that is an iteration of the while loop in line 3 of Algorithm 2. Figure 2 shows a somewhat linear correlation of the iteration time and the average number of non-zero coefficients per variable. More importantly, it shows that for many problems of MIPLIB 2017, an iteration can easily take less than a microsecond.

The next computational results will consider three different solvers:

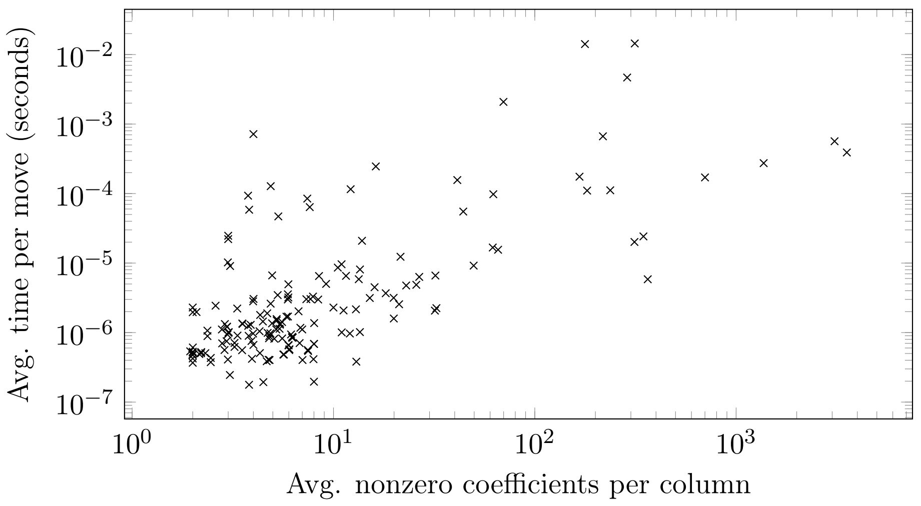

Figure 2: Average computation time (in seconds) for each iteration of the while loop of Algorithm (2), for each problem instance in MIPLIB 2017.

- `FJ`: Feasibility Jump, as described in Section 2.5 and Algorithm 2;
- `XPR`: FICO Xpress Solver 8.14 in its default settings;
- `XPR+FJ`: Feasibility Jump integrated into Xpress as described in Section 3.

We compare these solvers on the time it takes to find the first feasible solution of a MIP problem (Section 4.1), and the time it takes to find an optimal solution (Section 4.2). Note that FJ typically produces solutions very quickly, if it produces any at all. We did not see significant improvements in FJ's performance by increasing the available computational effort with the current implementation. Feasbility Jump uses a flexible and intuitive framework, which could be extended to increase its ability of finding feasibile solutions, perhaps at the cost more computational effort. But, when integrated in a more comprehensive MIP solver, a very quick heuristic may be preferred, so that more computation time is allocated to methods that are guaranteed to eventually produce solutions (for example, branch-and-bound).

4.1 Time to first feasible solution

We first compare `FJ`, `XPR`, and `XPR+FJ` on the time it takes to find the first feasible solution (TFS). We run all solvers with a time limit of 1 minute, although `FJ` will usually terminate earlier even when not finding a feasible solution (the termination criterion is described in Section 2.5).

Figure 3 shows the fraction of instances $P_s(\tau)$ for which a solver s is the fastest when its running times are scaled by $1/\tau$. It is not surprising to see `XPR+FJ` always in the lead, since it combines solutions both from `XPR` and `FJ`. Still, it represents a notable improvement compared to standalone Xpress, `XPR`. This is also summarized in Figure 4, where we show the ratio between the TFS of `XPR` and the TFS of `XPR+FJ`. Instances where only one of the solvers found a feasible solution within the time limit are represented by plus and minus infinity. Thanks to Feasbility Jump, `XPR+FJ` found a feasible solution to 6 more problems compared to `XPR`, and provided an average reduction of 25% on the time to first feasible solution.

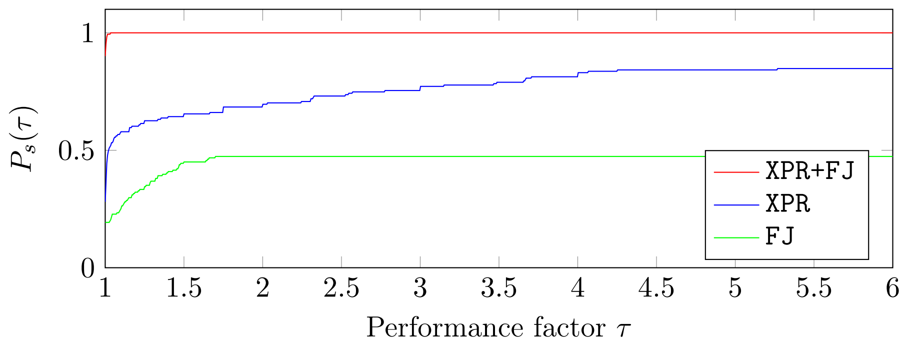

Figure 3: Performance profile of the three solvers in regards to the time they take to find a first feasible solution. Since `XPR+FJ` contains the best solution of either of the other two solvers, it is always the obvious winner.

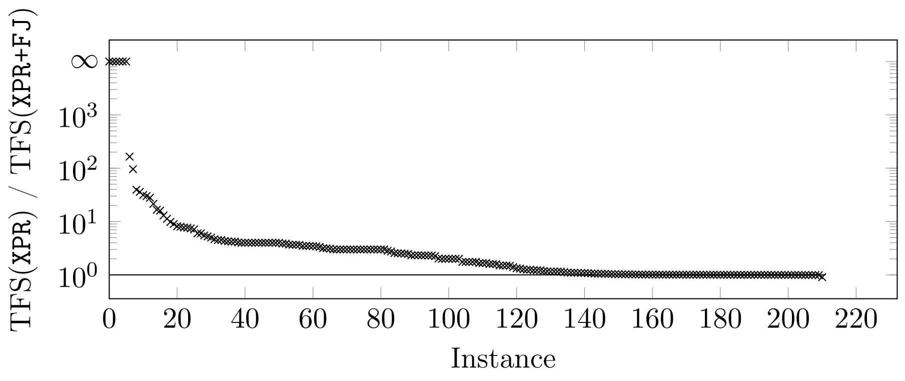

Figure 4: Logscale plot of the ratio between the time to first feasibile solution of `XPR` versus `XPR+FJ` in all instances of the MIPLIB 2017.

4.2 Time to optimal solution

Also of interest is to check whether quick feasible solutions found by Feasibility Jump can be exploited by a MIP solver to solve some problems faster. We run `XPR` and `XPR+FJ` with a 10 minute time limit, in which the solvers were able to find an optimal solution for 84 of the MIPLIB 2017 benchmark set instances. Similar to the previous section, Figure 5 shows the fraction of instances $P_s(\tau)$ for which a solver s is the fastest when its running times are scaled by $1/\tau$. Figure 6, instead, shows the ratio between the TOS of `XPR` and TOS of `XPR+FJ`. Instances where only one of the solvers found an optimal solution within the time limit are represented by plus and minus infinity. We found that the time to optimal solution (TOS) improved for a small subset of instances. But there are also other cases in which the solving time remained the same even though Feasibility Jump provided Xpress with good solutions (better than those found by `XPR` in the same time interval) early in the solution process. On average, we see a reduction on the solving time of about 3%.

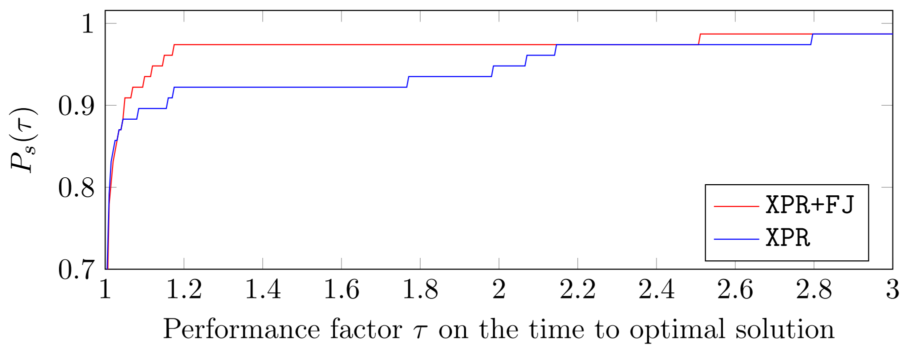

Figure 5: Performance profile of `XPR` versus `XPR+FJ` in regards to the time they take to find an optimal solution.

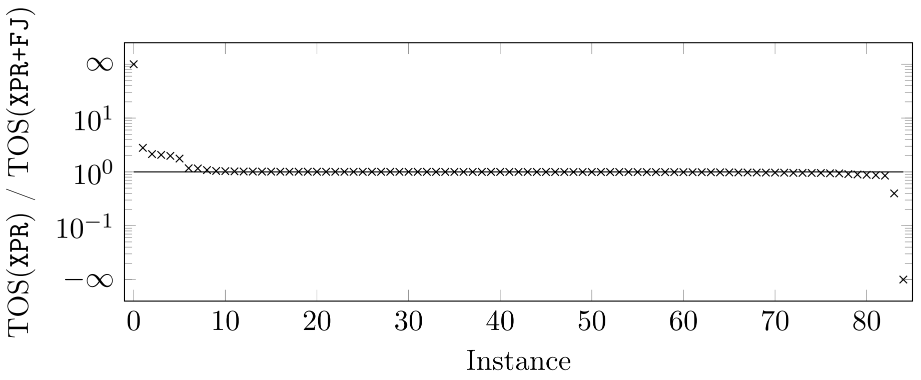

Figure 6: Logscale plot of the ratio between the time to optimal solution of `XPR` versus `XPR+FJ` in all instances of the MIPLIB 2017.

5 Conclusions

We have introduced the Feasibility Jump primal heuristic for mixed-integer linear programming. It is based on a guided local search approach over a Lagrangian relaxation of the original problem. The algorithm finds feasible solutions on some benchmark problems where even state-of-the-art commercial solvers can struggle. Because Feasibility Jump handles very large problem instances, and is not based on a solution to the root relaxation, it can find solutions to some instances very early in the solution process, sometimes even before presolve has finished.

We have integrated Feasibility Jump with FICO Xpress and shown that, in addition to providing feasible solutions early, supplying those solutions back to a MIP solver can also improve the time to find the optimal solution.

An efficient C++ implementation of the algorithm is available under an open-source license, and can be extended with additional neighborhood definitions or meta-heuristics.

The current algorithm uses a constant weight update function, but we expect that more sophisticated heuristics for setting the weights of individual constraints could increase the effectiveness of Feasibility Jump.

References

[1] Mip 2022 workshop. `https://www.mixedinteger.org/2022/` (2022)

[2] Aarts, E., Aarts, E.H., Lenstra, J.K.: Local search in combinatorial optimization. Princeton University Press (2003)

[3] Achterberg, T.: Constraint integer programming. Ph.D. thesis, Berlin Institute of Technology (2007). URL `http://opus.kobv.de/tuberlin/volltexte/2007/1611/`

[4] Alsheddy, A., Voudouris, C., Tsang, E.P.K., Alhindi, A.: Guided local search. In: R. Martí, P.M. Pardalos, M.G.C. Resende (eds.) Handbook of Heuristics, pp. 261–297. Springer (2018). DOI 10.1007/978-3-319-07124-4_2. URL `https://doi.org/10.1007/978-3-319-07124-4_2`

[5] Balas, E., Schmieta, S., Wallace, C.: Pivot and shift—a mixed integer programming heuristic. Discrete Optimization **1**(1), 3–12 (2004)

[6] Berthold, T.: Primal heuristics for mixed integer programs. Diplomarbeit, Zuse Institute Berlin (ZIB) (2006)

[7] Berthold, T.: Rens. Mathematical Programming Computation **6**(1), 33–54 (2014)

[8] Berthold, T., Hendel, G.: Shift-and-propagate. J. Heuristics **21**(1), 73–106 (2015). DOI 10.1007/s10732-014-9271-0. URL `https://doi.org/10.1007/s10732-014-9271-0`

[9] Bertsekas, D.P.: Constrained optimization and Lagrange multiplier methods. Academic press (2014)

[10] Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve mip solutions. Mathematical Programming **102**(1), 71–90 (2005)

[11] Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming **104**(1), 91–104 (2005)

[12] Fischetti, M., Lodi, A.: Local branching. Mathematical programming **98**(1), 23–47 (2003)

[13] Fischetti, M., Monaci, M.: Exploiting erraticism in search. Operations Research **62**(1), 114–122 (2014)

[14] Fischetti, M., Sartor, G., Zanette, A.: Mip-and-refine matheuristic for smart grid energy management. International Transactions in Operational Research **22**(1), 49–59 (2015)

[15] Frangioni, A.: About lagrangian methods in integer optimization. Annals of Operations Research **139**(1), 163–193 (2005)

[16] Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-driven fix-and-propagate heuristics for mixed integer programming. Mathematical Programming Computation **11**(4), 675–702 (2019)

[17] Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P.M., Jarck, K., Koch, T., Linderoth, J., Lübbecke, M., Mittelmann, H.D., Ozyurt, D., Ralphs, T.K., Salvagnin, D., Shinano, Y.: MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Mathematical Programming Computation (2021). DOI 10.1007/s12532-020-00194-3. URL `https://doi.org/10.1007/s12532-020-00194-3`

[18] Gomory, R.E.: An algorithm for integer solutions to linear programs. Recent advances in mathematical programming **64**(260-302), 14 (1963)

[19] Lei, Z., Cai, S., Luo, C., Hoos, H.H.: Efficient local search for pseudo boolean optimization. In: C. Li, F. Manyà (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Proceedings, *Lecture Notes in Computer Science*, vol. 12831, pp. 332–348. Springer (2021). DOI 10.1007/978-3-030-80223-3_23. URL `https://doi.org/10.1007/978-3-030-80223-3_23`

[20] Lodi, A.: Mixed integer programming computation. In: 50 years of integer programming 1958-2008, pp. 619–645. Springer (2010)

[21] Salvagnin, D.: Personal communication (October 2022)

[22] Shapiro, J.F.: A survey of lagrangean techniques for discrete optimization. In: Annals of Discrete Mathematics, vol. 5, pp. 113–138. Elsevier (1979)

[23] Shen, Y., Sun, Y., Eberhard, A., Li, X.: Learning primal heuristics for mixed integer programs. In: 2021 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2021)

[24] Song, J., Yue, Y., Dilkina, B., et al.: A general large neighborhood search framework for solving integer linear programs. Advances in Neural Information Processing Systems **33**, 20012–20023 (2020)

[25] Witzig, J., Gleixner, A.: Conflict-driven heuristics for mixed integer programming. INFORMS Journal on Computing **33**(2), 706–720 (2021)

[26] Wolsey, L.A.: Integer programming. John Wiley & Sons (2020)