Mathematical Programming Computation (2023) 15:365–388
https://doi.org/10.1007/s12532-023-00234-8

FULL LENGTH PAPER

Feasibility Jump: an LP-free Lagrangian MIP heuristic

Bjørnar Luteberget[1] · **Giorgio Sartor**[1]

Received: 4 November 2022 / Accepted: 22 January 2023 / Published online: 14 March 2023

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 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, significantly improving the average time to find a first feasible solution. We also show that providing these quick solutions to Xpress produces a modest reduction in the average time to optimality 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 Feasibility Jump now runs by default on FICO Xpress Solver 9.0, where similar results to the ones presented here could be observed.

Keywords Mixed integer linear programming · Metaheuristics · Lagrangian relaxation · Guided local search · Primal heuristics

Mathematics Subject Classification 90C11 · 90C59

✉ Giorgio Sartor
giorgio.sartor@sintef.no

Bjørnar Luteberget
bjornar.luteberget@sintef.no

[1] SINTEF, Forskningsveien 1, 0373 Oslo, Norway

1 Introduction

In mathematical programming, heuristic algorithms have always been an essential tool to quickly find good feasible solutions. Not only are they 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 [22]. 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 (note that finding a feasible solution for a MIP problem is NP-hard [31]). 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 LP relaxation at the root node, such as *local branching* [13], *pivot and shift* [4], *feasibility pump* [12], *RINS* [10], *RENS* [6], and tabu search methods (see, for example, [23]). Some more recent heuristics that exploit information obtained from the LP relaxation of a MIP problem include conflict-driven diving heuristics [30] and even ML-based heuristics, such as [28, 29].

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 [7] *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. [18], 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. [21] 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" [17].

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 [21] 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 [7]. 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, [2, 5, 16]).

The algorithm was originally developed for the MIP 2022 Workshop's Computational Competition [25], 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 within 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 [19]. 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

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

where $N = \{1, \ldots, n\}$, $M = \{1, \ldots, m\}$, $x \in \mathbb{R}^n$, $a_{ij}, c_j, b_i \in \mathbb{R}$, $l_j \in \mathbb{R}$ and $u_j \in \mathbb{R}$ 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 [1].

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* [12] (to which we owe the inspiration for our heuristic's name) is an 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* [27].

In the next sections, we describe all the pieces that contribute to Feasibility Jump. We first introduce a Lagrangian relaxation of the MIP problem (Sect. 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 (Sect. 2.2). We then describe how these values can be used to efficiently define new neighborhoods (Sect. 2.3) and how to proceed when reaching a local minimum (Sect. 2.4). Finally, Sect. 2.5 summarizes the entire algorithm and Sect. 2.6 describes how to extend FJ to take the original objective function into account.

2.1 Relaxing the linear constraints

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

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

where $F^w(x)$ is the total infeasibility penalty incurred by x. It is 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 $w_i \geq 0$ is a weight associated to each constraint and the infeasibility penalty $f_i(\cdot)$, $i \in M$, for a single linear constraint $\sum_{j \in N} a_{ij} x_j \leq b_i$ is

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

Note that problem (2) does not contain the original objective of (1), and solving (2) corresponds to finding feasible solutions to (1). We discuss in Sect. 2.6 how to extend (2) to also consider the original objective function.

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. It has been shown this can be beneficial for feasibility heuristics (see, for example, [12]), and we also hope this could yield feasible solutions with better objective value.

The first surveys on Lagrangian techniques for discrete optimization started appearing already in the 1970 s [27], but the basic idea did not change since then: minimize the Lagrangian function $F^{\bar{w}}(x)$ for fixed $\bar{w}$, produce modified weights $\bar{w}'$ (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 minimize the Lagrangian function exactly, but rather look for a local minimum in neighborhoods where we only change the value of one variable at a time. The next section describes how to compute a promising value for each variable.

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 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 to have a non-empty neighborhood for each variable so that there are always moves available, even if they do not improve the objective value. 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 $a_{ij} < 0$ [20]. In this way, we include the integrality constraint of x_j directly into the constraint violation measure.

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

$$t_{ij}(\bar{x}_{k\neq j}) = \begin{cases} \lfloor r \rfloor & a_{ij} > 0 \\ \\ \lceil r \rceil & a_{ij} < 0 \end{cases} \qquad r = \frac{1}{a_{ij}} \left(b_i - \sum_{k \neq j} a_{ik}\bar{x}_k \right), \tag{5}$$

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 constraint 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 $g_{ij}(t \mid \bar{x}_{k \neq j})$ such that:

$$g_{ij}(t|\bar{x}_{k\neq j}) = \begin{cases} \max\left\{0, w_i\big(t - t_{ij}(\bar{x}_{k\neq j})\big)\right\} & a_{ij} > 0 \\ \\ \max\left\{0, -w_i\big(t - t_{ij}(\bar{x}_{k\neq j})\big)\right\} & a_{ij} < 0, \end{cases} \tag{6}$$

where $t \in \mathbb{R}$ and $j \in N$. In other words, $g_{ij}(t|\bar{x}_{k\neq j})$ is equivalent to the function $f_i(x_j \mid x_k = \bar{x}_k,\ k \neq j)$ translated by the fractional part of r, as defined in (5).[1] Note that one could also choose to normalize the penalty function g_{ij} by a_{ij}, but we did not experiment with this.

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,

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

[1] We found that using the constraint violation functions $f_i(x_j \mid x_k = \bar{x}_k,\ k \neq j)$ (as opposed to $g_{ij}(t|\bar{x}_{k\neq j})$) to compute the *jump* value in (7), did not perform as well, reducing the number of problems from the MIPLIB 2017 benchmark set for which Feasibility Jump found a feasible solution by 5%.

Then we have:

$$\texttt{Jump}_j(\bar{x}_{k\neq j}) = \min\left(\underset{t\in[l_j,u_j],t\neq\bar{x}_j}{\arg\min} G_j(t|\bar{x}_{k\neq j})\right). \tag{7}$$

By looking at (6), 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 from $\bar{x}_j$, 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 finding the set of critical values $T_j(\bar{x})$, and by computing the cumulative slope of $G_j(t|x_{k\neq j})$ when approaching $t = l_j$ from below (see lines 8–11 and also Example 1 below). We then loop through the *feasible* values of $T_j(\bar{x})$ in ascending order, keeping track of the slope changes in $G_j(t|\bar{x}_{k\neq j})$ while storing the best value that is different from $\bar{x}_j$. Note that since we can assume that $l_j \neq u_j$ (otherwise the variable x_j is fixed), then we will always reach line 16 at least once, and the algorithm will return a value different from the incumbent one. If we reach the upper bound, or the slope becomes greater or equal to 0, then we have found the optimal solution of $\min_{t\in[l_j,u_j],\, t\neq\bar{x}_j} G_j(t|\bar{x}_{k\neq j})$. The correctness is easy to prove, since we are simply traversing a piecewise-linear convex function from l_j to u_j, where the only slope changes happen at the critical values. Starting from the correct slope at l_j and updating the slope at each feasible critical value (plus l_j and u_j), either we reach the upper bound or the slope becomes non-negative (i.e., we reached the bottom of $G_j(t|x_{k\neq j})$). Then the best value found up this point (which always exists) is the optimum.

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

Example 1 Consider a pure feasibility problem and consider the pair of constraints:

$$\begin{aligned} x_1 + x_2 &= 3 \\ x_2 + x_3 &\geq 3, \end{aligned}$$

where $x_1, x_2, x_3 \in \mathbb{Z}^+$, the current incumbent is $\bar{x}_1 = 1, \bar{x}_2 = 2, \bar{x}_3 = 0$, and $w_1, w_2 = 1$. Figure 1shows the constraint violation functions $g_{i,2}(t|\bar{x}_{k\neq j})$ and corresponding critical values for the first (Fig. 1a) and second (Fig. 1b) constraints, while Fig. 1c shows the sum of those functions and the jump value (the equality constraint is simply the sum of a $\geq$ and $\leq$ constraint). 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 we reached the bottom of the total constraint violation function for variable x_2. But since $\bar{x}_2 = 2$, then we continue to the next critical value, $x_2 = 3$, which becomes the *jump* value.

Algorithm 1 Jump value

Input Problem (2), incumbent solution $\bar{x}$, and variable index j
Output: The *jump* value.

$t^* \leftarrow l_j$ ▷ *Initialize the best value*
$\Delta \leftarrow [(l_j, 0), (u_j, 0)]$ ▷ *Initialize the list of (value,slope) pairs*
slope $\leftarrow 0$ ▷ *Initialize the cumulative slope*
for $i \in M$ where $a_{ij} \neq 0$ **do**
  $t \leftarrow t_{ij}(\bar{x}_{k \neq j})$ ▷ *Compute the critical value*
  if $l_j \leq t \leq u_j$ **then**
    Δ.insert$((t, w_i))$ ▷ *Add feasible critical value and slope change*
  if $t \geq l_j$ **and** $a_{ij} < 0$ **then**
    slope $\leftarrow$ slope $- w_i$ ▷ *Accumulate the negative slopes before* l_j
  if $t < l_j$ **and** $a_{ij} > 0$ **then**
    slope $\leftarrow$ slope $+ w_i$ ▷ *Accumulate the positive slopes before* l_j
for $(t, w) \in$ sorted(Δ) **do** ▷ *Iterate* Δ *in ascending order by the first component*
  slope $\leftarrow$ slope $+ w$ ▷ *Update the current slope*
  if $t = \bar{x}_j$ **then**
    continue ▷ *Skip values equal to the current incumbent*
  $t^* \leftarrow t$ ▷ *Update the best value*
  if slope ≥ 0 **then**
    break ▷ *Stop when the slope becomes non-negative*
return t^*

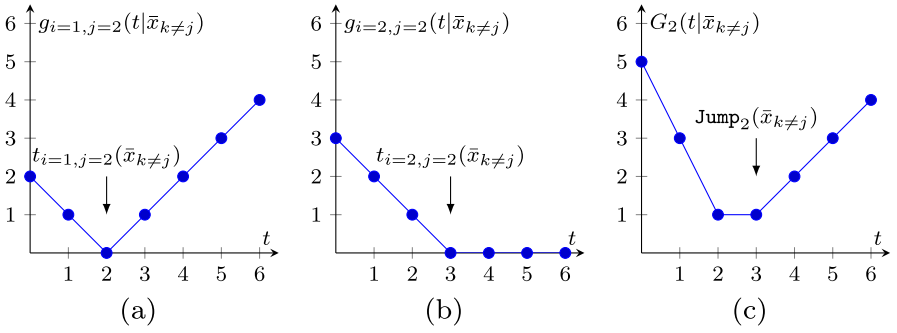

Fig. 1 The constraint violation functions for variable x_2 (**a**, **b**), and their sum (**c**)

The *jump* value was motivated by the *best shift* of the `Shift-and-Propagate` heuristic [7]. In [7], given an incumbent solution $\bar{x}$, the best-shift for a variable x_j is the value ψ_j such that $\bar{x}_j + \psi_j$ minimizes the 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 be no incentive for the algorithm to do so.

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

2.3 Lazy adaptive neighborhoods

Feasibility Jump looks for local minima of (2) by traversing neighborhoods 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:

$$s_j = G_j(\bar{x}_j|\bar{x}_{k\neq j}) - G_j(v_j|\bar{x}_{k\neq j}). \tag{8}$$

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.

The value v_j is initialized to the *jump* value, but it is updated in a lazy fashion. 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. One could also choose to recompute all of them at regular intervals or after a certain condition gets satisfied, but we decided not to experiment with this behavior (in limited preliminary experiments, we saw that even recomputing all *jump* values after each jump would not significantly reduce the amount of jumps necessary to reach a feasible solution, while increasing the computation time).

We consider instead "lazy" neighborhood updates. The idea is to initialize the very first neighborhood 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 neighborhood except the first one, only the value v_j of the variable that performed a jump in the previous neighborhood is guaranteed to be equal to its corresponding $jump$ value, as defined in (1). In other words, performing a jump with positive score will always improve feasibility since the scores are always updated correctly, but one might miss better *jump* values since only a single value v_j is updated after performing a jump. 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.

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. There are no guarantees, however, that the neighborhood will always contain such an improving assignment, i.e., that there exists 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 the 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 metaheuristic we use in Feasibility Jump is known as guided local search (see [3] 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. This penalty method is indeed similar to the classical Lagrangian relaxation method, and can be seen as its approximation.

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 (6) 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.

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 neighborhoods 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 the score as defined in (8). In each iteration (line 8–26), we assign a new value to a promising variable (line 23) and we perform two updates: we compute the jump value for the current variable (line 24), and we update the scores of the variables that appear in the same constraints (line 25). If no value exists that improves the current sum of constraint violations (line 11), we have reached a local minimum. If there are unsatisfied constraints, then we increase their weight (line 14), we update the scores of the values of the variables involved (line 16), and we choose the best move within one random unsatisfied constraint (line 18–19). If the current assignment is feasible, we return it (line 8–10).

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 can sometimes be beneficial when dealing with MIP problems (see, for example [14]). We use a sample size of $\min\{n, 25\}$. Also, with a probability of 0.1%, we use a sample size of 1.

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 $F^w(\bar{x})$ reached its lowest value yet) or if a given total amount of effort has been exceeded. This stopping criteria is referred to as ShouldTerminate() in line 7 of Algorithm 2. The value of the threshold has currently been tuned based on the MIPLIB 2017 benchmark set [19]. 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:

Algorithm 2 Feasibility Jump

Input Problem (2), initial assignment $\tilde{x}$
Output: a feasible solution or NULL

1: $x^* \leftarrow$ NULL ▷ *Initialize best feasible solution*
2: $\bar{x} \leftarrow \tilde{x}$ ▷ *Initialize incumbent*
3: $w_i = 1,\ i \in M$ ▷ *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** $F^w(\bar{x}) = 0$ **then**
9: $x^* \leftarrow \bar{x}$
10: **break** ▷ *Found a feasible solution*
11: **if** $P = \emptyset$ **then** ▷ *Whether we have reached a local minimum*
12: $U \leftarrow \emptyset$ ▷ *Set of violated constraints*
13: **for** $i \in N : f_i(\bar{x}) > 0$ **do**
14: $w_i \leftarrow w_i + 1$ ▷ *Put more emphasis on satisfying this constraint*
15: $U \leftarrow U \cup i$
16: Update scores $s_j : a_{ij} \neq 0, i \in U, j \in N$
17: Update P
18: $i^* =$ random choice in U ▷ *Random violated constraint*
19: $j^* = \underset{j \in N : a_{i^* j} \neq 0}{\arg\max} s_j$ ▷ *Best move in this constraint*
20: **else**
21: $P^* =$ randomly choose up to 25 indices from P
22: $j^* = \underset{j \in P^*}{\arg\max} s_j$ ▷ *Best move among a random set of moves with $s_j > 0$.*
23: $\bar{x}_{j^*} \leftarrow v_{j^*}$ ▷ *Make the move*
24: $v_{j^*} = \texttt{Jump}_{j^*}(\bar{x}_{k \neq j^*})$ ▷ *Recompute jump value*
25: Update scores s_j of the neighboring variables
26: Update P
27: **return** x^*

– Choosing a move takes constant time because it consists of sampling a constant amount of move scores, running in worst-case $O(1)$.
– Computing the jump value takes time proportional to the number of constraints that the selected variable appears in, meaning that it runs in worst-case $O(\eta)$, where η is the maximum number of constraints that any variable appears in.
– Computing the updated scores for neighboring variables (line 25) requires iterating over all the constraints that the selected variable appears in and all the variables appearing in those constraints, meaning that it runs in worst-case $O(\eta\mu)$, where μ is the maximum number of variables appearing in a constraint.
– Computing the updated scores of the variables appearing in the currently unsatisfied constraints (line 16) requires iterating over those constraints (in the worst case, all constraints) and over all variables appearing in those constraints, meaning that it runs in worst-case $O(m\mu)$, where m is the number of constraints in the problem.

In summary, each iteration of FJ (line 8–26) has a worst-case $O(\eta + \eta\mu + m\mu)$ running time. Note that both of the last two terms are proportional to the total number of non-zero coefficients in the problem instance. However, in many problem instances, the number of variables in each constraint is small, and the number of violated constraints in a local minimum is small, which makes an iteration very fast in practice.

2.6 How to optimize using the original objective function

In the previous sections, we described a heuristic algorithm designed specifically for feasibility, disregarding the original objective function and quitting as soon as a feasible solution has been found (or any other termination criteria has been reached). However, Feasibility Jump can be easily extended to support an "improving" behavior, that is to keep searching for feasible solutions with better objective value after a first feasible solution has been found.

A simple way to take into account the original objective function is to consider an extended Lagrangian function:

$$O^q(x) + F^w(x),$$

where $O^q(x) = q \sum_{j \in N} c_j x_j$ is the original objective weighted by $q \geq 0$. The factor q is introduced to adjust the relative importance of the feasibility objective versus the original MIP objective. One drawback of this extended Lagrangian function is the possibility of having heavily unbalanced objective components, so that it would difficult to update q and w to maintain the correct balance between feasibility and optimality.

A more elegant way to look for solutions with better objective value is to consider an additional constraint of the form:

$$c^T x \leq c^T x^* - \theta, \tag{9}$$

where x^* is the current best feasible solution, and $\theta > 0$ is an appropriate cutoff tolerance. This cutoff constraint has been used in other MIP heuristic approaches (see, for example, [15]). The idea is that selecting a suitable θ will help the algorithm find a sequence of improving solutions. This constraint could be also immediately integrated in the current Feasibility Jump framework without changes to the algorithm, simply by adding (9) to the Lagrangian function $F^w(x)$.

Note that Feasibility Jump was not initially developed to provide improving solutions, and the objective function was not taken into account in any of the computational results below. However, this is a very interesting direction for future research.

3 Implementation

We have developed a C++ reference implementation of Feasibility Jump that is not dependent on any other MIP solver. This solver is available online[2] [24] 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

[2] https://github.com/SINTEF/feasibilityjump.

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);
```

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 ΔW or testing other metaheuristics.

Feasibility Jump can be used both as 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 [11], 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 within 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 to 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 after a very short interval of time (to make sure we inject the solutions found by FJ as soon as possible). In this callback function we check if any solution has been found by any of the two FJ threads, and, if so, copy it into Xpress by using the `addMIPsolution` method.

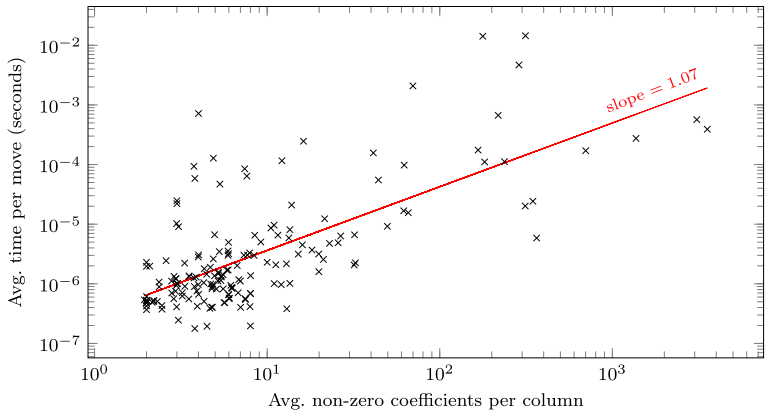

Fig. 2 Average computation time (in s) for each iteration of the while loop of Algorithm (2), for each problem instance in MIPLIB 2017

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 set, 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 comprehensive MIP solver. 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 an AMD Threadripper 3990x CPU running at 2.9 GHz with 128 GB of memory. The instances we used for testing belong to the MIPLIB 2017 benchmark set [19], a widely-used set of 240 mixed-integer problems of various size and difficulty. Our implementation of Feasibility Jump is able to find feasible solutions to 123 of these, when combining solutions from both the non-presolved and presolved versions of the problem. For 84 of these instances, solutions are found in both the non-presolved and the presolved case, while an additional 6 are found only using the non-presolved problem and 33 only using the presolved problem.

We first look at the average time it takes FJ to perform an iteration, that is an iteration of the while loop in line 7 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 take less than a microsecond.

The next computational results will consider three different solvers:

- `FJ`: Feasibility Jump, running on two threads, one with the non-presolved problem and one where the problem is presolved by Xpress (using the framework from Sect. 3);
- `XPR`: FICO Xpress Solver 8.14 with its default settings;
- `XPR+FJ`: Feasibility Jump integrated into Xpress as described in Sect. 3.

We compare these solvers on the time it takes to find the first feasible solution (Sect. 4.1), and the time it takes to prove optimality (Sect. 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 when increasing the computational effort with the current implementation. The local search framework could easily be extended to increase the ability to find feasible solutions, perhaps at the cost of more computational effort. 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).

Note that in both `XPR` and `XPR+FJ`, the MIP solver is configured to run on a single thread only, while FJ runs on two additional threads (as described in Sect. 3). In other words, we are simulating how FJ would influence a MIP solver if there were enough computational resources to run both programs at the same time. This gives a slight resource advantage to `XPR+FJ` when compared to `XPR`, but note that the Feasibility Jump heuristic is tuned to run for a very short amount of time, typically much less than the MIP solver (in these tests, the average running time of FJ was 0.6 s). Inside a comprehensive MIP solver, one would instead select between different heuristics with different trade-offs and tune FJ to run for as little time as necessary, taking problem characteristics into account. The objective of this paper is to show that integrating FJ in a MIP solver can indeed be beneficial, which has also been confirmed by the introduction of Feasibility Jump as a default heuristic in Xpress 9.0 [26].

4.1 Time to feasibility

We first compare `FJ`, `XPR`, and `XPR+FJ` on the time it takes to find the first feasible solution (TF). We run all solvers with a time limit of 1 min, although `FJ` will usually terminate much earlier even when not finding a feasible solution (the termination criterion is described in Sect. 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 Fig. 4, where we show the ratio between the TF of `XPR` and the TF 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 Feasibility Jump, `XPR+FJ` found a feasible solution to 6 more problems than `XPR`, and provided an average reduction of 25% on the time to first feasible solution. In about 10% of the instance, `XPR+FJ` found a feasible solution more than 10 times faster than `XPR` thanks to Feasibility

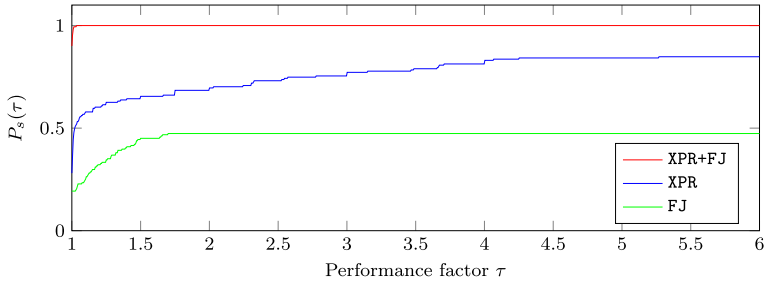

Fig. 3 Performance profile of time to first feasible solution (TF) for FJ, XPR and XPR+FJ on the MIPLIB 2017 benchmark set. Since XPR+FJ contains the best solution of either of the other two solvers, it is always faster

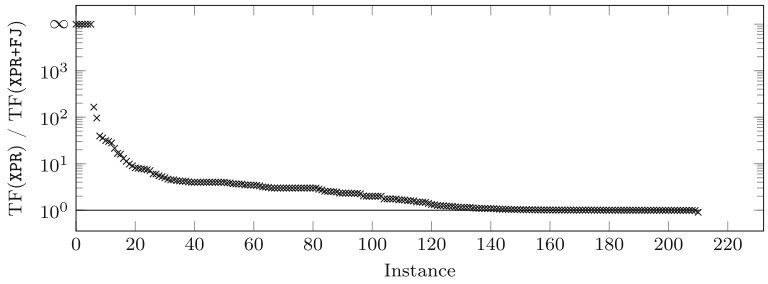

Fig. 4 Logscale plot of the ratio between the time to first feasible solution (TF) of XPR versus XPR+FJ on the MIPLIB 2017 benchmark set

Table 1 Categorization of the MIPLIB 2017 benchmark set instances based on the time to feasibility (TF) of FJ versus XPR

Category		# instances		
No solutions found		33	128	240
FJ found no solution but XPR did		84		
FJ found a solution after XPR		11		
FJ found a solution before XPR	during presolve	88	112	
	during root node	24		
	during branching	0		

In particular, when FJ is faster than XPR, we check whether FJ found a feasible solution (1) during presolve, (2) during root node (either while running the LP solver, other feasibility heuristics, or root cutting heuristics), or (3) during branching

Jump. Table 1 lists the number of instances on which FJ found a feasible solution before XPR, and which phase of the MIP solver was reached at that time.

4.2 Time to optimality

Sometimes, finding feasible solutions quickly may not be sufficient for improving the performance of a solver when considering the time to optimality. Therefore, it is of interest 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

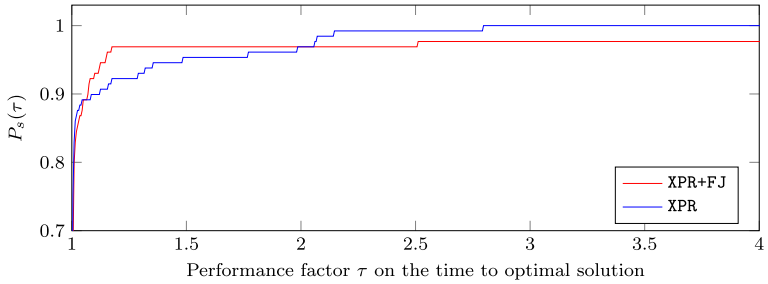

Fig. 5 Performance profile of time to optimality (TO) for XPR and XPR+FJ on the MIPLIB 2017 benchmark set

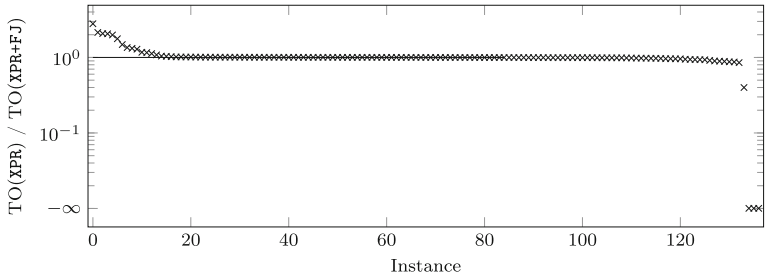

Fig. 6 Logscale plot of the ratio between the time to optimal solution (TO) of XPR versus XPR+FJ on the MIPLIB 2017 benchmark set

with a 10 min time limit, in which the solvers were able to find an optimal solution for 137 of the MIPLIB 2017 benchmark set instances. Similar to the previous section, Fig. 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 TO of XPR and TO 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 prove optimality (TO) 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). On average, we see a reduction on the solving time of about 3% on the instances where both solvers finish within the time limit. On 3 of the instances, XPR+FJ does not finish even though XPR does. Figure 7 shows how the best objective value improves over time for each of the solvers, for a set of instances where XPR+FJ proves optimality before XPR. In this figure, Feasibility Jump did not always provide a better bound than XPR, but Xpress could be taking advantage of the additional feasible solutions in other ways. Figure 8 shows similar plots for instances where solutions found by FJ did not impact the total running time of XPR. In this case, even though Feasibility Jump provided solutions with a better bound than the ones found by XPR, they were not useful to reduce the total running time.

We observe that Feasibility Jump has little impact on the time to optimality on a large number of instances, i.e., their TO ratios (Fig. 6) are very close to 1. This is not surprising, since FJ is just one of the many heuristics running in a state-of-the-art MIP

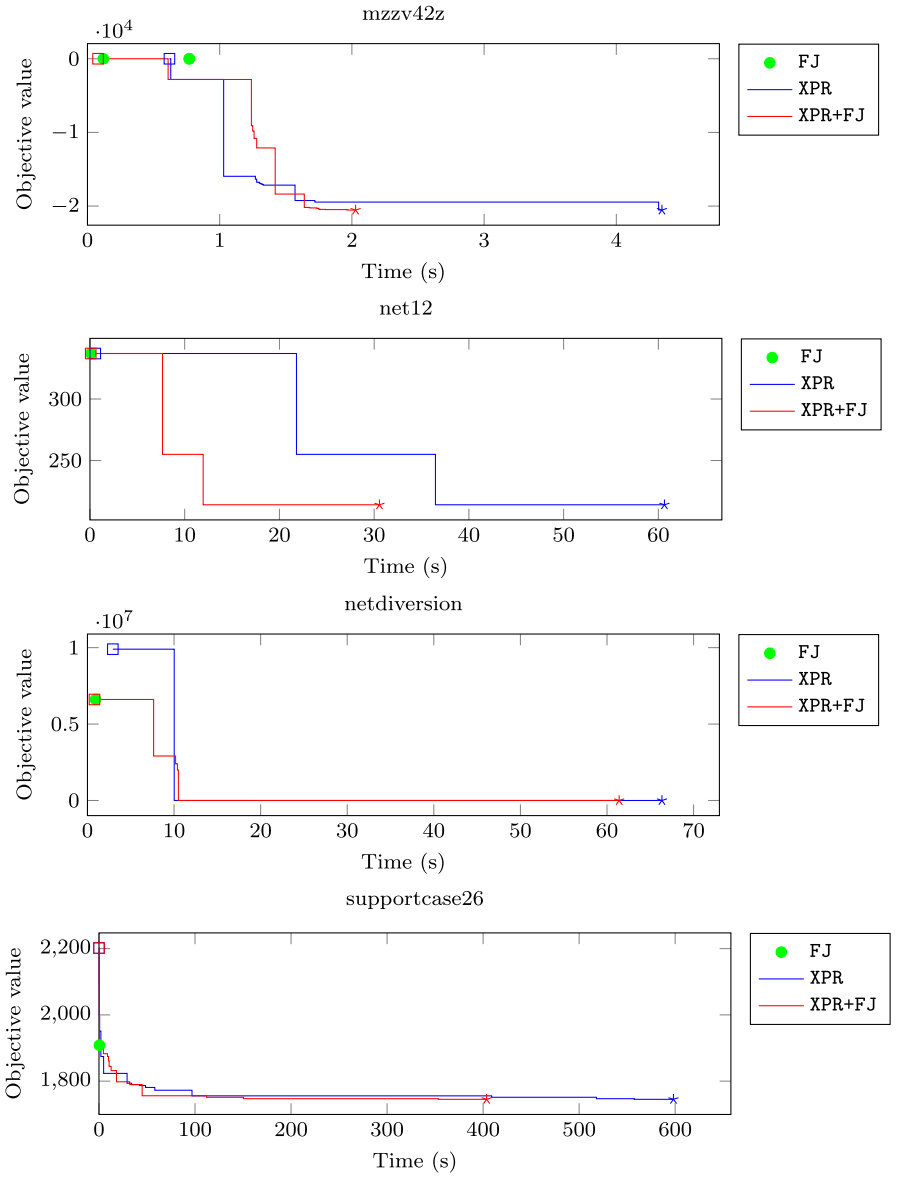

Fig. 7 Some examples of Feasibility Jump improving the solving time of Xpress. FJ solutions are shown as green circles, while the blue and red lines are the primal bounds for `XPR` and `XPR+FJ`, respectively. The squares mark the first feasible solution. The stars mark the time that the solution is known to be optimal (color figure online)

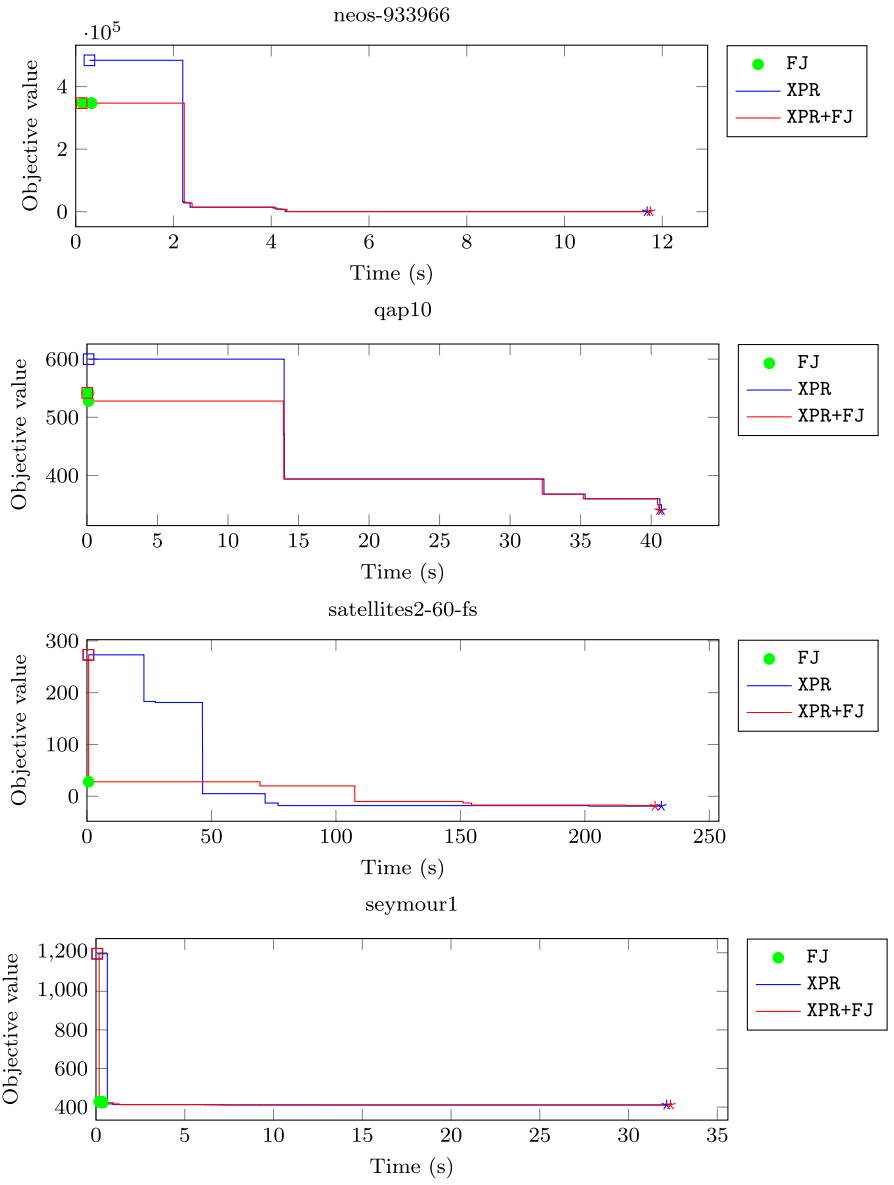

Fig. 8 Some examples of Feasibility Jump not improving (at least not significantly) the solving time of Xpress, even though it provided good solutions (better than the those found by Xpress) from the beginning. FJ solutions are shown as green circles, while the blue and red lines are the primal bound for `XPR` and `XPR+FJ`, respectively. The squares mark the first feasible solution. The stars mark the time that the solution is known to be optimal (color figure online)

solver such as Xpress. In fact, even if FJ finds a feasible solution quickly, the MIP solver may find a better solution before the branch-and-bound search starts, and the solution provided by FJ might then have little impact on the search. We measured that, in 22 cases, XPR+FJ had a better upper bound than XPR when branching starts (in 5 of those cases, XPR had no feasible solution at all). In 10 cases, XPR had a better upper bound than XPR+FJ when branching starts (in none of those cases XPR+FJ had no feasible solution at all). In the remaining 208 instances, the upper bound is the same in XPR and XPR+FJ at the time the branching search starts.

5 Conclusions and future directions

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

We have also integrated Feasibility Jump with the FICO Xpress Solver [11] and shown that, in addition to providing feasible solutions early in the solution process, supplying those solutions back to a MIP solver can also improve the time it takes to prove optimality.

An efficient C++ implementation of Feasibility Jump is available under an open-source license, and can be easily extended with custom, experimental designs. In particular, we refrained (on purpose) from "over-optimizing" the algorithm with sophisticated tuning, hoping to inspire the research community to come up with innovative solutions. Among others, we recognize the following directions as the most promising ones to consider for improving FJ:

- *Neighborhoods* The neighborhood of each variable is currently made of a single value v_j. In a previous implementation, we considered two more values (a positive or negative increment of the current incumbent value) and we saw little to no improvements. However, in general, we expect that having a larger neighborhood with different values could to be beneficial if easy to compute.
- *Metaheuristics* The algorithm is currently based on a guided search metaheuristic that favors constraints that remain violated across many assignments. Other metaheuristics, such as tabu search [23], or a combination of them, could provide different advantages.
- *Weight updates* The weights of the Lagrangian function are currently updated with a simple constant update function, but we expect that more sophisticated heuristics (perhaps even based on machine learning techniques) could increase their effectiveness in steering the algorithm towards a feasible solution. In particular, there can be local minima affected by so-called short cycles, whereby the algorithm repeatedly goes back to a previous infeasible assignment until the weights of the corresponding infeasible constraints have risen enough. We did not witness any

long cycles, but that might also be due to the randomness steps added in each local minimum. With that said, we did witness cases in which a lot of weight updates were necessary to get out of a local minimum. Then one could try to determine the minimum weight update required by those constraints to escape from the current local minimum.
- *Specializations* Feasibility Jump is currently completely problem agnostic, but several other primal MIP heuristics have been shown to take advantage of known constraint types (see, for example, [8]) or problem specific structures (see, for example, [18]).
- *Solution quality* The current implementation of FJ emphasizes finding a feasible solution. We experimented with giving the original objective function some importance through a simple weight update function. A more sophisticated version of FJ could include a better way to exploit existing feasible solutions (perhaps even provided by the MIP solver in which it is embedded) and incorporate a behavior usually found in improving heuristics.

Acknowledgements The authors would like to thank Carlo Mannino (SINTEF), Oddvar Kloster (SINTEF), and Volker Hoffmann (SINTEF) for their invaluable discussions and support during the development of Feasibility Jump. A special thanks goes to the organizers of the MIP 2022 Computational Competition (Gonzalo Muñoz, Timo Berthold, Yuri Faenza, Andrés Gómez). The authors also wish to thank the associate editor and three anonymous reviewers, whose insightful comments and careful proof-checks helped to improve the paper.

Author Contributions Both authors contributed equally to this work.

Funding Open access funding provided by SINTEF. The work has been funded by SINTEF (Norway) as a Strategical Self-funded Project.

Data availability The results presented in this paper are based on the MIPLIB 2017 dataset [19].

Code availability The code for reproducing the experiments of this paper is available open-source [24].

Declaration

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

References

1. Aarts, E., Lenstra, J.K.: Local Search in Combinatorial Optimization. Princeton University Press, Princeton (2003)

2. Achterberg, T.: Constraint integer programming. Ph.D. thesis, Berlin Institute of Technology (2007). http://opus.kobv.de/tuberlin/volltexte/2007/1611/
3. 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. Springer, Berlin, pp. 261–297 (2018). https://doi.org/10.1007/978-3-319-07124-4_2
4. Balas, E., Schmieta, S., Wallace, C.: Pivot and shift-a mixed integer programming heuristic. Discrete Optim. **1**(1), 3–12 (2004)
5. Berthold, T.: Primal Heuristics for Mixed Integer Programs. Diplomarbeit, Zuse Institute Berlin (ZIB) (2006)
6. Berthold, T.: Rens. Math. Program. Comput. **6**(1), 33–54 (2014)
7. Berthold, T., Hendel, G.: Shift-and-propagate. J. Heuristics **21**(1), 73–106 (2015). https://doi.org/10.1007/s10732-014-9271-0
8. Berthold, T., Hendel, G.: Shift-and-propagate. J. Heuristics **21**(1), 73–106 (2015)
9. Bertsekas, D.P.: Constrained Optimization and Lagrange Multiplier Methods. Academic Press, Cambridge (2014)
10. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Program. **102**(1), 71–90 (2005)
11. FICO Xpress Solver. https://www.fico.com/en/products/fico-xpress-solver
12. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. **104**(1), 91–104 (2005)
13. Fischetti, M., Lodi, A.: Local branching. Math. program. **98**(1), 23–47 (2003)
14. Fischetti, M., Monaci, M.: Exploiting erraticism in search. Oper. Res. **62**(1), 114–122 (2014)
15. Fischetti, M., Monaci, M.: Proximity search for 0–1 mixed-integer convex programming. J. Heuristics **20**(6), 709–731 (2014)
16. Fischetti, M., Sartor, G., Zanette, A.: MIP-and-refine matheuristic for smart grid energy management. Int. Trans. Oper. Res. **22**(1), 49–59 (2015)
17. Frangioni, A.: About Lagrangian methods in integer optimization. Ann. Oper. Res. **139**(1), 163–193 (2005)
18. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-driven fix-and-propagate heuristics for mixed integer programming. Math. Program. Comput. **11**(4), 675–702 (2019)
19. 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. Math. Program. Comput. (2021). https://doi.org/10.1007/s12532-020-00194-3
20. Gomory, R.E.: An algorithm for integer solutions to linear programs. Recent Adv. Math. Program. **64**(260–302), 14 (1963)
21. Lei, Z., Cai, S., Luo, C., Hoos, H.H.: Efficient local search for pseudo Boolean optimization. In: Li, C., Manyà, F. (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). https://doi.org/10.1007/978-3-030-80223-3_23
22. Lodi, A.: Mixed integer programming computation. In: 50 Years of Integer Programming 1958–2008, pp. 619–645. Springer (2010)
23. Lokketangen, A., Glover, F.: Solving zero-one mixed integer programming problems using tabu search. Eur. J. Oper. Res. **106**(2–3), 624–658 (1998)
24. Luteberget, B., Sartor, G.: Feasibility jump reference implementation. https://doi.org/10.5281/zenodo.7595090 (2023)
25. MIP 2022 Workshop. https://www.mixedinteger.org/2022/ (2022)
26. Salvagnin, D.: Personal Communication (2022)
27. Shapiro, J.F.: A survey of Lagrangean techniques for discrete optimization. Ann. Discrete Math. **5**, 113–138 (1979)
28. 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)
29. Song, J., Lanka, R., Dilkina, B., Yue, Y.: A general large neighborhood search framework for solving integer linear programs. Adv. Neural Inf. Process. Syst. **33**, 20012–20023 (2020)

30. Witzig, J., Gleixner, A.: Conflict-driven heuristics for mixed integer programming. INFORMS J. Comput. **33**(2), 706–720 (2021)
31. Wolsey, L.A.: Integer Programming. Wiley, Hoboken (2020)