Math. Prog. Comp. (2009) 1:201–222
DOI 10.1007/s12532-009-0007-3

FULL LENGTH PAPER

Feasibility pump 2.0

Matteo Fischetti · Domenico Salvagnin

Received: 27 October 2008 / Accepted: 1 September 2009 / Published online: 17 September 2009

Abstract Finding a feasible solution of a given mixed-integer programming (MIP) model is a very important $\mathcal{NP}$-complete problem that can be extremely hard in practice. Feasibility Pump (FP) is a heuristic scheme for finding a feasible solution to general MIPs that can be viewed as a clever way to round a sequence of fractional solutions of the LP relaxation, until a feasible one is eventually found. In this paper we study the effect of replacing the original rounding function (which is fast and simple, but somehow blind) with more clever rounding heuristics. In particular, we investigate the use of a diving-like procedure based on rounding and constraint propagation—a basic tool in Constraint Programming. Extensive computational results on binary and general integer MIPs from the literature show that the new approach produces a substantial improvement of the FP success rate, without slowing-down the method and with a significantly better quality of the feasible solutions found.

Keywords Mixed-integer programming · Primal heuristics · Constraint programming · Constraint propagation

Mathematics Subject Classification (2000) 90C11 · 90C27 · 90C57 · 90C59

1 Introduction

Finding a feasible solution of a given mixed-integer programming (MIP) model is a very important $\mathcal{NP}$-complete problem that can be extremely hard in practice.

M. Fischetti (✉)
DEI, University of Padova, Padova, Italy
e-mail: matteo.fischetti@unipd.it

D. Salvagnin
DMPA, University of Padova, Padova, Italy
e-mail: salvagni@math.unipd.it

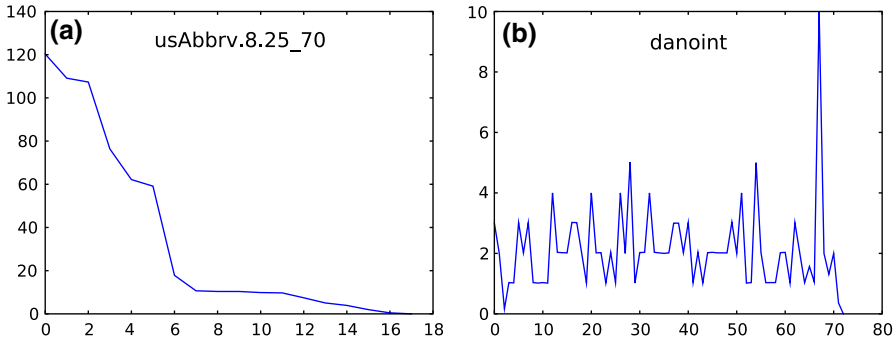

Fig. 1 Two very different FP behaviors (integrality distance vs. iterations)

Heuristics for general-purpose MIPs include [4–6,9,10,12,13,15–17,19,20,22,25,26], among others.

A heuristic scheme for finding a feasible solution to general MIPs, called *Feasibility Pump* (FP), was recently proposed by Fischetti, Glover, and Lodi [11] and further improved by Fischetti, Bertacco and Lodi [8] and Achterberg and Berthold [2]. The FP heuristic turns out to be quite successful in finding feasible solutions even for hard MIP instances, and is currently implemented in several optimization solvers, both commercial and open-source.

FP works with a pair of points x^* and $\tilde{x}$, with x^* feasible for the LP relaxation and $\tilde{x}$ integer, that are iteratively updated with the aim of reducing their distance as much as possible. To be more specific, one starts with any LP-feasible x^*, and initializes a (typically LP-infeasible) integer $\tilde{x}$ as the rounding of x^*. At each FP iteration, $\tilde{x}$ is fixed and one finds, through linear programming, the LP-feasible point x^* which is as close as possible to $\tilde{x}$. If the distance between x^* and $\tilde{x}$ is zero, then x^* is a MIP feasible solution, and the heuristic stops. Otherwise, $\tilde{x}$ is replaced by the rounding of x^* so as to further reduce their distance, and the process is iterated.

A main drawback of the basic FP scheme is its tendency to stall, in which case a random perturbation (or even a restart) step is performed in the attempt of escaping the local optimum. Figure 1 shows the different behavior of FP on two sample instances: while in Fig. 1a the distance between x^* and $\tilde{x}$ is rapidly brought to zero without the need of any perturbation/restart, in Fig. 1b FP exhibits a much less satisfactory behavior, with frequent restarts and perturbations that yield large oscillations of the distance function and hence reduce the probability of success of the method.

In the attempt of improving the FP success rate, we observe that FP can be interpreted as a clever way of rounding a sequence of fractional solutions of the LP relaxation, until a feasible one is eventually found. It is therefore quite natural to try to replace the original rounding operation (which is fast and simple, but somehow blind) with a more clever rounding heuristic.

In this paper we investigate the use of a diving-like procedure based on rounding and constraint propagation. The latter is a basic tool from Constraint Programming, which is used in modern Branch&Cut (B&C) codes for node preprocessing.

Extensive computational results on a large testbed of both binary and general integer MIPs from the literature show that this is a promising direction for improving significantly both the FP success rate and the quality of the feasible solutions found.

The paper is organized as follows. In Sect. 2 we provide a brief review of the basic FP scheme, while in Sect. 3 we present some Constraint Programming concepts and techniques useful to devise a "smarter" rounding strategy. In Sect. 4 we present our new proposal of propagation-based rounding and we discuss its properties. Some crucial implementation issues are explained in Sect. 5. Computational results are given in Sect. 6, with a detailed performance comparison of the new method with both the FP schemes of Fischetti, Bertacco and Lodi [8] and Achterberg and Berthold [2]. Conclusions are finally drawn in Sect. 7.

2 The feasibility pump

Suppose we are given a MIP:

$$\min\{c^T x : Ax \leq b,\ x_j \text{ integer } \forall j \in I\}$$

where A is an $m \times n$ matrix and $I \subseteq \{1, 2, \ldots, n\}$ is the index set of the variables constrained to be integer. Let $P = \{x : Ax \leq b\}$ be the corresponding LP relaxation polyhedron. We assume that system $Ax \leq b$ includes finite lower and upper bound constraints of the form

$$l_j \leq x_j \leq u_j \quad \forall j \in I$$

With a little abuse of terminology, we say that a point x is "integer" if x_j is integer for all $j \in I$ (thus ignoring the continuous components). Moreover, we call a point $x \in P$ LP-feasible. Finally, we define the distance between two given points x and $\tilde{x}$, with $\tilde{x}$ integer, as

$$\Delta(x, \tilde{x}) = \sum_{j \in I} |x_j - \tilde{x}|$$

again ignoring the contribution of the continuous components.

Starting from an LP-feasible point, the basic FP scheme generates two (hopefully converging) trajectories of points x^* and $\tilde{x}$, which satisfy feasibility in a complementary way: points x^* are LP-feasible, but may not be integer, whereas points $\tilde{x}$ are integer, but may not be LP-feasible.

The two sequences of points are obtained as follows: at each iteration (called *pumping cycle*), a new integer point $\tilde{x}$ is obtained from the fractional x^* by simply rounding its integer-constrained components to the nearest integer, while a new fractional point x^* is obtained as a point of the LP relaxation that minimizes $\Delta(x, \tilde{x})$. The procedure stops if the new x^* is integer or if we have reached a termination condition—usually, a time or iteration limit. Some care must be taken to avoid cycling, usually through random perturbations. An outline of this basic scheme is given in Fig. 2.

input : MIP $\equiv \min\{c^T x : x \in P,\ x_j \text{ integer } \forall j \in I\}$
output: a feasible MIP solution x^* (if found)

$x^* = \arg\min\{c^T x : x \in P\}$
while *not termination condition* **do**
  if x^* *is integer* **then return** x^*
  $\tilde{x} =$ `Round` (x^*)
  if *cycle detected* **then** `Perturb` $(\tilde{x})$
  $x^* =$ `LinearProj` $(\tilde{x})$
end

Fig. 2 Feasibility Pump—the basic scheme

According to this scheme, there are three essential FP ingredients:

Round This is the function called to transform an LP-feasible point into an integer one. The standard choice is to simply round each component x_j^* with $j \in I$ to the nearest integer $[x_j^*]$ (say), while leaving the continuous components unchanged.

LinearProj This function is somehow the inverse of the previous one, and is responsible for calculating an LP-feasible point x^* from the current integer $\tilde{x}$. A standard choice is to solve the following LP:

$$x^* = \arg\min\{\Delta(x, \tilde{x}) : x \in P\}$$

In the binary case the distance function $\Delta(\cdot, \tilde{x})$ can easily be linearized as

$$\Delta(x, \tilde{x}) = \sum_{j \in I : \tilde{x}_j = 1} (1 - x_j) + \sum_{j \in I : \tilde{x}_j = 0} x_j$$

In the general integer case, however, the linearization requires the use of additional variables and constraints to deal with integer-constrained components $\tilde{x}_j$ with $l_j < \tilde{x}_j < u_j$. To be more specific, the distance function reads

$$\Delta(x, \tilde{x}) = \sum_{j \in I : \tilde{x}_j = u_j} (u_j - x_j) + \sum_{j \in I : \tilde{x}_j = l_j} (x_j - l_j) + \sum_{j \in I : l_j < \tilde{x}_j < u_j} d_j$$

with the addition of constraints

$$d_j \geq x_j - \tilde{x}_j \quad \text{and} \quad d_j \geq \tilde{x}_j - x_j$$

Perturb This function is used to perturb an integer point when a cycle is detected. The standard choice is to apply a weak perturbation if a cycle of length one is detected, and a strong perturbation (akin to a restart) otherwise. More details can be found in [8,11].

2.1 The general integer case

The above scheme, although applicable "as is" to the general integer case, is not particularly effective in that scenario. This is easily explained by observing that, for general integer variables, one has to decide not only the rounding direction (up or down), as for binary variables, but also the new value. For the same reasons, also the random perturbation phases must be designed much more carefully.

To overcome some of these difficulties, an extended scheme suited for the general integer case has been presented in Bertacco, Fischetti and Lodi [8]. The scheme is essentially a three-staged approach. In Stage 1, the integrality constraint on the general-integer variables is relaxed, and the pumping cycles are carried out only on the binary variables. Stage 1 terminates as soon as a "binary feasible" solution is found, or some termination criterion is reached. The rationale behind this approach is trying to find quickly a solution which is feasible with respect to the binary components, in the hope that the general integer ones will be "almost integer" as well. In Stage 2, the integrality constraint on the general-integer variables is restored and the FP scheme continues. If a feasible solution is not found in Stage 2, a local search phase (Stage 3) around the rounding of a "best" $\tilde{x}$ is triggered as last resort, using a MIP solver as a black box heuristic.

2.2 The objective feasibility pump

A drawback of the feasibility pump scheme presented in [8,11] is the often poor quality of the feasible solutions found. This is easily explained by the fact that the original scheme uses the objective function of the problem only in the first iteration. This issue has been addressed in several different ways in the literature.

Fischetti, Glover and Lodi [11] propose to add an objective cutoff of the form $c^T x \leq UB$, where UB is dynamically updated whenever a new incumbent is found. In particular UB consists in a convex combination of the values of the optimal LP relaxation and of the current incumbent.

In Bertacco, Fischetti and Lodi [8], the use of improvement heuristics based on *local search*, such as *local branching* [12] or *RINS* [9], is presented.

A different approach, called *objective feasibility pump*, is developed by Achterberg and Berthold [2], who propose to use in the `LinearProj` phase a convex combination of the original objective with the distance function $\Delta(x, \tilde{x})$. In particular, the objective function of the LPs is given by

$$\Delta_\alpha(x, \tilde{x}) = \frac{1-\alpha}{||\Delta||}\Delta(x, \tilde{x}) + \frac{\alpha}{||c||}c^T x$$

where $||\cdot||$ is the Euclidean norm and $\alpha \in [0, 1]$. At each iteration i, coefficient α_i is geometrically decreased by a factor $\phi < 1$, i.e., $\alpha_{i+1} = \phi\alpha_i$. This modification calls for an update of the cycle detection algorithm. Indeed, while in the original scheme if we visit the same integer point $\tilde{x}$ twice we know for sure that we are in a cycle, this is not the case in the modified scheme, because the objective function Δ_α has

changed in the meantime. Thus, at each iteration the pair $(\tilde{x}_i, \alpha_i)$ is stored and a cycle is detected if there exists two iteration i and j such that the corresponding pairs $(\tilde{x}_i, \alpha_i)$ and $(\tilde{x}_j, \alpha_j)$ satisfy $\tilde{x}_i = \tilde{x}_j$ and $|\alpha_i - \alpha_j| \leq \delta_\alpha$, for a given threshold δ_α.

3 Constraint propagation

Constraint propagation is a very general concept that appears under different names in different fields of computer science and mathematical programming. It is essentially a form of inference which consists in explicitly forbidding values—or combinations of values—for some problem variables.

To get a practical constraint propagation system, two questions need to be answered:

- What does it mean to propagate a single constraint? In our particular case, this means understanding how to propagate a general linear constraint with both integer and continuous variables. The logic behind this goes under the name of bound strengthening [27,30] (a form of preprocessing) in the integer programming community.
- How do we coordinate the propagation of the whole set of constraints defining our problem?

In the remaining part of this section we will first describe bound strengthening (Sect. 3.1) and then we will describe our constraint propagation system (Sect. 3.2), following the propagator-based approach given by Schulte and Stuckey in [32].

3.1 Bound strengthening

Bound strengthening [1,18,21,27,30] is a preprocessing technique that, given the original domain of a set of variables and a linear constraint on them, tries to infer tighter bounds on the variables. We will now describe the logic behind this technique in the case of a linear inequality of the form:

$$\sum_{j \in C^+} a_j x_j + \sum_{j \in C^-} a_j x_j \leq b$$

where C^+ and C^- denote the index set of the variables with positive and negative coefficients, respectively. We will assume that *all* variables are bounded, with lower and upper bounds denoted by l_j and u_j, respectively. Simple extensions can be made to deal with unbounded (continuous) variables and equality constraints.

The first step to propagate the constraint above is to compute the minimum ($L_{\min}$) and maximum ($L_{\max}$) "activity level" of the constraint:

$$L_{\min} = \sum_{j \in C^+} a_j l_j + \sum_{j \in C^-} a_j u_j$$

$$L_{\max} = \sum_{j \in C^+} a_j u_j + \sum_{j \in C^-} a_j l_j$$

Now we can compute updated upper bounds for variables in C^+ as

$$\overline{u}_j = l_j + \frac{b - L_{\min}}{a_j} \tag{1}$$

and updated lower bounds for variables in C^- as

$$\overline{l}_j = u_j + \frac{b - L_{\min}}{a_j} \tag{2}$$

Moreover, for variables constrained to be integer we can also apply the floor $\lfloor \cdot \rfloor$ and ceiling $\lceil \cdot \rceil$ operators to the new upper and lower bounds, respectively.

It is worth noting that no propagation is possible in case the maximum potential activity change due to a single variable, computed as

$$\max_j \{|a_j(u_j - l_j)|\}$$

is not greater than the quantity $b - L_{\min}$. This observation is very important for the efficiency of the propagation algorithm, since it can save several useless propagator calls. Finally, equations (1) and (2) greatly simplify in case of binary variables, and the simplified versions should be used in the propagation code for the sake of efficiency.

3.2 Propagation algorithm

Constraint propagation systems [29,31,32] are built upon the basic concepts of *domain*, *constraint* and *propagator*.

A *domain* D is the set of values a solution x can possibly take. In general, x is a vector $(x_1, \ldots, x_n)$ and D is a Cartesian product $D_1 \times \cdots \times D_n$, where each D_i is the domain of variable x_i. We will denote the set of variables as X.

A *constraint* c is a relation among a subset $var(c) \subseteq X$ of variables, listing the tuples allowed by the constraint itself. This definition is of little use from the computational point of view; in practice, constraint propagation systems implement constraints through *propagators*.

A propagator p implementing[1] a constraint c is a function that maps domains to domains and that satisfies the following conditions:

- p is a *decreasing* function, i.e., $p(D) \subseteq D$ for all domains. This guarantees that propagators only remove values.
- p is a *monotonic* function, i.e., if $D_1 \subseteq D_2$, then $p(D_1) \subseteq p(D_2)$.
- p is *correct* for c, i.e., it does not remove any tuple allowed by c.
- p is *checking* for c, i.e., all domains D corresponding to solutions of c are *fixpoints* for p, i.e., $p(D) = D$. In other words, for every domain D in which all variables

[1] In general, a constraint c is implemented by a collection of propagators; we will consider only the case where a single propagator suffices.

```
input  : a domain D
input  : a set P_f of (fixpoint) propagators p with p(D) = D
input  : a set P_n of (non-fixpoint) propagators p with p(D) ⊆ D
output: an updated domain D
1 Q = P_n
2 R = P_f ∪ P_n
3 while Q not empty do
4     p = Pop (Q)
5     D = p(D)
6     K = set of variables whose domain was changed by p
7     Q = Q ∪ {q ∈ R : var(q) ∩ K ≠ ∅}
8 end
9 return D
```

Fig. 3 Basic propagation engine

involved in the constraint are fixed and the corresponding tuple is valid for c, we must have $p(D) = D$.

A *propagation solver* for a set of propagators R and some initial domain D finds a fixpoint for propagators $p \in R$.

A basic propagation algorithm is outlined in Fig. 3. On input, the propagator set is partitioned into the sets P_f and P_n, depending on the known fixpoint status of the propagators for domain D—this feature is essential for implementing efficient incremental propagation. The algorithm maintains a queue Q of pending propagators (initially P_n). At each iteration, a propagator p is popped from the queue and executed. At the same time the set K of variables whose domains have been modified is computed and all propagators that share variables with K are added to Q (hence they are *scheduled* for execution).

The complexity of the above algorithm is highly dependent on the domain of the variables. For integer (finite domain) variables, the algorithm terminates in a finite number of steps, although the complexity is exponential in the size of the domain (it is however polynomial in the pure binary case, provided that the propagators are also polynomial, which is usually the case). For continuous variables, this algorithm may not converge in a finite number of steps, as shown in the following example (Hooker [21]):

$$\begin{cases} \alpha x_1 - x_2 \geq 0 \\ -x_1 + x_2 \geq 0 \end{cases} \tag{3}$$

where $0 < \alpha < 1$ and the initial domain is $[0, 1]$ for both variables: it can be easily seen that the upper bound on x_1 converges only asymptotically to zero.

4 The new FP scheme

As already observed, the rounding function described in the original feasibility pump scheme has the advantage of being extremely fast and simple, but it has also the

```
input  : a domain D and a set R of propagators
input  : a fractional vector x*
output: an integer vector x̃
1 I = index set of integer variables
2 while I not empty do
3     j = Choose (I)
4     D = Propagate (D, R, x_j = [x*_j])
5     I = index set of integer non-fixed variables
6 end
7 read x̃ from D
8 return x̃
```

Fig. 4 New rounding procedure

drawback of completely ignoring the linear constraints of the model. While it is true that the linear part is taken into account in the LinearProj phase, still the "blind" rounding operation can let the scheme fail (or take more iterations than necessary) to reach a feasible solution, even on trivial instances.

Suppose, for example, we are given a simple binary knapsack problem. It is well known that an LP optimal solution has only one fractional component, corresponding to the so-called critical item. Unfortunately, a value greater than 0.5 for this component will be rounded up to one, thus resulting in an infeasible integer solution (all other components will retain their value, being already zero or one). A similar reasoning can also be done for set covering instances, where finding a feasible solution is also a trivial matter but can require several pumping cycles to an FP scheme.

Our proposal is to merge constraint propagation with the rounding phase, so as to have a more clever strategy that better exploits information about the linear constraints. This integration is based on the following observation: rounding a variable means temporarily fixing it to a given value, so we can, in principle, propagate this temporary fixing before rounding the remaining variables. This is very similar to what is done in modern MIP solvers during diving phases, but without the overhead of solving the linear relaxations. A sketch of the new rounding procedure is given in Fig. 4. The procedure works as follows. At each iteration, a non-fixed variable $x_j \in I$ is selected and rounded to $[x_j^*]$. The new fixing $x_j = [x_j^*]$ is then propagated by the propagation engine. When there are no non-fixed variables left, the domain D_j of every variable x_j with $j \in I$ has been reduced to a singleton and we can "read" the corresponding $\tilde{x}$ from D.

With respect to the original "simple" rounding scheme, it is worth noting that:

- When rounding a general integer variable, one can exploit the reduced domain derived by the current propagation. For example, if the domain of a variable y with fractional value 6.3 is reduced to [8, 10], then it is not clever to round it to values 6 or 7 (as simple rounding would do), but value 8 should be chosen instead.
- On a single iteration, the new rounding scheme strictly dominates the original one, because a feasible simple rounding cannot be ruled out because of constraint

propagation. On the other hand, as outlined above, there are cases where the new scheme (but not the original one) can find a feasible solution in just one iteration.
- There is no dominance relation between the two rounding schemes as far as the "feasibility degree" of the resulting $\tilde{x}$ is concerned. This is because constraint propagation can only try to enforce feasibility, but there is no guarantee that the resulting rounded vector will be "more feasible" than the one we would have obtained through simple rounding.
- There is also no dominance relation at the feasibility pump level, i.e., we are not guaranteed to find a feasible solution in fewer iterations.
- As in diving, but differently from standard rounding, the final $\tilde{x}$ depends on the order in which we choose the next variable to round (and also on the order in which we execute propagators); this ordering can have an impact on the performance of the overall scheme, as we will see in Sect. 6.2. In addition, a dynamic ordering policy can be beneficial for reducing cycling, without resorting to random pertubations.

Finally, while in a constraint propagation system it is essential to be able to detect the infeasibility of the final solution as soon as possible (mainly to reduce propagation overhead, but also to get smaller search trees or to infer smaller infeasibility proofs), in the FP application we always need to bring propagation to the end, because we need to choose a value for all integer variables. In our implementation, failed propagators leading to infeasible solutions are simply ignored for the rest of the current rounding, but the propagation continues in the attempt of reducing the degree of infeasibility of the final solution found.

5 Implementation

While for binary MIPs the implementation of an FP scheme is quite straightforward, in the general integer case some care must be taken to get a satisfactory behavior of the algorithm. In addition, the overhead of constraint propagation can be quite large, mainly if compared with the extremely fast simple rounding operation. Hence we need a very efficient implementation of the whole constraint propagation system, including both the single propagators and the overall propagation engine. In the rest of the section we will describe important implementation details concerning these issues.

5.1 Optimizing FP restarts

Random perturbations, and in particular restarts, are a key ingredient of the original FP scheme even for binary MIPs. They are even more important in the general integer case, where the FP is more prone to cycling. As a matter of fact, our computational experience shows that just adapting the original restart procedure for binary MIPs to the general integer case results in a considerably worse overall behavior.

The FP implementation of Bertacco, Fischetti and Lodi [8] uses a quite elaborated restart scheme to decide how much a single variable has to be perturbed, and how many variables have to be changed. A careful analysis of the original source code

[7] shows that a single variable x_j is perturbed by taking into account the size of its domain: if $u_j - l_j < M$ with a suitable large coefficient M, then the new value is picked randomly within the domain. Otherwise, the new value is picked uniformly in a large neighborhood around the lower or upper bound (if the old value is sufficiently close to one of them), or around the old value.

The number of variables to be perturbed, say *RP*, is also very important and has to be defined in a conservative way. According to [7], *RP* is changed dynamically according to the frequency of restarts. In particular, *RP* decreases geometrically with constant 0.85 at every iteration in which restarts are not performed, while it increases linearly (with a factor of 40) on the others. Finally, *RP* is bounded by a small percentage (10%) of the number of general integer variables, and the variables to be changed are picked at random at every restart.

5.2 Optimizing propagators

The default linear propagator is, by design, general purpose: it must deal with all possible combinations of variable bounds and variable types, and can make no assumption on the coefficients (sign, distribution, etc.).

While it is true that constraints of this type are really used in practice, nevertheless they are often only a small part of the MIP model. As a matter of fact, most linear constraints used in MIP modeling have some specific structure that can be exploited to provide a more efficient propagation [1].

For the reasons above, we implemented specialized propagators for several classes of constraints, and an automated analyzer for assigning each constraint to the appropriate class. In particular, we implemented specialized propagators for

- *Knapsack constraints*, i.e., constraints with positive coefficients involving only bounded variables. These assumptions allow several optimization to be performed. Note that this class of constraints includes both covering and packing constraints.
- *Cardinality constraints*, i.e., constraints providing lower and/or upper bounds on the sum of binary variables. Propagation can be greatly simplified in this case, since we just need to count the number of variables fixed to zero or one. This class includes classical set covering/packing/partioning constraints.
- *Binary logic constraints*, i.e., linear constraints expressing logical conditions between two binary variables, such as implications and equivalences.

According to our computational experience, the above specializations can lead to a speedup of up to one order of magnitude in the propagation phase.

5.3 Optimizing constraint propagation

A propagation engine should exploit all available knowledge to avoid unnecessary propagator execution [32]. Moreover, when the engine invokes the execution of a propagator, it should provide enough information to allow the most efficient algorithms to be used. Thus the constraint system must support

- *Propagator states*, needed to store data structures for incremental propagation.

- *Modification information* (which variables have been modified and how), needed to implement sub-linear propagators and to implement more accurate fixpoint considerations (see [29,31,32]).

A simple and efficient way for supporting these services is to use the so-called *advisors*. Advisors were introduced by Lagerkvist and Schulte in [24] and are used in the open-source constraint programming solver Gecode [14]. They are responsible for updating the propagator state in response to domain changes, and to decide whether a propagator needs to be executed. Each advisor is tied to a (variable, propagator) pair, so that the most specialized behavior can be implemented. Modification information is provided by *propagation events*, see [32]. More details about advised propagation are available in [24].

Finally, since the presence of continuous variables or integer variables with huge domains can make the complexity of propagation too high, we impose a small bound on the number of times the domain of a single variable can be tightened (e.g., 10): when this bound is reached, the domain at hand stops triggering propagator executions within the scope of the current rounding.

6 Computational experiments

In this section we report computational results to compare the two rounding schemes (with and without propagation). Our testbed consists of 78 instances from MIPLIB 2003 [3] and [9,28], which are the same instances used in [8]. One instance (`stp3d`) was left out because even the first LP relaxation was very computationally demanding, while instances `momentum*` were left out because of numerical problems.

All algorithms, including those of the propagation engine, were implemented in C++. We used a commercial LP solver (ILOG Cplex 11.2 [23]) to solve the linear relaxations and to preprocess our instances. All tests have been run on an Intel Core2 Q6600 system (2.40 GHz) with 4GB of RAM.

We denote by `std` the original FP based on simple rounding and with `prop` the one using propagation-based rounding. Within `prop`, variables are ranked in order of increasing fractionality, with binary variables always preceding the general integer ones. This corresponds to the "smallest domain first" rule in Constraint Programming, and it is also common wisdom in Integer Programming. As a matter of fact, binary variables very often model the most important decisions in the model, and their fixing usually triggers more powerful propagations. To reduce cycling, after the 10th iteration we randomly swap 10% of the variables in the permutation giving the ranking order.

We compared the two rounding schemes in two different scenarios:

`alone` To evaluate the effect of the new rounding scheme when the feasibility pump is used as a standalone primal heuristic. The feasibility pump was run with standard parameters, as described in [8], except for the time limit on Stage 3, that was set equal to the overall computing time spent in Stages 1 and 2 so as to avoid that Stage 3 dominates the entire run (for `rococo*` instances we did not force the use of the primal simplex method for reoptimizing the LPs, because it was much slower than using Cplex defaults).

`embed` To evaluate the feasibility pump as a primal heuristic embedded in a general purpose B&C code. In this scenario, it is not typically worthwhile to spend a large computing time on a single heuristic, hence some parameter changes are needed. In particular, in this setting we lower to 20 the iteration limit and we skip the very expensive Stage 3 (that is also skipped in other FP implementations, for example in SCIP [1]).

Moreover, since we want to test the ability of FP of finding good-quality solutions, for each instance in our testbed we generated a set of instances with increasing difficulty through the addition of an upper bound constraint of the type:

$$c^T x \leq UB := z^*(1 + \beta)$$

where z^* is the best known solution from the literature, and $\beta \in \{0.1, 1.0, +\infty\}$ is the relative allowed optimality gap.

The performance figures used to compare the algorithms are:

success The ratio between the number of solutions found and the number of trials. The feasibility pump being a primal heuristic, this is the most important figure for benchmarking.

iter Number of iterations needed to find a solution.

time Time needed to find a solution.

s3 The percentage of times in which FP reached Stage 3 (of course, this applies only to the `alone` scenario). Stage 3 is usually quite expensive and is somewhat external to the FP scheme, so the lower the value of this figure the better.

As already discussed, random perturbations are a fundamental ingredient of the FP scheme, so it is not completely fair to compare different variants on a single run. Indeed, even if the random seed may not be crucial for the success of the FP on a given instance, it may still greatly affect the number of iterations needed to find a feasible solution. For these reasons, for each instance we ran 10 times each FP variant, using different seeds for the internal pseudo-random number generator, and we used average results for the comparison of the different algorithms. In particular, for each instance we computed the arithmetic means for `iter` and `time`, while an algorithm is considered successful if the number of solutions found is greater than 6 (out of 10). A typical behavior is illustrated in Fig. 5, for instance `air04`, where the coordinates of each point are the number of iterations needed to find a solution for `std` (x axis) and `prop` (y axis), for different seeds. According to the figure, `std` required a number of iterations ranging from 4 to 49, whereas for `prop` the range was 3 to 25. For this instance, `prop` required fewer iterations than `std` in 8 out of 10 runs (points below the line), while the opposite happened twice (points above the line).

Aggregated results are reported in Table 1. The structure of the table is as follows. On the vertical axis, we have the performance figures (success, iter, time, and s3), grouped by upper bound UB on the solution cost. On the horizontal axis we have results for `std` and `prop` and the percentage improvement (`impr%`) achieved by `prop` with respect to `std` (a positive percentage meaning that the new method based on propagation outperformed the standard one), for the two scenarios `alone` and `embed`. To assess the statistical significance of the performance difference for the

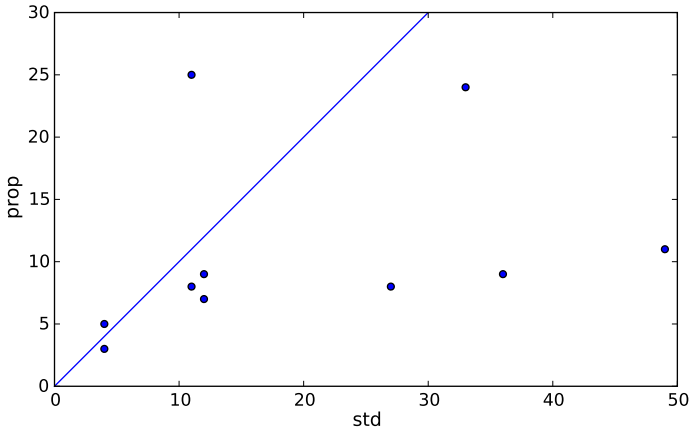

Fig. 5 Effect of the random seed on the number of iterations needed to find a feasible solution on instance `air04`, for both rounding schemes

Table 1 Aggregated testbed results

UB	Figure	Alone					Embed				
		Std	Prop	Impr%	*L*	*U*	Std	Prop	Impr%	*L*	*U*
None	Success	87	90	3	−1	7	49	68	**39**	22	57
	Iter	27	10	**62**	50	72	9	5	**44**	33	53
	Time	1.05	0.82	**21**	7	34	0.39	0.39	0	−5	8
	s3	18	14	21	−2	45	–	–	–	–	–
100%	Success	85	88	**5**	0	10	32	55	**72**	43	101
	Iter	87	24	**72**	60	81	12	7	**43**	32	52
	Time	2.22	1.46	**34**	18	47	0.42	0.42	0	−7	6
	s3	36	27	**25**	7	43	–	–	–	–	–
10%	Success	50	59	**18**	3	32	17	21	23	−3	49
	Iter	285	201	**30**	11	44	17	14	**16**	7	24
	Time	5.80	7.75	**−34**	−68	−6	0.47	0.56	**−19**	−32	−7
	s3	71	56	**20**	8	32	–	–	–	–	–

A positive impr% means that the new method (`prop`) outperformed the standard one (`std`). In columns impr%, boldface entries passed a 95% significance test. Iter and time are given as geometric means

two rounding schemes, we also report the 95% confidence intervals for the entries of columns `impr%` (columns L and U). In addition, the entries of Table 1 (columns `impr%`) corresponding to a statistically significant change are highlighted in boldface.

According to Table 1, the typical performance of the new method (`prop`) is significantly better than the previous one (`std`), in particular for the `embed` scenario where the success rate improves by 39% when no upper bound is imposed, and by 72% and 23% for $UB = 100\%$ and 10%, respectively. Our statistical tests confirm that this increase is statistically significant, with the exception of case $UB = 10\%$ where the number of successful instances is too small.

For the alone scenario the success rate improvement is less impressive (3, 5 and 18% in the three UB scenarios), due to the large number of iterations allowed and to the presence of the time-consuming Stage 3, both of which tend to mask the improved behavior of prop. However, even in the alone scenario, prop requires much fewer iterations, which has a positive effect on the quality of the solutions found—this very important aspect will be discussed in the next subsection. In addition, prop needs to resort to Stage 3 less frequently.

As to computational efficiency, the two methods requires comparable computing time: in the alone scenario, prop is 20–30% faster than std for $UB \geq 100\%$, and about 30% slower for $UB = 10\%$, whereas in the embed scenario the two methods are equivalent for $UB \geq 100\%$ while prop is about 20% slower than std for $UB = 10\%$ (which is acceptable, also in view of the improved success rate).

6.1 Impact of propagation on the solution quality

So far, we have evaluated the FP capability of finding good-quality solutions by adding an artificial upper bound constraint. This procedure has the advantage of producing solutions of "guaranteed" quality and it is also a quick method for generating difficult test instances. However, this is not always a viable option, since we may not have a reasonable upper bound to impose. In this situation, it is appropriate to just compare the objective value of the solutions found by the two methods, with no upper bound constraint added.

As already mentioned, the state-of-the-art FP scheme for producing good-quality solutions is the objective FP of Achterberg and Berthold [2]. Thus, we compared the effect of the new rounding scheme when implemented inside both versions of the FP (original and objective), corresponding to setting parameter α of Sect. 2.2 to 0 and to 1, respectively.

Detailed results for all instances in our testbed are presented in Table 2. We compared four FP variants, namely the Bertacco, Fischetti, Lodi [8] FP version ($\alpha = 0$) and the Achterberg, Berthold [2] objective FP ($\alpha = 1$), both with (prop) and without (std) propagation. For each instance and for each FP variant, we report the average (out of 10 runs with different random seeds) of four figures: objective function value (obj), percentage gap with respect to the best known solution (gap%), iterations (iter), and computing time (time). Averages in columns obj and gap% refer to the successful runs only (when different from 10, this number is reported, in parenthesis, in column obj). In boldface, we highlight the FP variant producing the best (average) solution. Geometric means are reported in the last line of Table 2.

As anticipated, even in the non-objective FP version ($\alpha = 0$) propagation proved quite effective in improving the solution quality, the average gap being decreased from 77.2 to 57.6%. This is mainly due to the greatly reduced number of iterations and restarts, that has a positive effect in keeping the final solution closer the initial (LP optimal) one.

As to the objective FP version ($\alpha = 1$), the std version produces solutions with a 52.4% average gap, i.e., only 10% better than the original FP with prop. The average gap is further improved to 35.5% when propagation is added. Out of 78 instances,

Table 2 Solution quality comparison with and without propagation, for both the original ($\alpha = 0$) and objective ($\alpha = 1$) FP versions

Problem	Std ($\alpha = 0$)				Prop ($\alpha = 0$)				Std ($\alpha = 1$)				Prop ($\alpha = 1$)			
	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time
10teams	$1,006.5^{(8)}$	8.9	245	17.6	1,020.0	10.4	78	5.3	$960.9^{(9)}$	**4.0**	191.9	12.4	1,018.0	10.2	74	4.9
a1c1s1	25,342.1	120.3	12	0.5	21,060.3	83.1	3	0.3	18,821.0	63.6	46.6	0.2	17,486.9	**52.0**	33	0.1
aflow30a	4,257.0	267.6	15	0.0	2,923.4	152.5	7	0.1	2,986.7	157.9	36.2	0.0	2,180.7	**88.3**	11	0.1
aflow40b	5,253.6	349.8	17	0.0	4,189.7	258.7	11	0.1	3,742.8	220.4	43.7	0.1	2,004.0	**71.6**	12	0.1
air04	59,797.6	6.5	20	35.2	59,861.9	6.6	8	16.3	57,741.2	2.9	55.2	95.7	57,359.4	**2.2**	10	16.3
air05	32,154.5	21.9	11	7.7	29,954.8	13.6	2	2.3	28,012.0	6.2	21.4	10.6	27,432.5	**4.0**	6	3.0
arki001	$7.781E6^{(1)}$	2.6	604	24.9	$7.758E6^{(1)}$	2.3	605	28.4	$7.750E6^{(3)}$	**2.2**	608.5	23.5	$7.748E6^{(3)}$	**2.2**	606	24.8
atlanta-ip	$152.3^{(7)}$	69.2	200	82.8	138.7	54.1	21	30.9	$172.0^{(1)}$	91.1	540.6	179.7	111.3	**23.7**	25	43.4
cap6000	$-2.376E6$	3.1	2	0.0	$-2.443E6$	**0.4**	1	0.1	$-2.153E6$	12.2	39.3	0.1	$-2.424E6$	1.1	31	0.2
dano3mip	1,000.0	45.7	4	20.2	1,000.0	45.7	1	17.6	741.4	**8.0**	70	281.2	795.6	15.9	69	361.5
danoint	$80.5^{(8)}$	22.6	94	0.5	87.3	32.9	41	0.3	76.3	**16.1**	69.8	1.5	80.7	22.8	93	2.7
disctom	$-5,000.0$	**0.0**	10	7.3	$-5,000.0$	**0.0**	2	4.3	$-5,000.0$	**0.0**	12.6	6.7	$-5,000.0$	**0.0**	3	4.7
ds	–	–	242	1,806.3	1,000.4	**136.9**	55	449.5	–	–	302.4	1,803.9	1,172.0	177.5	87	671.8
fast0507	349.0	100.6	2	14.0	307.5	76.7	1	15.8	182.2	**4.7**	5.4	26.2	182.8	5.1	4	18.7
fiber	$2.836E6$	598.7	5	0.0	$1.109E6$	**173.2**	1	0.1	4.015E6	889.0	32.5	0.0	$1.447E6$	256.4	18	0.1
fixnet6	17,055.8	328.2	10	0.0	4,596.0	15.4	1	0.1	11,893.0	198.6	34	0.0	4,441.0	**11.5**	15	0.1
gesa2	$3.060E7$	18.7	11	0.0	$2.654E7$	2.9	4	0.1	$2.862E7$	11.0	25.3	0.1	$2.582E7$	**0.1**	6	0.1
gesa2-o	$3.626E7$	40.7	15	0.0	$2.902E7$	12.6	6	0.1	$3.477E7$	34.9	28.8	0.1	$2.690E7$	**4.3**	14	0.1
glass4	5.673E9	372.7	66	0.0	$7.386E9$	515.5	50	0.1	$4.299E9$	258.3	221	0.1	$3.798E9$	**216.5**	189	0.1
harp2	$-6.031E7$	18.4	149	0.0	$-6.280E7$	15.0	6	0.1	$-5.842E7$	21.0	364.7	0.1	$-6.786E7$	**8.2**	12	0.1

Table 2 continued

Problem	Std ($\alpha = 0$)				Prop ($\alpha = 0$)				Std ($\alpha = 1$)				Prop ($\alpha = 1$)			
	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time
ic97p	4,288.1	8.8	858	2.1	4,274.3	**8.4**	677	2.2	4,291.0	8.9	1008.2	2.5	4,307.9	9.3	711	2.2
ic97t	$4{,}277.8^{(6)}$	8.5	754	1.2	$4{,}318.4^{(5)}$	9.5	684	1.1	$4{,}268.6^{(8)}$	8.3	721	1.1	$4{,}215.8^{(8)}$	**6.9**	851	1.4
icir97p	$7{,}142.7^{(3)}$	12.3	766	26.6	$7{,}225.3^{(6)}$	13.6	697	30.5	$7{,}129.2^{(6)}$	**12.1**	829.2	24.7	$7{,}215.9^{(8)}$	13.4	710	26.0
icir97t	$7{,}338.0^{(1)}$	15.4	655	25.0	$7{,}059.0^{(1)}$	–	669	28.7	$7{,}240.0^{(1)}$	13.8	732.5	25.1	$7{,}022.5^{(2)}$	**10**.4	558	17.1
liu	6,249.6	457.0	1	0.0	6,249.6	457.0	1	0.1	4,258.2	**279.5**	201.9	0.4	4,258.2	**279.5**	183	0.9
manna81	−12,885.4	2.1	4	0.1	−12,994.8	1.3	3	0.1	−12,889.2	2.1	7.6	0.2	−13,164.0	**0.0**	4	0.1
markshare1	460.7	45,970.0	2	0.0	258.8	**25,780**.0	1	0.1	454.7	45,370.0	67.7	0.0	467.0	46,600.0	68	0.1
markshare2	649.5	64,850.0	2	0.0	516.1	**51,510.0**	1	0.1	658.3	65,730.0	69.6	0.0	554.7	55,370.0	70	0.1
mas74	34,633.8	193.5	2	0.0	24,049.8	103.8	1	0.1	20,289.2	**71.9**	107	0.0	20,973.9	77.7	106	0.1
mas76	61,759.2	54.4	2	0.0	51,520.3	28.8	1	0.1	51,702.4	29.2	108.2	0.0	49,251.1	**23.1**	106	0.1
misc07	4,333.5	54.2	132	0.1	4,320.5	53.8	18	0.1	3,828.0	**36.2**	164.3	0.2	3,921.0	39.5	23	0.1
mkc	−220.6	60.9	3	0.0	−263.2	53.3	1	0.1	−255.7	54.7	11.9	0.1	−293.3	**48.0**	12	0.1
mod011	$-2.314E7$	57.6	1	0.0	$-2.314E7$	57.6	1	0.1	$-4.698E7$	**13.9**	11.5	0.1	$-4.698E7$	**13.9**	11	0.1
modglob	$3.096E7$	49.3	1	0.0	$3.096E7$	49.3	1	0.1	$2.202E7$	**6.2**	64	0.0	$2.202E7$	**6.2**	64	0.1
msc98-ip	$3.043E7$	53.4	99	25.4	$2.664E7$	34.3	14	19.9	$2.946E7$	48.5	91.4	24.4	$2.246E7$	**13.2**	20	18.2
mzzv11	−6,534.4	69.9	149	61.5	−14,347.4	33.9	3	27.9	−16,426.2	24.4	258.8	90.5	−18, 315.0	**15.7**	19	23.7
mzzv42z	−10,236.0	50.2	11	12.3	−15,176.6	26.1	2	9.8	−14,453.0	29.6	96.4	24.4	−16, 525.8	**19.5**	20	10.1
neos10	−79.0	93.0	110	1.8	−627.1	**44.7**	1	0.1	−364.4	67.9	105.4	1.5	−262.7	76.9	13	0.1
neos16	–	–	691	12.2	–	–	1,050	22.2	–	–	739.6	11.9	$453.7^{(3)}$	**1.5**	1,099	19.4
neos20	$-147.7^{(9)}$	66.0	1, 063	3.9	−148.4	65.8	1,149	5.8	$-165.0^{(8)}$	**62.0**	937.5	3.7	−162.6	62.5	1,172	5.6

Table 2 continued

Problem	Std ($\alpha = 0$)				Prop ($\alpha = 0$)				Std ($\alpha = 1$)				Prop ($\alpha = 1$)			
	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time
neos7	$8.801E6$	1,119.1	60	0.1	$2.452E6$	239.7	8	0.1	$9.464E5$	31.1	93.5	0.3	$8.567E5$	**18.7**	32	0.1
neos8	−808.0	78.3	105	1.9	−3,347.1	10.0	2	0.1	−1,129.0	69.6	172.4	3.2	−3,475.5	**6.5**	1	0.1
net12	337.0	57.5	127	7.4	310.0	44.9	8	4.5	337.0	57.5	261.1	12.1	291.9	**36.4**	36	7.8
nh97p	$1{,}486.8^{(6)}$	4.9	1,100	12.7	$1{,}471.6^{(8)}$	**3.8**	634	10.2	$1{,}486.8^{(6)}$	4.9	1135.5	12.7	$1{,}491.2^{(6)}$	5.2	621	10.6
nh97t	$1{,}596.0^{(1)}$	12.6	885	14.5	$1{,}433.0^{(1)}$	**1.1**	849	17.4	–	–	803.7	12.8	–	–	750	14.8
noswot	−31.6	22.9	36	0.0	−39.6	**3.4**	1	0.1	−32.0	22.0	156.4	0.1	−29.1	29.0	67	0.1
nsrand-ipx	138,448.0	170.4	4	0.1	86,144.0	68.3	1	0.1	95,888.0	87.3	9.6	0.1	84,128.0	**64.3**	10	0.2
nw04	19,938.8	18.2	2	0.3	26,718.2	58.5	6	0.7	19,264.2	14.2	26	2.5	18,215.4	**8.0**	12	1.4
opt1217	−15.6	2.5	5	0.0	−14.8	7.5	1	0.1	−15.8	1.3	45	0.0	−16.0	**0.0**	36	0.1
p2756	4,489.0	43.7	680	1.1	8,586.0	174.8	2	0.1	4,406.9	**41.1**	684.6	1.0	7,211.4	130.8	15	0.1
pk1	144.1	1,210.0	1	0.0	144.1	1,210.0	1	0.1	58.7	**433.6**	57.6	0.0	58.7	**433.6**	58	0.1
pp08a	12,211.0	66.1	2	0.0	12,211.0	66.1	2	0.1	11,408.0	**55.2**	11.2	0.0	11,408.0	**55.2**	11	0.1
pp08aCUTS	12,080.0	64.4	2	0.0	12,080.0	64.4	2	0.1	9,897.0	**34.7**	10.1	0.0	9,897.0	**34.7**	10	0.1
protfold	−12.0	61.3	402	156.0	−8.0	74.2	29	14.1	−12.8	**58.7**	413.6	182.8	−10.0	67.7	42	27.5
qiu	1,715.9	1,391.4	2	0.1	1,586.7	1,294.1	1	0.1	881.2	763.2	7.7	0.1	755.3	**668.4**	8	0.2
rd-rplusc-21	$181{,}833.0^{(1)}$	9.9	584	218.7	$182{,}912.5^{(4)}$	10.6	717	553.5	$178{,}649.3^{(3)}$	8.0	637.7	242.1	$177{,}801.0^{(3)}$	**7.5**	790	594.0
rB10-011000	114,848.1	480.9	12	0.2	103,371.3	422.8	13	0.3	117,756.2	495.6	37.6	0.3	69,949.3	**253.8**	29	0.3
rB10-011001	116,200.3	407.6	14	0.2	101,639.4	344.0	13	0.3	116,437.5	408.6	38.1	0.3	72,435.8	**216.4**	31	0.4
rB11-010000	217,883.3	525.6	14	0.6	199,134.8	471.7	11	0.8	216,750.3	522.3	36.3	0.9	136,067.2	**290.7**	30	1.1
rB11-110001	208,165.6	356.6	19	1.9	175,492.6	284.9	12	1.9	206,799.2	353.6	40	2.5	117,574.3	**157.9**	30	2.6
rB12-111111	66,876.2	**63.6**	1,001	39.8	72,278.7	76.8	273	12.8	67,578.1	65.3	872.1	34.8	71,550.3	75.0	319	15.2

Table 2 continued

Problem	Std ($\alpha = 0$)				Prop ($\alpha = 0$)				Std ($\alpha = 1$)				Prop ($\alpha = 1$)			
	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time	Obj	Gap%	Iter	Time
rC10-001000	238,324.8	1,976.9	44	0.1	153,219.4	1, 235.2	28	0.1	206,743.5	1,701.7	73	0.2	37,448.7	**226.4**	28	0.1
rC10-100001	229,615.3	1,092.7	55	0.5	176,113.3	814.8	123	1.2	165,094.0	757.6	95.5	0.6	122,325.5	**535.4**	160	1.9
rC11-010100	145,222.8	373.9	12	0.5	137,185.2	347.7	1	0.4	145,569.2	375.0	43.8	0.9	117,582.9	**283.7**	31	1.0
rC11-011100	139,798.7	501.0	12	0.3	121,784.2	423.6	1	0.3	135,137.7	481.0	33.6	0.5	104,231.1	**348.1**	27	0.6
rC12-100000	526,605.8	1,264.0	27	1.6	405,461.1	950.2	24	2.0	398,206.4	931.4	53.1	2.7	174,330.1	**351.6**	47	3.1
rC12-111100	114,857.9	199.2	5	0.8	113,210.0	194.9	1	0.5	115,055.8	199.7	26.3	1.0	79,168.1	**106.2**	24	1.2
roll3000	19,095.5	48.1	322	5.9	20,205.2	56.8	43	0.5	19,796.4	53.6	406.8	6.0	16,416.1	**27.4**	66	0.6
rout	1,584.4	47.0	42	0.0	1,703.5	58.1	20	0.1	1,520.3	**41.1**	77.1	0.0	1,542.4	43.1	33	0.1
set1ch	89,012.4	63.2	9	0.0	86,068.0	**57.8**	1	0.1	96,497.2	76.9	39.5	0.0	87,838.3	61.1	24	0.1
seymour	587.7	38.9	2	1.4	555.6	31.3	1	1.8	462.6	9.4	7.5	2.3	437.4	**3.4**	5	2.4
sp97ar	$1.530E9$	131.2	2	1.3	$1.049E9$	58.6	1	0.9	$9.118E8$	37.8	9.4	2.9	$8.446E8$	**27.6**	9	2.5
swath	1,457.9	211.9	105	2.3	1,409.1	201.5	57	1.9	1,398.4	199.2	252	7.6	1,180.2	**152.5**	146	7.1
t1717	248,563.6	27.0	52	237.3	279,939.5	43.0	9	44.3	231,006.0	**18.0**	56.5	174.5	427,522.2	118.4	37	152.0
timtab1	$1.324E6$	73.2	162	0.3	$1.320E6$	72.6	63	0.2	1.324E6	73.1	317.4	0.5	$1.315E6$	**71.9**	275	0.5
timtab2	$1.798E6^{(3)}$	**64.0**	800	6.0	$1.900E6^{(6)}$	73.3	783	5.8	$1.813E6^{(1)}$	65.3	746.6	5.7	$1.833E6^{(5)}$	67.1	708	5.1
tr12-30	260,163.9	99.2	6	0.0	252,994.8	93.7	7	0.1	182,675.4	**39.9**	21.1	0.0	183,105.4	40.2	21	0.1
vpm2	18.3	32.7	2	0.0	17.3	25.8	1	0.1	17.2	24.9	10.6	0.0	16.5	**19.6**	11	0.1
Geom. means		77.2	27	1.0		57.6	9	0.8		52.4	84	1.3		35.5	42	1.0

Averages over 10 runs with different random seeds

Table 3 Success rate when using different variable orders

Scenario	UB	Std	Prop	Frac	Rev	lr	rnd
Alone	None	87	90	90	88	90	91
	100%	85	88	88	86	86	87
	10%	50	59	55	53	51	56
Emb	None	49	68	64	71	**56**	72
	100%	32	55	53	50	**44**	54
	10%	17	21	21	18	21	17

`prop` produced strictly better (average) solution than `std` in 50 cases, whereas the opposite happened in 19 cases. These results confirm the effectiveness of exploiting propagation within a FP scheme. As to computing times, the four variants are almost the same, with `prop` giving a small speedup with respect to `std`.

6.2 Impact of different variable orders

As already mentioned in Sect. 4, the order in which we choose the next variable to round can have, at least in principle, a large impact on the performance of the overall propagation scheme. To assess the actual effect of different variable orders, we compared the following four variants of our FP2.0 algorithm where all FP parameters but the variable order are kept unchanged:

frac Variables are ranked in the same order used in `prop`, but without the random swaps after the 10th iteration (i.e., in order of increasing fractionality, with binary variables always preceding the general integer ones).

rev Variables are ranked in order of *decreasing* fractionality. This is exactly the reverse of `frac`.

lr Variables are ranked from left to right, according to their index in the formulation of the problem.

rnd The order is randomly chosen at each iteration.

The success rate of all the variants is reported in Table 3, for each scenario and upper bound combination. For reference purposes, the success rates of `std` and `prop` are also reported.

At first sight, the variable ordering seems not to have a great impact on the success rate. Indeed, a standard Cochran Q-test followed by post hoc McNemar comparisons confirms that the proportions are not statistically different, except for the `lr` order which is significantly worse in the `emb` scenario for upper bounds None and 100%. As far as the number of iterations and computing time needed to find a solution are concerned, the differences are minimal, except again for `lr` with certain UB's.

While similar from the success rate and computing time point of view, the different variants are however significantly different from the solution quality point of view. The average gap closed in the settings of Sect. 6.1, namely scenario `alone` and no upper bound, is reported in Table 4. According to the table, while `prop` and `frac` reduce the average gap considerably, the performance of `lr`, `rnd` and `rev` is

Table 4 Solution quality when using different variable orders, for both the original ($\alpha = 0$) and objective ($\alpha = 1$) FP versions

Variant	Gap% ($\alpha = 0$)	Gap% ($\alpha = 1$)
Std	77.2	52.4
Prop	57.6	35.5
Frac	62.2	40.2
Rev	75.0	43.8
lr	76.1	50.6
rnd	73.8	45.2

considerably worse, about the same as `std`. Overall, among the proposed orders, `prop` qualifies as the method of choice because it is at least as successful and fast as the other variants, with a better solution quality of the solutions found.

7 Conclusions

In this paper a new version of the Feasibility Pump heuristic has been proposed and evaluated computationally. The idea is to exploit a concept borrowed from Constraint Programming (namely, constraint propagation) to have a more powerful rounding operation. The computational results on both binary and general integer MIPs show that the new method finds more feasible solutions than its predecessor, typically in a shorter computing time. A main feature of the new method is the reduced number of iterations to reach convergence, that implies a significantly improved solution quality of the final outcome.

Acknowledgments This work was supported by the University of Padova "Progetti di Ricerca di Ateneo", under contract no. CPDA051592 (project: Integrating Integer Programming and Constraint Programming) and by the Future and Emerging Technologies unit of the EC (IST priority), under contract no. FP6-021235-2 (project ARRIVAL). We thank three anonymous referees for their constructive comments, leading to an improved version of the paper.

References

1. Achterberg, T.: Constraint integer programming. PhD thesis, Technische Universität Berlin, (2007)
2. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optim. **4**, 77–86 (2007)
3. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett., **34**:361–372. Problems available at http://miplib.zib.de (2006)
4. Balas, E., Ceria, S., Dawande, M., Margot, F., Pataki, G.: OCTANE: A new heuristic for pure 0–1 programs. Oper. Res. **49**, 207–225 (2001)
5. Balas, E., Martin, C.: Pivot-and-complement: A heuristic for 0-1 programming. Manage. Sci. **26**, 86–96 (1980)
6. Balas, E., Schmieta, S., Wallace, C.: Pivot and shift—a mixed integer programming heuristic. Discrete Optim. **1**, 3–12 (2004)
7. Bertacco, L.: Feasibility pump. Source code available at http://www.or.deis.unibo.it/research_pages/ORcodes/FP-gen.html
8. Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optim. **4**, 63–76 (2007)

9. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Program. **102**, 71–90 (2005)
10. Eckstein, J., Nediak, M.: Pivot, cut, and dive: a heuristic for 0–1 mixed integer programming. J. Heuristics **13**, 471–503 (2007)
11. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. **104**, 91–104 (2005)
12. Fischetti, M., Lodi, A.: Local branching. Math. Program. **98**(1–3), 23–47 (2003)
13. Fischetti, M., Polo, C., Scantamburlo, M.: A local branching heuristic for mixed-integer programs with 2-level variables, with an application to a telecommunication network design problem. Networks **44**, 61–72 (2004)
14. Gecode Team: Gecode: Generic constraint development environment, 2008. Available at http://www.gecode.org (2008)
15. Glover, F., Laguna, M.: General purpose heuristics for integer programming—part I. J. Heuristics **2**, 343–358 (1997)
16. Glover, F., Laguna, M.: General purpose heuristics for integer programming—part II. J. Heuristics **3**, 161–179 (1997)
17. Glover, F., Laguna, M.: Tabu Search. Kluwer, Dordrecht (1997)
18. Gondzio, J.: Presolve analysis of linear programs prior to applying an interior point method. INFORMS J. Comput. **9**, 73–91 (1997)
19. Hansen, P., Mladenović, N., Urosevic, D.: Variable neighborhood search and local branching. Comput. Oper. Res. **33**, 3034–3045 (2006)
20. Hillier, F.: Efficient heuristic procedures for integer linear programming with an interior. Oper. Res. **17**, 600–637 (1969)
21. Hooker, J.N.: Integrated Methods for Optimization. Springer, New York (2006)
22. Ibaraki, T., Ohashi, T., Mine, F.: A heuristic algorithm for mixed-integer programming problems. Math. Program. Study **2**, 115–136 (1974)
23. ILOG. CPLEX 11.0 User's manual (2008)
24. Lagerkvist M.Z., Schulte C.: Advisors for incremental propagation. In: Bessière C. (ed.) Thirteenth international conference on principles and practice of constraint programming. Lecture notes in computer science, vol. 4741, pp. 409–422. Springer-Verlag, New York (2007)
25. Løkketangen, A: Heuristics for 0–1 mixed-integer programming. In: Pardalos, P., Garey, M.R. (eds.) Handbook of Applied Optimization, pp. 474–477. Oxford University Press, Oxford (2002)
26. Løkketangen, A., Glover, F.: Solving zero/one mixed integer programming problems using tabu search. Eur. J. Oper. Res. **106**, 624–658 (1998)
27. Maros, I.: Computational Techniques of the Simplex Method. Kluwer, Dordrecht (2002)
28. Mittelmann, H.D.: Benchmarks for optimization software: Testcases. Problems available at http://plato.asu.edu/sub/testcases.html
29. Rossi, F., van Beek, P., Walsh, T. (eds.): Computational Techniques of the Simplex Method. Elsevier, Amsterdam (2006)
30. Savelsbergh, M.W.P.: Preprocessing and probing for mixed integer programming problems. ORSA J. Comput. **6**, 445–454 (1994)
31. Schulte, C.: Programming constraint services. PhD thesis, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany (2000)
32. Schulte, C., Stuckey, P.J.: Speeding up constraint propagation. In Wallace M. (ed.) Principles and practice of constraint programming-CP 2004, 10th international conference. Lecture notes in computer science, vol. 3258, pp. 619–633. Springer (2004)