An Algorithm-Independent Measure of Progress for Linear Constraint Propagation

Boro Sofranac sofranac@zib.de
Berlin Institute of Technology and Zuse Institute Berlin

Ambros Gleixner gleixner@zib.de
HTW Berlin and Zuse Institute Berlin

Sebastian Pokutta pokutta@zib.de
Berlin Institute of Technology and Zuse Institute Berlin

Abstract

Propagation of linear constraints has become a crucial sub-routine in modern Mixed-Integer Programming (MIP) solvers. In practice, iterative algorithms with tolerance-based stopping criteria are used to avoid problems with slow or infinite convergence. However, these heuristic stopping criteria can pose difficulties for fairly comparing the efficiency of different implementations of iterative propagation algorithms in a real-world setting. Most significantly, the presence of unbounded variable domains in the problem formulation makes it difficult to quantify the relative size of reductions performed on them. In this work, we develop a method to measure—independently of the algorithmic design—the progress that a given iterative propagation procedure has made at a given point in time during its execution. Our measure makes it possible to study and better compare the behavior of bounds propagation algorithms for linear constraints. We apply the new measure to answer two questions of practical relevance: (i) We investigate to what extent heuristic stopping criteria can lead to premature termination on real-world MIP instances. (ii) We compare a GPU-parallel propagation algorithm against a sequential state-of-the-art implementation and show that the parallel version is even more competitive in a real-world setting than originally reported.

1 Introduction

This paper is concerned with *Mixed-Integer Linear Programs* (MIPs) of the form

$$\min\{c^T x \mid Ax \leq b, \ell \leq x \leq u, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in I\}, \tag{1}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $I \subseteq \mathbb{N} = \{1, \ldots, n\}$. Additionally, $\ell \in \mathbb{R}^n_{-\infty}$ and $u \in \mathbb{R}^n_{\infty}$, where $\mathbb{R}_{\infty} := \mathbb{R} \cup \{\infty\}$ and $\mathbb{R}_{-\infty} := \mathbb{R} \cup \{-\infty\}$. For each variable x_j, the interval $[\ell_j, u_j]$ is called its *domain*, which is defined by its lower and upper *bounds* ℓ_j and u_j, which may be infinite.

Surprisingly fast solvers for solving MIPs have been developed in practice despite MIPs being $\mathcal{NP}$-hard in the worst case [5, 15]. To this end, the most successful method has been the *branch-and-bound* algorithm [16] and its numerous extensions. The key idea of this method is to split the original problem into several sub-problems (*branching*) which are hopefully easier to solve. By doing this recursively, a *search tree* is created with nodes being the individual sub-problems. The *bounding* step solves relaxations of sub-problems to obtain a lower bound on their solutions. This bound can then be used to prune sub-optimal nodes which cannot lead to improving solutions. By doing this, the algorithm tries to avoid having to enumerate exponentially many sub-problems. The most common way to obtain a relaxation of a sub-problem is to drop the integrality constraints of the variables. This yields a *Linear Program* (LP) which can be solved e.g., by the simplex method [21].

This core idea is extended by numerous techniques to speed up the solution process. One of the most important techniques is called *constraint propagation*. It improves the formulation of the (sub)problem by removing parts of domains of each variable that it detects cannot lead to *feasible* solutions [23]. The more descriptive term *bounds propagation* or *bounds tightening* is used to denote the variants that maintain a continuous interval as domain. Modern MIP solvers make use of this technique during *presolving* in order to improve the global problem formulation [24], as well as during the branch-and-bound algorithm to improve the formulation of the sub-problems at the nodes of the search tree [2].

In practice, efficient implementations exist in MIP solvers [4, 2] and recently even a GPU-parallel algorithm [26] has been developed. These are iterative methods, which may converge to the tightest bounds only at infinity. For such methods, the presence of unbounded variable domains in the problem formulation makes the quantification of the relative distance to the final result at a given iteration difficult. (Iterative bounds tightening has a unique fixed point to which it converges, see Section 2.2.) In turn, this makes it difficult to define an implementation-independent measure of how much progress these algorithms have achieved at a given iteration.

In this paper, we address this difficulty and introduce tools to study and compare the behavior of iterative bounds tightening algorithms in MIP. We show that the reduction of infinite bounds to some finite values is a fundamentally different process from the subsequent (finite) improvements thereafter, and thus propose to measure the ability of an algorithm to make progress in each of the processes independently. We show how the challenge posed by infinite starting bounds can be solved and provide methods for measuring the progress of both the infinite and the finite domain reductions. Pseudocode and hints are provided to aid independent implementation of our procedure. Additionally, the code of our own implementation is made publicly available.

On the applications side, the new procedure is used to investigate two questions. First, we analyze to what extent heuristic, tolerance-based stopping criteria as typically imposed by real-world MIP solvers can cause iterative bounds tightening algorithms to terminate prematurely; we find that this situation occurs rarely in practice. Second, we compare a newly developed, GPU-based propagation algorithm [26] to a state-of-the-art sequential implementation in a real-world setting where both are terminated early; we show that the GPU-parallel version is even more competitive than originally reported.

The rest of the paper is organized as follows. After presenting the necessary background and motivation in Section 2, we discuss the properties of bounds propagation and its ability to perform reductions on infinite and on finite bounds in Section 3. Based on the findings, we present functions used to measure the progress of bounds tightening algorithms in Section 4. Lastly, in Section 5, we apply the developed procedure to answer the above-mentioned questions and present our computational results. Section 6 gives a brief outlook.

2 Background and Motivation

In Section 2.1, we introduce some basic terminology used in the Constraint Programming (CP) and MIP communities, related to constraint propagation. Section 2.2 formally presents bounds propagation of linear constraints alongside some known results from literature that are relevant for the discussions in the paper. In Section 2.3 we outline the problems that motivate the paper.

2.1 Constraint Propagation in CP and MIP

In the Constraint Programming (CP) community, constraint propagation appears in a variety of forms, both in terms of the algorithms and its desired goals [23]. The propagation algorithms are implemented via mappings called *propagators*. A propagator is a monotonically decreasing function from variable domains to variable domains [25]. The goal of most propagation algorithms is fomalized through the notion of *consistency*, which these algorithms strive to achieve. The most successful consistency technique is *arc consistency* [18]. Multivariate extension of arc consistency has been called *generalized arc consistency* [20], as well as *domain consistency* [27], and *hyper-arc consistency* [19]. Informally speaking, a given domain is *domain consistent* for a given constraint if it is the least domain containing all solutions to the constraint (see [23] for a formal definition).

The main idea of *bounds consistency* is to relax the consistency requirement to only require the lower and the upper bounds of domains of each variable to fulfill it. There are several bounds consistency notions in the CP literature [10]. In this paper, we adopt the notion of bounds consistency from [2, Definition 2.7].

Modern CP solvers often work with a number of propagators which might or might not strive for different levels of consistency [25]. In this setting, the notions such as *greatest common fixed point* (see [9, Definition 4]) and consistency of a system of constraints are often analysed as a product of a set of propagators. Solvers often focus on optimizing the interplay between different propagators (e.g., see [25]) to quickly decide feasibility.

In MIP solving, constraint propagation additionally interacts with many other components that are mostly focused on reaching and proving optimality, see [3, 6, 7, 1] for examples of different approaches to integrate

constraint propagation and MIP. As a result, the role of constraint propagation in the larger solving process changes and developers are faced with different computational trade-offs. In practice, propagation is almost always terminated before the fixed point is reached [2]. In this paper, we are concerned with constraint propagation of a set of linear constraints, where we explicitly include the presence of continuous variables and of variables with initially unbounded domains, which frequently occur in real-world MIP formulations.

2.2 Bounds Propagation of Linear Constraints

A *linear constraint* can be written in the form

$$\underline{\beta} \leq \sum_{i=1}^{n} a_i x_i \leq \overline{\beta}, \tag{2}$$

where $\underline{\beta} \in \mathbb{R}_{-\infty}$ and $\overline{\beta} \in \mathbb{R}_{\infty}$ are left and right hand sides, respectively, and $a \in \mathbb{R}^n$ is the vector of constraint coefficients. Variables x_i have lower and upper bounds $\ell_i \in \mathbb{R}_{-\infty}$ and $u_i \in \mathbb{R}_{\infty}$, respectively.[1] We require the following definitions:

Definition 1 (activity bounds and residuals)**.** *Given a constraint of the form* (2) *and bounds* $\ell \leq x \leq u$*, the functions* $\underline{\alpha} : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *and* $\overline{\alpha} : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *are called the* minimum *and* maximum activities *of the constraint, respectively, and are defined as*

$$\underline{\alpha} = \underline{\alpha}(\ell, u) = \sum_{i=1}^{n} a_i b_i \text{ with } b_i = \begin{cases} \ell_i & \text{if } a_i > 0 \\ u_i & \text{if } a_i < 0 \end{cases}, \tag{3a}$$

and

$$\overline{\alpha} = \overline{\alpha}(\ell, u) = \sum_{i=1}^{n} a_i b_i \text{ with } b_i = \begin{cases} u_i & \text{if } a_i > 0 \\ \ell_i & \text{if } a_i < 0 \end{cases}. \tag{3b}$$

The functions $\underline{\alpha}_j : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty}, \{1, \ldots, n\} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *and* $\overline{\alpha}_j : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty}, \{1, \ldots, n\} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *are called the* j-th minimum activity residual *and the* j-th maximum activity residual *of the constraint, and are defined as*

$$\underline{\alpha}_j = \underline{\alpha}_j(\ell, u, j) = \sum_{i=1, i \neq j}^{n} a_i b_i \text{ with } b_i = \begin{cases} \ell_i & \text{if } a_i > 0 \\ u_i & \text{if } a_i < 0 \end{cases}, \tag{4a}$$

and

$$\overline{\alpha}_j = \overline{\alpha}_j(\ell, u, j) = \sum_{i=1, i \neq j}^{n} a_i b_i \text{ with } b_i = \begin{cases} u_i & \text{if } a_i > 0 \\ \ell_i & \text{if } a_i < 0 \end{cases}. \tag{4b}$$

Definition 2 (bound candidate functions)**.** *The functions* $\mathcal{B}^j_{surplus} : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *and* $\mathcal{B}^j_{slack} : \mathbb{R}^n_{-\infty}, \mathbb{R}^n_{\infty} \mapsto \mathbb{R} \cup \{-\infty, \infty\}$ *are called the* bound candidate functions *and are defined as*

$$\mathcal{B}^j_{surplus}(\ell, u) = \frac{\overline{\beta} - \underline{\alpha}_j}{a_j}, \tag{5a}$$

and

$$\mathcal{B}^j_{slack}(\ell, u) = \frac{\underline{\beta} - \overline{\alpha}_j}{a_j}. \tag{5b}$$

[1] When $x \in \mathbb{Z}$, then $\ell \in \mathbb{Z}_{-\infty}$ and $u \in \mathbb{Z}_{\infty}$, however, because $\mathbb{Z} \subset \mathbb{R}$, integer variables can be handled the same way as real ones. In the remainder of the paper, $\mathbb{Z}$ will be used only where necessary.

Then, the following observations are true and can be be translated into algorithmic steps, see, e.g., [2, 14, 9]:

Observation 1 (linear constraint propagation)**.** *1. If $\underline{\beta} \leq \underline{\alpha}$ and $\overline{\alpha} \leq \overline{\beta}$, then the constraint is redundant and can be removed.*

2. If $\underline{\alpha} > \overline{\beta}$ or $\underline{\beta} > \overline{\alpha}$, then the constraint cannot be satisfied and hence the entire (sub)problem is infeasible.

3. Let x satisfy (2)*, i.e., $\underline{\beta} \leq \sum_{i=1}^{n} a_i x_i \leq \overline{\beta}$, then for all $j = \{1, \ldots, n\}$ with $a_j > 0$,*

$$\ell^{new} = \mathcal{B}^j_{slack}(\ell, u) \leq x_j \leq \mathcal{B}^j_{surplus}(\ell, u) = u^{new}, \tag{6a}$$

and for all $j = \{1, \ldots, n\}$ with $a_j < 0$,

$$\ell^{new} = \mathcal{B}^j_{surplus}(\ell, u) \leq x_j \leq \mathcal{B}^j_{slack}(\ell, u) = u^{new}. \tag{6b}$$

4. For all $j \in \{1, \ldots, n\}$ such that $x_j \in \mathbb{Z}$,

$$\lceil \ell^{new}_j \rceil \leq x_j \leq \lfloor u^{new}_j \rfloor \tag{7}$$

If the first two steps are not applicable, the algorithm computes the new bounds ℓ^{new} and u^{new} in Steps 3 and 4. For a given variable j, if $\ell^{\text{new}}_j > \ell_j$, then the bound is updated with the new value. Similarly, u_j is updated if $u^{\text{new}}_j < u_j$.

An actual implementation may skip Steps 1 and 2 without changing the result. This is because for redundant constraints Steps 3 and 4 correctly detect no bound tightenings, and for infeasible constraints, Steps 3 and 4 lead to at least one variable with an empty domain, i.e., $\ell^{\text{new}}_j > u^{\text{new}}_j$.

When propagating a system of the type (1) which consists of several constraints, one simply applies the above steps to each constraint independently. Notice that in such systems, it is possible for two or more constraints to share the same variables (i.e., coefficients a_j are non-zero in several constraints). Therefore, if a bound of a variable is changed in one constraint, this can trigger further bound changes in the constraints which also have this variable. This gives the propagation algorithm its iterative nature, as one has to repeat the propagation process over the constraints as long as at least one bound change is found. A pass over all the constraints is also called a *propagation round*. If no bound changes are found during a given round, then no further progress is possible and the algorithm terminates. At this point, all constraints are guaranteed to be bound consistent [2].

This algorithm can be interpreted as a fixed-point iteration in the space of variable and activity bounds with a unique fixed point [9]; it converges to this fixed point, however not necessarily in finite time [8]. Additionally, even when it does converge to the fixed point in finite time, convergence can be very slow in practice [2, 9, 17]. To deal with this, practical implementations of bounds propagation introduce tolerance-based termination criteria which stop the algorithm if the progress becomes too slow, i.e., the relative size of improvements on the bounds falls below a specified threshold. With this modification, the algorithm always terminates in finite time (but not in worst-case polynomial-time), however, it may fail to compute the best bounds possible.

To distinguish the above-described approach from alternative methods to compute consistent bounds (see, e.g., [8] for a method solving a single LP instead), we will use the following definition:

Definition 3 (Iterative Bounds Tightening Algorithm)**.** *Given variable bounds ℓ, u of a problem of the form* (1)*, any algorithm updating these bounds by calculating ℓ^{new}, u^{new} via* (6a)*,* (6b)*, and* (7) *iteratively as described in Observation 1, thus traversing a sequence of bounds $(\ell, u)_1, (\ell, u)_2, \ldots$ is called an* iterative bounds tightening algorithm *(IBTA).*

Note that this definition leaves the flexibility for individual algorithmic choices, for example, the timing of when bound changes are applied or the order in which the constraints are processed. If a given algorithm applies the found changes immediately, making them available to subsequent constraints in the same iteration, it might traverse a shorter sequence of bounds to the fixed point than the algorithm which delays updates of bounds until the end of the current iteration (e.g., because it processes constraints in parallel). The ordering of processed constraints can lead to different traversed sequences because a given bound change that depends on other changes being applied first might be missed in a given iteration if the constraint it depends on is not processed first.

2.3 Motivation

Our motivation for this paper is threefold:

1. *Estimating the premature stalling effect of IBTAs:* In the context of MINLP, Belotti et al. [8] propose an alternative bounds propagation algorithm that computes the bounds at the fixed point directly by solving a single linear program. This approach circumvents non-finite convergence behavior and shows that the bounds at the fixed point can in theory be computed in polynomial time.

Nevertheless, in practice, the trade-off between the quality of obtained bounds including their effect on the wider branch-and-bound algorithm and the algorithm's runtime makes the iterative bounds propagation with tolerance-based stopping criteria the most effective method in most cases, despite its exponential worst-case runtime. The use of stopping criteria still leaves individual instances or potentially even instance classes susceptible to the following effect stated by Belotti et al. as a motivation for their LP-based approach, which is also the motivation for our paper: *"However, because the improvements are not guaranteed to be monotonically non-increasing, terminating the procedure after one or perhaps several small improvements might in principle overlook the possibility of a larger improvement later on."* In their paper, no attempt is made to quantify this statement, as likely out-of-scope and non-trivial to answer.

In this work, we aim to develop a methodology to quantify the overall progress that a given IBTA achieved up to a given point in its execution. Ideally, we would like to have a function f, which maps current variable bounds to a scalar value, for example in $[0, 100]$, which measures the achieved progress. The main difficulty in developing such a function comes in the form of unbounded variable domains in the input instances (and potentially during the algorithm's execution). Observing the values of such a function over the execution time of the algorithm could then be used to study the behavior of IBTAs on instances of interest and quantify the effect brought up by Belotti et al., which we call *premature stalling* (see Section 5.2 for formal definition). Furthermore, an algorithm-independent f would allow comparing the behavior of different IBTAs with respect to premature stalling.

2. *Performance comparison of different IBTAs in practice:* As already motivated by Definition 3, different IBTAs might traverse different sequences of bounds from the initial values to the fixed point. Additionally, we stated in Section 2.2 that in practice, iterative bounds propagation is used exclusively with tolerance-based stopping criteria, meaning that the algorithm is stopped potentially before reaching the fixed point. The following problem then arises: for two such algorithms traversing different sequences of bounds that are stopped before reaching the unique fixed point, how do we judge which one performed better? Perhaps a more natural way to formulate this question is: *in how much time do the two algorithms achieve the same amount of progress?* A function measuring the progress of iterative bounds propagation as already proposed can be used to answer this question.

As a concrete example, we will compare the following two IBTAs: the canonical, state-of-the-art sequential implementation, for example from [2], and a GPU-parallel algorithm recently proposed in [26]. In the preliminary computational study on the MIPLIB 2017 test set [13] presented in [26], the two algorithms are compared for the propagation to the fixed point (no tolerance-based stopping criteria). In this work, we will compare the performance of the two algorithms in a real-world setting, i.e., when terminated before reaching the fixed point.

3. *Designing stopping criteria:* as already stated, the tolerance-based stopping criteria are crucial for effective IBTAs. Notice that because different IBTAs might traverse different sequences of bounds, their average individual improvements on the bounds might be different in size. In fact, the study in [26] shows that on average, the size of improvements by the GPU-parallel algorithm is smaller than that of the canonical sequential implementation over the MIPLIB 2017 test set, despite its higher performance in terms of runtime to the fixed point. An important implication of this effect is that given two such algorithms, a given stopping criterion might be effective for one of them, but ineffective for the other. In this context, quantifying the magnitude and distribution of expected improvements of a given algorithm for a given problem class and its likelihood to prematurely stall, would allow one to make more informed decisions when designing *effective* stopping criteria.

Lastly, we believe that gaining insight into the behavior of these algorithms is a motivation in itself that could potentially benefit future and existing methodologies in the context of linear constraint propagation.

3 Finite and infinite domain reductions

Any IBTA starts with arrays of initial lower and upper bounds, $\ell^s \in \mathbb{R}^n_{-\infty}$ and $u^s \in \mathbb{R}^n_{\infty}$, respectively, and incrementally updates individual bounds towards the uniquely defined fixed-point bounds which we denote by ℓ^l and u^l. To denote the arrays of bounds at any given time between the start and the fixed point we simply use ℓ and u and call them *current bounds*. Obviously, it holds that $\ell^s_j \leq \ell_j \leq \ell^l_j$ and $u^s_j \geq u_j \geq u^l_j$ for all $j \in \{1, \ldots, n\}$. Observe that both initial and limit bounds may contain infinite values.

3.1 Reducing Infinite Bounds to Finite Values

Variables that start with infinite value in either lower or upper bound, will either remain infinite if no bound change is possible or will become finite values. We start with the following simple observation:

Observation 2. *Given a constraint of the form* (2) *and a given variable $j \in \{1, \ldots, n\}$ with a bound $\ell_j = -\infty$ (or $u_j = \infty$), the possibility of tightening this bound to some finite value depends on the signs of coefficients $a_j, j \in \{1, \ldots, n\}$, the finiteness of variable bounds $\ell_j, j \in \{1, \ldots, n\} \setminus j$ and $u_j, j \in \{1, \ldots, n\} \setminus j$, and the finiteness of $\underline{\beta}$ and $\overline{\beta}$, but not on the values that these variables take, if they are finite.*

Proof. To see the dependence on the sign of coefficients a, let the lower bound of a given variable j be $\ell_j = -\infty$ and let $\underline{\beta}$ and $\overline{\alpha}_j$ be finite, $\overline{\beta} = \infty$ and $\underline{\alpha}_j = -\infty$. Then, by (6a) and (6b), $a_j > 0$ implies $\ell^{\text{new}}_j \in \mathbb{R} > -\infty$ and the bound is updated. Else, if $a_j < 0$, then $\ell^{\text{new}}_j = -\infty$ and no bound change is possible.

The dependence on the finiteness of $\underline{\beta}$ and $\overline{\beta}$ is trivial, while the coefficients a are finite by problem definition. To see the dependence on the finiteness of variable bounds, consider the activities $\underline{\alpha}_j$ and $\overline{\alpha}_j$ of a variable j with $\ell_j = -\infty$, $u_j = \infty$, and $a_i > 0$ for all $i \in \{1, \ldots, n\}$. If there exists k such that $u_k = \infty, k \in \{1, \ldots, n\} \setminus j$ then $\overline{\alpha} = \infty$ and consequently ℓ_j cannot be tightened. Otherwise, if $u_k \in \mathbb{R}$ for all $k \in \{1, \ldots, n\} \setminus j$, then $\overline{\alpha} \in \mathbb{R}$ and a bound tightening is possible.

The specific finite values that the variables in (6a) and (6b) take have no effect on the possibility to reduce an infinite bound to a finite value because arithmetic operations between finite values again produce a finite value (also $a_j \neq 0$ by definition) and $-\infty < k < \infty$ for all $k \in \mathbb{R}$. Variables which are restricted to integer values also do not affect this process, as the operations $\lceil \ell_j \rceil$ and $\lfloor u_j \rfloor$ give $\ell_j, u_j \in \mathbb{Z}$ and $\mathbb{Z} \subset \mathbb{R}$. The same argument from above then applies. □

Notice that the same effect of finite bound changes triggering new bound changes in the subsequent propagation rounds is also true for infinite domain reductions, hence this process might also require more than one iteration. Furthermore, these iterations have the following property:

Corollary 1. *Let $k \in \mathbb{N}$ be the number of iterations a given IBTA takes to reach the fixed point. Then there is a number $c \leq k$, $c \in \mathbb{N}_0$ such that the first c propagation rounds have at least 1 reduction of an infinite to a finite bound, and none thereafter. By pigeonhole principle, c is at most the number of initially infinite bounds.*

Proof. The coefficients a and the left- and right-hand sides $\underline{\beta}$ and $\overline{\beta}$ are constants that do not change during the course of the algorithm. By Observation 2 the only thing left influencing the infinity reductions is the finiteness of variable bounds. If no infinite to finite reductions are made at any given round, then none can be made thereafter. Finite to infinite reductions are not possible as the algorithm only accepts improving bounds. □

In conclusion, the process of reducing infinite bounds to some finite values is independent and fundamentally different from the incremental improvements of finite values thereafter, which is driven by the values of the variables in (6b) and (6a). Accordingly, we will measure the ability of an algorithm to reduce the infinite bounds to some finite values separately from its ability to make improvements on the finite values of the bounds thereafter.

3.2 Finite Domain Reductions

Our main approach in measuring the progress of finite domain reductions (see Section 4.2) relies on the observation that the starting as well as the fixed point of propagation is uniquely defined for a given MIP problem and hence independent from the algorithm used. The measuring function then answers the following question: for given bounds ℓ and u at some time during the propagation process, how far have we gotten from the starting point ℓ^s and u^s, relative to the endpoint ℓ^l and u^l. When the bounds of a given variable did not change during the propagation process, or they are finite at both the start and the end, there is no difficulty in calculating such a measure. However, when a given variable bound started as an infinite value but was tightened to some finite value by the end of propagation, special care is needed to handle this case, which we address in this section.

In Section 2, we discussed how a sequential and a parallel propagation algorithm might traverse different sequences of bounds during their executions. Let us consider the first round of two such algorithms, and see what might happen to the bounds which start as infinite but are tightened during the course of the algorithm. When the sequential algorithm finds a bound change, it is immediately made available to the subsequent constraints in the same round. Consequently, if an infinite domain reduction happens in the subsequent constraints, it may produce a stronger finite value compared to the parallel algorithm which used the older (weaker) bound information. This serves to show that the first finite values that such bounds take may not be the same in different IBTAs. Hence they cannot be used safely to compare finite domain reductions across different implementations. In what follows, we construct a procedure to compute algorithm-independent reference values for each bound.

Definition 4 (weakest variable bounds)**.** *Given an optimization problem of the form* (1) *with starting variable bounds ℓ^s and u^s, we call $\overline{\ell}_j$* weakest lower bound *of variable j if*

- $\overline{\ell}_j = -\infty$ *and no IBTA can produce a finite lower bound $\ell_j \in \mathbb{R}$, or*
- $\overline{\ell}_j \in \mathbb{R}$ *and no IBTA can produce a finite lower bound $\ell_j \in \mathbb{R}$ with $\ell_j < \overline{\ell}_j$.*

We call $\overline{u}_j$ weakest upper bound *of variable j if*

- $\overline{u}_j = \infty$ *and no IBTA can produce a finite upper bound $u_j \in \mathbb{R}$, or*
- $\overline{u}_j \in \mathbb{R}$ *and no IBTA can produce a finite upper bound $u_j \in \mathbb{R}$ with $u_j > \overline{u}_j$.*

When both the starting and the limit bounds are finite we have $\overline{\ell}_j = \ell^s_j$ resp. $\overline{u}_j = u^s_j$ (because all IBTAs only accept improving bounds) and when they are both infinite we have $\ell^s_j = \ell^l_j = \overline{\ell}_j = -\infty$ resp. $u^s_j = u^l_j = \overline{u}_j = \infty$. Notice that the cases of $\ell^s_j \in \mathbb{R}$ with $\ell^l_j = -\infty$ and $u^s_j \in \mathbb{R}$ with $u^l_j = \infty$ are not possible as $\ell^s_j \leq \ell^l_j$ and $u^s_j \geq u^l_j$. The main challenge in computing $\overline{\ell}$ and $\overline{u}$ is due to the remaining case of $\ell^s_j = -\infty, \ell^l_j \in \mathbb{R}$ resp. $u^s_j = \infty, u^l_j \in \mathbb{R}$. In what follows, we will extend the notation introduced in Section 2.2 with $\mathcal{B}^{ij}_{\text{slack}}$ and $\mathcal{B}^{ij}_{\text{surplus}}$ denoting $\mathcal{B}^{j}_{\text{slack}}$ and $\mathcal{B}^{j}_{\text{surplus}}$ applied to constraint i and variable j, respectively. The procedure presented in Algorithm 1 computes $\overline{\ell}$ and $\overline{u}$.

The procedure starts by setting $\overline{\ell} = \ell^s$ and $\overline{u} = u^s$ and will proceed to iteratively update these bounds until they are all weakest bounds. Up to Lines 18 and 21, the procedure is very similar to the usual bounds propagation: it evaluates (6a), (6b), and (7) on the latest available bounds for all constraints and variables. As the bounds which start as finite values are already weakest by definition, the first part of the checks in Lines 18 and 21 makes sure that these variables are not considered. For bounds that are infinite at the start, the algorithm checks if the new candidate is finite. If so, the new candidate becomes the weakest bound incumbent if the current weakest bound is infinite, or the new candidate is weaker than the current one. This process then repeats in iterations until no further weakenings are possible. Notice that the *constraint marking mechanism*, implemented in Lines 1, 7, 8, 24, and 25 is not necessary for the correctness of the weakest bounds procedure, but as it can substantially speed up the execution of the algorithm, we include it in the pseudocode.

4 An Algorithm-Independent Measure of Progress

As pointed out in Section 3.1, we will measure the ability of an IBTA to reduce infinite bounds to some finite values separately from the improvements of finite bounds. Section 4.1 presents the functions measuring infinite

Algorithm 1 The Weakest Bounds Algorithm

Input: System of m linear constraints $\underline{\beta} \leq \sum_{i=1}^{n} a_i x_i \leq \overline{\beta}$, $\ell \leq x \leq u$
Output: Weakest variable bounds $\overline{\ell}$ and $\overline{u}$

mark all constraints
`bound_change_found` $\leftarrow$ `true`
$\overline{\ell} = \ell, \overline{u} = u$
while `bound_change_found` **do**
  `bound_change_found` $\leftarrow$ `false`
  for each constraint i **do**
    if i marked **then**
      unmark i
      for each variable j such that $a_{ij} \neq 0$ **do**
        if $a_{ij} > 0$ **then**
          $\ell_j^{\text{new}} = \mathcal{B}_{\text{slack}}^{ij}(\overline{\ell}, \overline{u})$
          $u_j^{\text{new}} = \mathcal{B}_{\text{surplus}}^{ij}(\overline{\ell}, \overline{u})$
        else
          $\ell_j^{\text{new}} = \mathcal{B}_{\text{surplus}}^{ij}(\overline{\ell}, \overline{u})$
          $u_j^{\text{new}} = \mathcal{B}_{\text{slack}}^{ij}(\overline{\ell}, \overline{u})$
        if $x_j \in \mathbb{Z}$ **then**
          $\ell^{\text{new}} = \lceil \ell_j^{\text{new}} \rceil, u_j^{\text{new}} = \lfloor u_j^{\text{new}} \rfloor$
        if $\ell_j = -\infty$ and $\ell_j^{\text{new}} \in \mathbb{R}$ and ($\overline{\ell}_j = -\infty$ or ($\overline{\ell}_j \in \mathbb{R}$ and $\ell_j^{\text{new}} < \overline{\ell}_j$)) **then**
          $\overline{\ell}_j \leftarrow \ell_j^{\text{new}}$
          `bound_change_found` $\leftarrow$ `true`
        if $u_j = \infty$ and $u_j^{\text{new}} \in \mathbb{R}$ and ($\overline{u}_j = \infty$ or ($\overline{u}_j \in \mathbb{R}$ and $u_j^{\text{new}} > \overline{u}_j$)) **then**
          $\overline{u}_j \leftarrow u_j^{\text{new}}$
          `bound_change_found` $\leftarrow$ `true`
        if `bound_change_found` **then**
          mark all constraints k such that $a_{kj} \neq 0$
return $\overline{\ell}$, $\overline{u}$

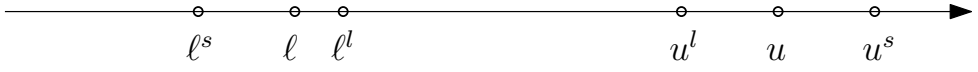

Figure 1: Schematic representation of the starting (index s), current (no index) and limit bounds (index l) for a given variable on the real line. In this example, $\ell^s = \overline{\ell} \in \mathbb{R}$ and $u^s = \overline{u} \in \mathbb{R}$.

domain reductions, while Section 4.2 presents the functions measuring the progress in finite domain reductions.

As before, we denote the starting bounds of a variable j as ℓ_j^s and u_j^s, the weakest bounds as $\overline{\ell}_j$ and $\overline{u}_j$, the limit bounds as ℓ_j^l and u_j^l, and the bounds at a given point in time during the propagation as ℓ_j and u_j. Recall that the following relations hold: $\ell_j^s \leq \ell_j \leq \ell_j^l$ and $u_j^s \geq u_j \geq u_j^l$ for all $j \in \{1, \ldots, n\}$. Additionally, if $\ell_j \in \mathbb{R}$, then $\ell_j^l \in \mathbb{R}$ and $\ell_j^s \leq \overline{\ell}_j \leq \ell_j \leq \ell_j^l$. Likewise, if $u_j \in \mathbb{R}$ then $u_j^l \in \mathbb{R}$ and $u_j^s \geq \overline{u}_j \geq u_j \geq u_j^l$. Lastly, if $\ell_j^s \in \mathbb{R}$ then $\overline{\ell}_j = \ell_j^s$ and if $u_j^s \in \mathbb{R}$ then $\overline{u}_j = u_j^s$. Figure 1 illustrates example starting, current, and limit bounds of a given variable on the real line.

4.1 Measuring Progress in Infinite Domain Reductions

As bounds propagation has a unique fixed point to which it converges, we know the state of the algorithm at both the beginning and the end (a given bound is either finite or infinite). Denote by $n^{\text{total}} \in \mathbb{N}$ the total number of bounds that change from an infinite to some finite value between the starting and the limit bounds of the problem, and by $n^{\text{current}} \in \mathbb{N} \leq n^{\text{total}}$ the number of infinite bounds reduced to finite values by a given IBTA at

a given point during its execution:

$$n^{\text{total}} = |\{j = 1, \ldots, n : \ell_j^s = -\infty, \ell_j^l \in \mathbb{R}\}| + |\{j = 1, \ldots, n : u_j^s = \infty, u_j^l \in \mathbb{R}\}|, \tag{8a}$$

and,

$$n^{\text{current}} = |\{j = 1, \ldots, n : \ell_j^s = -\infty, \ell_j \in \mathbb{R}\}| + |\{j = 1, \ldots, n : u_j^s = \infty, u_j \in \mathbb{R}\}|. \tag{8b}$$

Then, the progress in infinite domain reductions of the IBTA at that point is calculated as:

$$\mathcal{P}^{\text{inf}} = \frac{n^{\text{current}}}{n^{\text{total}}}, n^{\text{total}} \neq 0. \tag{9}$$

Observe that the total number of infinite domain reductions n^{total} is algorithm-independent and can be precomputed from the starting and the limit bounds. Because IBTAs never relax bounds, $\mathcal{P}^{\text{inf}}$ is trivially non-decreasing.

4.2 Measuring Progress in Finite Domain Reductions

The concept of the weakest variable bounds developed in Section 3.2 gives us a natural starting point for finite domain reductions. As bounds propagation converges towards its unique fixed point, the endpoint is also well defined. Notice that the bounds which are infinite at the endpoint, also had to be infinite at the starting point, meaning that no change was made on this bound. The rest of the bounds are either infinite at the beginning, in which case we can compute the weakest bound by Algorithm 1, or the bound is finite at both the start and the end.

Our main approach is to measure the relative progress of each individual bound from its weakest value towards the limit value. Given a variable $j \in \{1, \ldots, n\}$ we will denote by $\mathcal{P}_{\ell_j} \in \mathbb{R}$ and $\mathcal{P}_{u_j} \in \mathbb{R}$ the scores which measure the amount of progress made on its lower and upper bounds ℓ_j and u_j, respectively, at a given point in time. Afterward, we will combine the scores of all the variable bounds into the global progress in the form of a single scalar value $\mathcal{P}^{\text{fin}} \in \mathbb{R}$, which measures the global progress in finite domain reductions at a given point in time.

For variable j, $\mathcal{P}_{\ell_j}$ and $\mathcal{P}_{u_j}$ are computed as

$$\mathcal{P}_{\ell_j} = \begin{cases} \frac{\ell_j - \overline{\ell}_j}{\ell_j^l - \overline{\ell}_j} & \text{if } \ell_j > \overline{\ell}_j \text{ and } \overline{\ell}_j \neq \ell_j^l \\ 0 & \text{otherwise} \end{cases}, \tag{10a}$$

and

$$\mathcal{P}_{u_j} = \begin{cases} \frac{\overline{u}_j - u_j}{\overline{u}_j - u_j^l} & \text{if } u_j < \overline{u}_j \text{ and } \overline{u}_j \neq u_j^l \\ 0 & \text{otherwise} \end{cases}. \tag{10b}$$

Given the vectors of scores for individual bounds $\mathcal{P}_\ell \in \mathbb{R}^n$ and $\mathcal{P}_u \in \mathbb{R}^n$, we calculate $\mathcal{P}^{\text{fin}}$ as

$$\mathcal{P}^{\text{fin}} = \|\mathcal{P}_\ell\|_1 + \|\mathcal{P}_u\|_1 = \sum_j (\mathcal{P}_{\ell_j} + \mathcal{P}_{u_j}), \tag{11}$$

where $\|\cdot\|_1$ denotes the ℓ_1 norm. It holds that $\mathcal{P}_{\ell_j}, \mathcal{P}_{u_j} \in [0, 1]$ and

$$\mathcal{P}^{\text{fin}} \leq |\{j = 1, \ldots, n : \overline{\ell}_j \neq \ell_j^l\}| + |\{j = 1, \ldots, n : \overline{u}_j \neq u_j^l\}|. \tag{12}$$

This maximum score is algorithm-independent and can be precomputed for each instance. This makes it possible to normalize the maximum score to, e.g., 100%. Again, because IBTAs never relax bounds, this progress function is trivially non-decreasing.

4.3 Implementation Details

To precompute $\overline{\ell}$ and $\overline{u}$, we implemented Algorithm 1. To obtain ℓ^l and u^l, any correct bounds propagation algorithm can be run on the original problem, assuming that it propagates the problem to the fixed point (no tolerance-based stopping criteria).

Computing the progress measure is expensive relative to the amount of work that bounds propagation normally performs. Hence, it can considerably slow down the execution and incur unrealistic runtime measurements. To avoid this effect, we proceed as follows in our implementation. First, we run the bounds propagation algorithm together with progress measure computation and record the scores after each round. Then, we run the same bounds propagation algorithm but without the progress measure calculation and record the time elapsed to the end of each round. This gives us progress scores and times for each round, but also the time it took to reach the scores at the end of each round.

5 Applications of the Progress Measure

In this section, we apply the progress measure in order to answer two questions of practical relevance. In Section 5.1, we first describe the experimental setup that will form the base for subsequent evaluations. In Section 5.2 we show that MIP instances in practice rarely cause IBTAs to stall prematurely, i.e., have very slow progress followed by larger improvements thereafter, a concern brought up in [8] (see Section 2.3). In Section 5.3, we show that the newly-developed GPU-based propagation algorithm from [26] is even more competitive in a practical setting than reported in the original paper.

5.1 Experimental Setup

We will refer to two linear constraint propagation algorithms:

1. ***gpu_prop*** is the GPU-based algorithm from [26], and
2. ***seq_prop*** is the canonical sequential propagation as described in e.g. [2]. Our implementation closely follows the implementation in the academic solver SCIP [12].

We use the MIPLIB 2017 test set, which is currently the most adopted and widely used testbed of MIP instances [13]. This test set contains 1065 instances, however, the open-source MIP file reader we used had problems with reading 133 instances, leaving the test set at 932 instances. On 72 instances *gpu_prop* and *seq_prop* failed to obtain the same fixed point (due to e.g., numerical difficulties and other problems), and we remove these instances from the test set as well. Additionally, we impose an iteration limit of 100 for both propagation algorithms, with 2 instances hitting this limit.

During MIP solving, the case where no bound changes are found during propagation is valid and common. However, this is of no interest to us here, as we could make no measurements of progress. There are 310 such instances in the test set. Furthermore, 8 instances with challenging numerical properties showed inconsistent behavior with our implementations, and we remove these instances from the test set as well. Finally, the test set used for the evaluations is left with 540 MIP instances.

In terms of hardware, we execute the *gpu_prop* algorithm on a NVIDIA Tesla V100 PCIe 32GB GPU, and the *seq_prop* algorithm on a 24-core Intel Xeon Gold 6246 @ 3.30GHz with 384 GB RAM CPU. All executions are performed with double-precision arithmetic.

As we use this test set to measure the progress of propagation algorithms, they were run until the fixed point is reached with the progress recorded as described in Section 4.3. In this setting, IBTAs terminate after no bound changes are found at a given propagation round. What this means is that the last two rounds will both have the same maximum score (no bound changes in the last round). Because this feature reflects the design of the algorithms, in the results we assume that the maximum score is reached after the last round, and not after the second-to-last round. This is equivalent to removing the second-to-last round. On the other hand, when the (finite or infinite) score does not change its value between two rounds which are not the last and the second-to-last one, we assume that the score is reached at the first time when it is recorded.

Due to implementation reasons, we will sample progress after each propagation round of an algorithm, rather than after every single bound change. Then, we use linear interpolation to build the progress functions $\mathcal{P}^{\text{fin}}$ and $\mathcal{P}^{\text{inf}}$ and thus obtain an approximation of the true progress function.

5.2 Analyzing Premature Stalling in Linear Constraint Propagation

First, we have to quantitatively define the premature stalling effect. The danger it poses is that the stopping criteria might terminate the algorithm after an iteration with slow progress, and potentially miss on substantial improvements later on. While infinite domain reductions are usually easy to find by bounds propagation algorithms, they are nevertheless considered significant and the algorithm is usually not stopped after an iteration that contains these tightenings [2]. Accordingly, we will reflect this in our premature stalling effect definition.

We slightly adapt the notation introduced in Section 4.2 and define the progress in finite domain reductions as a function of time denoted by $\mathcal{P} : [0, 100] \to [0, 100]$. Observe that the input (time) and output (progress) of this function are normalized to values between 0 and 100. In this notation we assume that $\mathcal{P}$ is continuous and twice differentiable, however, in practice, the progress is sampled only after each propagation round and $\mathcal{P}$ built by linear interpolation. In our implementation, we approximate the derivatives of $\mathcal{P}$ by second-order accurate central differences in the interior points and either first or second-order accurate one-sided (forward or backward) differences at the boundaries [22, 11]. Additionally, given a propagation round r, $t(r)$ denotes the normalized time at the end of propagation round r. All derivates are w.r.t. time: $\mathcal{P}' = \frac{d}{dt}\mathcal{P}$. We denote by $k \in \mathbb{N}$ the number of iterations the propagation algorithm takes to reach the fixed point and by $\ell^r, u^r \in \mathbb{R}^n$ the arrays of lower and upper bounds at iteration r, respectively. Then, the premature stalling effect is defined as follows.

Definition 5. *Let $\mathcal{P}$ be a progress function of finite domain reductions for the propagation of a given MIP instance. Then, the propagation algorithm is said to* prematurely stall *with coefficients $p, q \in \mathbb{R}_{\infty \geq 0}$ at round $r \in \{2, \ldots, k\}$ if the following conditions are true:*

1. *there does not exist $j \in \{1, \ldots, n\}$ such that $\ell_j^{r-1} = -\infty$ and $\ell_j^r \in \mathbb{R}$,*
2. *there does not exist $j \in \{1, \ldots, n\}$ such that $u_j^{r-1} = \infty$ and $u_j^r \in \mathbb{R}$,*
3. *$\mathcal{P}'(t(r)) < p$, and*
4. *there exists $x \in [t(r), 100]$ such that $\mathcal{P}''(x) > q$.*

The first two conditions simply state that there were no infinite domain reductions in round r. To understand the third condition, let $p = 0.1$ at r. This would mean that the algorithm is progressing at a rate of 1 percent of progress in 10 percent of the time at r (recall the normalized domains of $\mathcal{P}$). Taking another derivative and looking at the remainder of the time interval reveals if this rate will increase (is greater than 0), meaning that there are bigger improvements to follow than the improvements the algorithm is currently making. The parameter $q \geq 0$ allows quantification of increase in size of these improvements. Also, recall from Section 4.2 that $\mathcal{P}$ is non-decreasing and hence $\mathcal{P}'(t) \geq 0$ for all $t \in [0, 100]$. With this, we can now detect instances where slow progress is followed by a significant increase in improvements.

Table 1 reports the number of premature stalls in the test set for several different combinations of parameters p and q. Notice that the 310 instances for which no bound changes are found cannot stall by definition. Additionally, 57 instances in the test set only recorded infinite domain reductions, and these instances also cannot prematurely stall by definition. The results of testing the remaining 432 instances which do record at least one finite domain reduction for premature stalling are shown in Table 1.

Let us first look into the results for *seq_prop*. From the first row of the table, we can see that only 48 instances experience any kind of increase in the second derivative during the execution, i.e., the improvements get smaller in time for all but 48 instances in the test set (equivalently, $\mathcal{P}$ is concave for all but 48 instances). From the second row, we can see that among these 48 instances that experience any kind of second derivative increase, 14 experience slow progress of $p = 0.1$ at least once during their execution. Among these, only 1 instance experiences an increase in second derivative of more than $q = 0.2$ following the slow progress of $p = 0.1$. If we further restrict the increase in the second derivative to $q = 0.5$, then no instances are shown to stall prematurely.

Table 1: Number of premature stalls in the test for different values of parameters p and q.

		# stalls	
p	**q**	*seq_prop*	*gpu_prop*
∞	0.0	48	44
0.1	0.0	14	18
0.1	0.2	1	0
0.1	0.5	0	0
0.5	0.5	1	0
0.5	2.0	0	0

In the last row we see that even if the slow progress is relaxed to $p = 0.5$, there are no instances that record a more significant increase in the second derivative of 2.0.

Additionally, even though *gpu_prop* performed similarly to *seq_prop* with respect to stalling, we can still observe that it is on average less susceptible to premature stalling than *seq_prop*, as it recorded a smaller or equal amount of instances with premature stalling for all but one parameter combinations.

We conclude that in practice, the premature stalling effect seems to occur only rarely and on individual instances. This shows that termination criteria based on local progress are reasonable.

5.3 Analyzing GPU-parallel Bounds Propagation in Practice

As pointed out in Section 2.3, *gpu_prop* traverses a potentially different sequence of bounds from the start to the fixed point than *seq_prop*. Because of this, computational experiments in [26] report the speedup of *gpu_prop* over *seq_prop* for propagation runs to the fixed point. As bounds propagation is stopped early in practice, we will now use the progress measure to compare the two algorithms when stopped at different points in the execution. For each instance in the set, given a progress value $x \in [0, 100]$, the speedup of *gpu_prop* over *seq_prop* is computed by $t_x^{\text{seq\_prop}}/t_x^{\text{gpu\_prop}}$, where t_x is the wall-clock time the algorithm takes to reach progress value x.[2] Then, the geometric mean of speedups over all the instances in the test set is reported. The results are shown on Figure 2. When a given instance only has bound changes in the infinite phase, it is excluded from the finite phase comparisons (57 instances). Likewise, instances with only finite progress are removed from the infinite phase (164 instances).

As we can see, for the propagation to the fixed point (100 percent progress), *gpu_prop* is about 4.9 times faster than *seq_prop* in finite domain reductions. For the infinite domain reductions, *gpu_prop* is a factor of about 5.4 times faster than *seq_prop*. Next, we can see that the speedup is minimal at the fixed point, i.e., for any progress value between 10 and 100, *gpu_prop* increases its speedup over *seq_prop* compared to the fixed-point speedup. The maximum speedups of around 7.8 for the finite domain reductions and about 7.0 for infinite domain reductions are achieved at the progress of roughly 50 percent. Additionally, notice that in the last few percent of progress there is a steep drop in speedup. This means that even for very weak stopping criteria which would stop the algorithms at the same point just before the limit is reached, *gpu_prop* would significantly increase its speedup over *seq_prop*. We conclude that *gpu_prop* is even more competitive against *seq_prop* in conjunction with stopping criteria than for the case of propagation to the fixed point.

6 Outlook

In this work, we proposed a method to measure progress achieved by a given algorithm in the propagation of linear constraints with continuous and/or discrete variables. We showed how such a measure can be used to answer questions of practical relevance in the field of Mixed-Integer Programming.

One question that remains open is to what extent the finite reference bounds produced by the weakest bounds

[2] For $x = 100$, we get the identical speedup at the fixed point evaluation as done in [26].

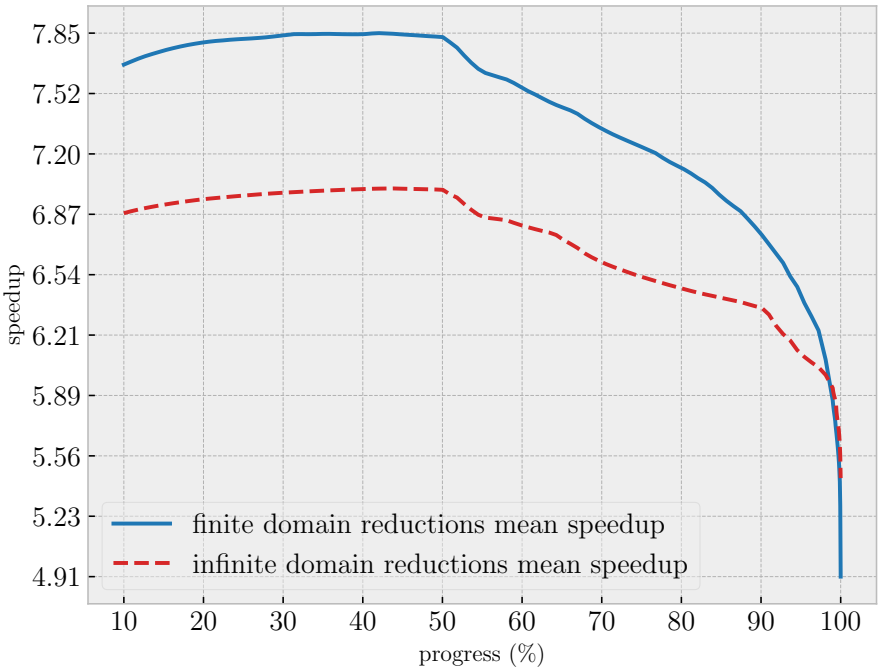

Figure 2: Speedup of the finite and the infinite domain reductions of *gpu_prop* over *seq_prop* for different percentages of progress made.

procedure used here are actually realized by at least one iterative bounds tightening algorithm. The current procedure only guarantees that they are finite if iterative bounds propagation can produce a finite bound, and that no iterative bounds propagation algorithm can produce a weaker bound. A deeper analysis could yield a refined method to produce weakest bounds that are tightest in the sense that they are actually achieved by at least one iterative bounds propagation algorithm. This is part of future research and could provide a stronger version of the framework.

Though our development was described for linear constraints, there are no conceptual barriers that prevent the notion of weakest bounds to be extended to more general classes of constraints. We demonstrated how the key issue of unbounded variable domains can be solved in order to obtain an algorithm-independent measure of progress. In this sense, our method is also relevant for constraint systems on (partially) unbounded domains, where normalization can be nontrivial. An important example is the class of factorable programs from the field of Global Optimization and Mixed-Integer Nonlinear Programming.

References

[1] Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS J. on Computing*, 10(3):287–300, Aug. 1998.

[2] T. Achterberg. *Constraint Integer Programming*. PhD thesis, TU Berlin, 2009.

[3] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, jan 2009.

[4] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020.

[5] T. Achterberg and R. Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. Scil — symbolic constraints in integer linear programming. In R. Möhring and R. Raman, editors, *Algorithms — ESA 2002*, pages 75–87, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[7] I. Aron, J. N. Hooker, and T. H. Yunes. Simpl: A system for integrating optimization techniques. In J.-C. Régin and M. Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 21–36, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] P. Belotti, S. Cafieri, J. Lee, and L. Liberti. Feasibility-based bounds tightening via fixed points. In W. Wu and O. Daescu, editors, *Combinatorial Optimization and Applications, Proc. of COCOA 2010*, pages 65–76, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] L. Bordeaux, G. Katsirelos, N. Narodytska, and M. Y. Vardi. The complexity of integer bound propagation. *J. Artif. Int. Res.*, 40(1):657–676, Jan. 2011.

[10] C. W. Choi, W. Harvey, J. H. M. Lee, and P. J. Stuckey. Finite domain bounds consistency revisited. In A. Sattar and B.-h. Kang, editors, *AI 2006: Advances in Artificial Intelligence*, pages 49–58, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] B. Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51:699–706, 1988.

[12] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.

[13] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, 2019.

[14] W. Harvey and P. J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.

[15] T. Koch, A. Martin, and M. E. Pfetsch. *Progress in Academic Computational Integer Programming*, pages 483–506. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[16] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[17] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, page 232–238, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[18] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[19] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 02 1998.

[20] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, ECAI'88, page 651–656, USA, 1988. Pitman Publishing, Inc.

[21] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., 1988.

[22] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37. Springer, 2007.

[23] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., USA, 2006.

[24] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[25] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), Dec. 2008.

[26] B. Sofranac, A. Gleixner, and S. Pokutta. Accelerating domain propagation: An efficient gpu-parallel algorithm over sparse matrices. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 1–11, 2020.

[27] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1):139–164, 1998.