Strong Bounds Consistencies and Their Application to Linear Constraints

Christian Bessière, Anastasia Paparrizou, Kostas Stergiou

Strong Bounds Consistencies and Their Application to Linear Constraints*

Christian Bessiere
CNRS, University of Montpellier
Montpellier, France
bessiere@lirmm.fr

Anastasia Paparrizou
CNRS, University of Montpellier
Montpellier, France
paparrizou@lirmm.fr

Kostas Stergiou
University of Western Macedonia
Kozani, Greece
kstergiou@uowm.gr

Abstract

We propose two local consistencies that extend bounds consistency (BC) by simultaneously considering combinations of constraints as opposed to single constraints. We prove that these two local consistencies are both stronger than BC, but are NP-hard to enforce even when constraints are linear. Hence, we propose two polynomial-time techniques to enforce approximations of these two consistencies on linear constraints. One is a reformulation of the constraints on which we enforce BC whereas the other is a polynomial time algorithm. Both achieve stronger pruning than BC. Our experiments show large differences in favor of our approaches.

Introduction

Generalized Arc Consistency (GAC) and Bounds Consistency (BC) are the two local consistencies that are predominantly used for propagation by finite domain constraint solvers. Many stronger local consistencies based on GAC have been proposed, both for binary and non-binary constraints, e.g. (Debruyne and Bessière 2001; Bessiere, Stergiou, and Walsh 2008). However, similar consistencies that are based on BC have been comparatively overlooked.

There is a also a considerable body of work dedicated to the study of propagation methods for linear constraints (Zhang and Yap 2000; Harvey and Stuckey 2003; Apt and Zoeteweij 2007; Bordeaux et al. 2011). However, these works focus almost exclusively on BC propagation. One exception is (Trick 2003) where the idea of considering many linear constraints simultaneously to strengthen propagation was investigated in the context of knapsack constraints. Another exception is Singleton Bounds Consistency (SBC), which has been mainly studied on numerical CSPs (Lhomme 1993).

In this paper we introduce rBC2 and rBCall, two new strong local consistencies for non-binary constraints that extend BC. rBC2 is related to the domain filtering consistency relational Path Inverse Consistency (rPIC) and rBCall is related to maxRPWC (Bessiere, Stergiou, and Walsh 2008). The application of rBC2 and rBCall results in the shrinking of domain bounds just like BC, but unlike BC it simultaneously considers combinations of constraints as opposed to single constraints. Naturally, this results in stronger filtering.

We present theoretical results that investigate the pruning power of the new local consistencies. For example, we show that BC is strictly weaker than rBCall and rBC2, which are incomparable to SBC, just as GAC is strictly weaker than maxRPWC and rPIC, which are incomparable to SAC. These hold even when constraints are linear inequalities where we know that GAC is equivalent to BC (Zhang and Yap 2000). However, the stronger pruning of rBC2 and rBCall comes at a cost: we prove that rBC2 and rBCall are NP-hard to enforce even when constraints are linear.

Thereafter, we focus exclusively on linear constraints. This is the most common class of constraints on which BC propagators are applied. We propose two polynomial-time techniques to enforce approximations of rBC2 and rBCall that are still stronger than BC. The first technique is a reformulation of the constraints which introduces extra variables to capture intersections (i.e., shared variables) between constraints. Then we simply enforce BC on the reformulation. This is a flexible technique that depending on the extra variables introduced achieves approximations of varying pruning strength. The drawback, that becomes evident on dense problems, is its memory requirements. The second technique is a polynomial time algorithm that operates on the original model of the problem. This algorithm exploits intersections between pairs of constraints to approximate rBC2. Both techniques can be easily crafted in CP solvers to enhance their pruning power on linear constraints.

Finally, we make an experimental study on various types of problems. Results demonstrate that both of our techniques can outperform BC, being exponentially better in many cases. The reformulation technique is faster on sparse problems, but the algorithm is the preferred technique on dense problems with many constraint intersections.

Background

A (finite domain) *Constraint Satisfaction Problem* (CSP) is defined as a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, with maximum cardinality d, and $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c is a pair $(var(c), rel(c))$, where

*Funded by the EU project ICON (FP7-284715).

$var(c) = (x_1, \ldots, x_m)$ is an ordered subset of $\mathcal{X}$ called the *scope* of c, and $rel(c)$ is a subset of the Cartesian product $D(x_1) \times \ldots \times D(x_m)$ that specifies the allowed combinations of values for the variables in $var(c)$. Each tuple $\tau \in rel(c)$ is an ordered list of values $(v_1, \ldots, v_m)$ such that $v_j \in D(x_j), j = 1, \ldots, m$. Given a tuple $\tau \in rel(c)$, we denote by $\tau[x]$ the projection of τ on a variable $x \in var(c)$.

An important class of non-binary constraints is the class of linear arithmetic constraints. Such a constraint c with $var(c) = \{x_1, ..., x_m\}$ is of the form $a_1x_1 + a_2x_2 + \ldots + a_mx_m \diamond b$, where $a_i, b \in Z$ and $\diamond \in \{<, >, \leq, \geq, =\}$.

Here we are concerned with domains composed of integers. For any variable x_i, $min_D(x_i)$ will denote the smallest value in $D(x_i)$ and $max_D(x_i)$ the greatest one. These two values are called the *bounds* of $D(x_i)$. Given two constraints c_i and c_j, if $var(c_i) \cap var(c_j) \neq \emptyset$ then we say that the constraints *intersect* or *overlap*.

The most commonly used local consistency is *Generalized Arc Consistency* (GAC). A tuple $\tau = (v_1, \ldots, v_m)$ is a *support* on a constraint c, with $var(c) = \{x_1, \ldots, x_m\}$, iff $\tau \in rel(c)$ and for each $x_i, 1 \leq i \leq m$, $v_i \in D(x_i)$. A value $v_i \in D(x_i)$ is GAC iff for every constraint c, s.t. $x_i \in vars(c)$, there exists a support on c s.t. $\tau[x_i] = v_i$. A variable is GAC iff all values in its domain are GAC.

Bounds consistency (BC) has been proposed in order to overcome the prohibitive cost of GAC on some non-binary constraints. A tuple $\tau = (v_1, \ldots, v_m)$ is a *bound-support* on a constraint c, with $var(c) = \{x_1, \ldots, x_m\}$, iff τ satisfies c and for each $x_i \in var(c)$, $min_D(x_i) \leq v_i \leq max_D(x_i)$. A CSP is BC iff $\forall c_j \in \mathcal{C}$ and $\forall x_i \in \mathcal{X}$, where $x_i \in var(c)$, there exist bound-supports τ_{min} and τ_{max} on c s.t. $\tau_{min}[x_i] = min_D(x_i)$ and $\tau_{max}[x_i] = max_D(x_i)$ (i.e. both bounds belong to a bound-support on c).

Strong local consistencies

Several local consistencies stronger than GAC have also been proposed. Those that only remove values from domains are particularly interesting because they do not alter the constraint (hyper)graph or the relations. Among such consistencies, we find rPIC and maxRPWC, proposed in (Bessiere, Stergiou, and Walsh 2008), and that have inspired our work. A CSP is *relational Path Inverse Consistent* (rPIC) iff for each pair of overlapping constraints, each value of each variable in the first constraint belongs to a support that can be extended into a support of the second constraint. A CSP is *max Restricted Pairwise Consistent* (maxRPWC) iff for each constraint, each value of each variable in this constraint belongs to a support that can be extended into a support of any second constraint overlapping the first.

Singleton Bounds Consistency (SBC) (Lecoutre and Prosser 2006) is a strong local consistency based on BC. It is an adaptation of 3B Consistency (Lhomme 1993) from numerical to finite domain CSPs. A CSP is SBC iff $\forall x_i \in \mathcal{X}$ and $\forall v_i \in \{min_D(x_i), max_D(x_i)\}$ the problem derived from assigning v_i to x_i (i.e., $N|x_i = \{v_i\}$) is BC.

Following (Debruyne and Bessière 2001) we will call a local consistency A stronger than B ($A \succeq B$) iff in any problem in which A holds then B holds, and strictly stronger ($A \succ B$) iff A is stronger and there is at least one problem in which B holds but A does not. Accordingly, A is incomparable to B ($A \not\succ B$) iff none is stronger than the other. The relation $\succ$ is transitive.

Strong Consistencies on Bounds

We now define two new strong bounds consistencies, which are relaxed versions of rPIC and maxRPWC.

Definition 1 (rBC2) A CSP is *relational BC on pairs* (rBC2) iff for each $x_i \in \mathcal{X}$ and each value $v_i \in \{min_D(x_i), max_D(x_i)\}$, for each $c_j \in \mathcal{C}$, where $x_i \in var(c_j)$, and for each $c_l \in \mathcal{C}$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset$, there exist bound-supports τ on c_j and τ' on c_l, s. t. $\tau[x_i] = v_i$ and $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$.

Definition 2 (rBCall) A CSP is *relational BC on all* (rBCall) iff for each $x_i \in \mathcal{X}$ and for each value $v_i \in \{min_D(x_i), max_D(x_i)\}$, for each $c_j \in \mathcal{C}$, where $x_i \in var(c_j)$, there exists a bound-support τ on c_j s. t. $\tau[x_i] = v_i$, and for all $c_l \in \mathcal{C}$ ($c_l \neq c_j$), s.t. $var(c_j) \cap var(c_l) \neq \emptyset$, there exists a bound-support τ' on c_l s.t. $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$.

Comparison of Propagation Strength

We prove the relationships between the aforementioned local consistencies. Surprisingly, all consistencies remain distinct even when we consider linear constraints only. Hence, Propositions 1 to 5 below hold even in the case of linear constraints.

Proposition 1 *rBCall $\succ$ rBC2 $\succ$ BC.*

Proof. By definition, rBCall $\succeq$ rBC2 $\succeq$ BC. Strictness is shown by the following examples.

Consider a problem where $D(x_1) = \{0, \ldots, 10\}$, $D(x_2) = \{0, \ldots, 2\}$ and for $i > 2$ $D(x_i) = D(x'_i) = \{0, 1\}$ and the constraints:

$$c_1 : x_1 + x_2 + 8x_3 - 8x'_3 + 9x_4 - 9x'_4 = 10$$
$$c_2 : x_1 + x_2 + 8x_5 - 8x'_5 + 10x_6 - 10x'_6 = 10$$
$$c_3 : x_1 + x_2 + 9x_7 - 9x'_7 + 10x_8 - 10x'_8 = 10$$

All bounds of all variables are rBC2 whereas value 0 of x_1 is not rBCall because it does not belong to any bound-support that satisfies all three constraints at the same time. Hence, rBCall $\succ$ rBC2.

Now consider a problem with variables x_1 and x_2, domains $D(x_1) = \{1, 2\}$, $D(x_2) = \{0, 1\}$, and constraints $c_1 : x_1 + x_2 \leq 2$ and $c_2 : x_1 - x_2 \geq 1$. The problem is BC. However, value 1 of x_2 is not rBC2 because its only bound-support (1,1) on c_1 is not a bound-support on c_2. Hence, rBC2 $\succ$ BC. ■

Proposition 2 *SBC $\not\succ$ rBC2 and SBC $\not\succ$ rBCall.*

Proof. We need an example that is SBC but not rBC2 and an example that is rBCall but is not SBC. Consider a problem with four variables, all having domain $\{0, \ldots, 10\}$, and two constraints: c_1: $x_1 + x_2 + x_3 + x_4 \leq 20$ and c_2: $x_1 + x_2 + x_3 + x_4 > 20$. rBC2 will immediately detect inconsistency whereas SBC will not remove any value.

Now consider a problem with five variables, all having domain $\{0, 5\}$, and the constraints: $x_1 + x_2 + x_3 \geq 10$,

$x_1 \leq x_4$, $x_3 \leq x_5$ and $x_4 + x_5 \leq 5$. All bounds of all variables are rBCall but value 0 of x_2 is not SBC. Hence, SBC is incomparable to both rBC2 and rBCall. ■

We now compare our new consistencies to the related domain filtering consistencies.

Proposition 3 *maxRPWC $\succ$ rBCall and rPIC $\succ$ rBC2.*

Proof. By definition maxRPWC $\succeq$ rBCall and rPIC $\succeq$ rBC2. Strictness is shown by the following example. Consider a problem with variables $x_1, \ldots, x_4$, domains $D(x_1) = \{0, 1, 2\}$, $D(x_2) = \{0, 1, 3\}$, $D(x_3) = \{0, 2\}$, $D(x_4) = \{-2\}$, and constraints $c_1 : x_1 \leq x_2 - x_3$ and $c_2 : x_3 - x_2 \geq x_4$. The problem is rBCall and rBC2. But the only support for value 2 of x_1 on c_1 (tuple (2,3,0)) cannot be extended to c_2. Thus, if we apply maxRPWC or rPIC, value 2 of x_1 will be deleted. ■

Proposition 4 *rBCall $\not\succ$ GAC and rBC2 $\not\succ$ GAC.*

Proof. Consider a problem with variables x_1, x_2, domains $D(x_1) = \{0, 1, 2\}, D(x_2) = \{0, 2\}$, and constraint $x_1 = x_2$. This problem is already rBCall and rBC2 whereas GAC will remove value 1 from x_1. Consider the second example in the proof of Proposition 1. It is already GAC whereas it is neither rBCall nor rBC2. ■

Proposition 5 *rBCall $\not\succ$ rPIC.*

Proof. We need an example that is rPIC but not rBCall and another example that is rBCall but is not rPIC. For the former case, recall the first example in Proposition 1. If we replace the domain of x_1 by $\{0, 8, 9, 10\}$, then it is rPIC whereas rBCall still prunes value 0 of x_1. For the later, in the proof of Proposition 4 we have an example where rPIC is stronger than rBCall. ■

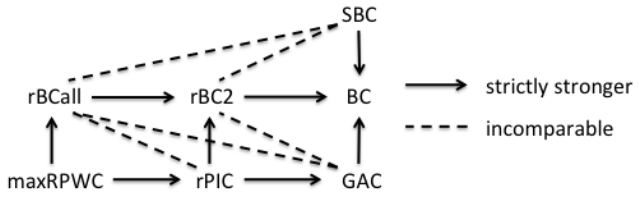

Figure 1: Summary of relationships.

Complexity of Strong Bounds Consistencies

We now analyse the complexity of enforcing rBC2 and rBCall. As shown in (Bessiere 2006), on general constraints, even the simple BC on binary constraints is NP-hard. So we concentrate on the case of linear constraints. In this case, enforcing BC on inequalities ($\{\leq, \geq\}$) is known to be linear (Zhang and Yap 2000). But as soon as we move to stronger bounds consistencies or allow equations, enforcing consistency is hard.

Proposition 6 ((Choi et al. 2006)) *Enforcing BC on linear equations is NP-hard.*

Proposition 7 *Enforcing rBC2 (or rBCall) on linear inequalities is NP-hard even when the inequalities have the same coefficients on the overlapping variables.*

Proof. We reduce SUBSETSUM to the problem of enforcing rBC2. Consider the SUBSETSUM instance $S = \{a_1, \ldots, a_n\}$ and integer k. Enforcing rBC2 on the two inequalities $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq k$ and $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq k$, with $D(x_i) = \{0, 1\}$ for all i is equivalent to GAC because domains are binary. Hence, rBC2 will finish without detecting a wipe out iff there is a solution to the constraint. Now, if I is a solution to the constraint, the set $\{a_i \mid I[x_i] = 1, i \in 1..n\}$ is a solution to the SUBSETSUM problem. ■

Enforcing Strong Bounds Consistencies

We have shown in the previous section that there is no way to enforce rBC2 and rBCall in polynomial time, unless P=NP. In this section we propose two polynomial-time techniques that enforce approximations of rBC2 and rBCall. The first one is a reformulation with extra variables whereas the second approach is a polynomial time algorithm that exploits intersections between pairs of constraints.

Constraint Reformulation

The first approach is a modelling reformulation of the original problem through the introduction of extra variables to capture common subexpressions among constraints. BC is enforced on the reformulation. Such a technique has been introduced in (Araya, Neveu, and Trombettoni 2008) for numerical CSPs. It has also been independently investigated in Savile Row for finite domain CSPs (Nightingale et al. 2014).

Allowing extra variables opens the door to many reformulation choices. We present two versions, one that mimics rBC2 whereas the other mimics rBCall. Our first reformulation, called rBC2-Y, approximates rBC2 by introducing an extra variable Y_{ij} each time a pair of constraints c_i and c_j share more than one variable with coefficients that are proportional. For instance, consider the two constraints

$$a_1x_1 + a_2x_2 + a_3x_3 = b_1 \qquad ka_1x_1 + ka_2x_2 + a_4x_4 = b_2$$

We encapsulate the variables that are common (i.e., x_1, x_2) in the extra variable Y and we replace the common subexpression with the Y variable in the two constraints. The problem is rewritten:

$$Y = a_1x_1 + a_2x_2 \qquad Y + a_3x_3 = b_1 \qquad kY + a_4x_4 = b_2$$

Our second reformulation, called rBCall-Y, approximates rBCall by introducing an extra variable Y each time a subset of variables belongs to several (possibly more than two) constraints with proportional coefficients. Consider the constraints $x_1 + x_2 + x_3 + x_4 \leq b_1$, $x_1 + x_2 + x_3 + x_5 \leq b_2$, $x_1 + x_2 + x_4 + x_5 \geq b_3$. rBC2-Y would reformulate with three variables Y_{12}, Y_{13}, Y_{23} representing the pairwise intersections among the three constraints. On top of this, rBCall-Y adds another extra variable $Y_{123} = x_1 + x_2$ to catch the subexpression common to all three constraints and rewrites all three constraints with this extra variable Y_{123}.

The reformulation technique used in Savile Raw is more elaborated than ours, handling an abstract syntax tree to detect subexpressions. Nevertheless, it is incomparable to ours.

In Savile Raw, once a subexpression is reformulated, all conflicting ones[1] become unusable. In rBC2-Y we reformulate all paiwise common subexpressions.

We characterize the amount of pruning that our two reformulations can achieve. As expected, the reformulations achieve stronger pruning than BC. More interestingly, the second reformulation sometimes prunes more than rBC2.

Proposition 8 *rBC2 $\succ$ BC(rBC2-Y) $\succ$ BC, rBCall $\succ$ BC(rBCall-Y) $\succ$ BC(rBC2-Y), and BC(rBCall-Y) $\not\sim$ rBC2.*

Proof. By construction rBC2 $\succeq$ BC(rBC2-Y) $\succeq$ BC, and rBCall $\succeq$ BC(rBCall-Y) $\succeq$ BC(rBC2-Y).

Consider the first example in Proposition 2, which is BC. rBC2-Y introduces an extra variable $Y = x_1 + x_2 + x_3 + x_4$ and the constraints $Y \leq 20$ and $Y \geq 21$. BC immediately detects inconsistency on this reformulation. Hence, BC(rBC2-Y) $\succ$ BC. Now consider the second example in Proposition 1, which is BC. rBC2-Y and rBCall-Y do not change anything to the original formulation and are thus BC. However, we saw that rBC2, and therefore also rBCall, prunes value 1 from x_2. Thus, rBC2 $\succ$ BC(rBC2-Y) and rBCall $\succ$ BC(rBCall-Y). Consider the first example in Proposition 1, which is already rBC2. rBCall-Y sets $Y = x_1 + x_2$ and replaces it in c_1, c_2, c_3, resulting in c'_1, c'_2, c'_3. BC on c'_1 prunes value 0 from Y, on c'_2 it prunes 1, and on c'_3 it prunes 2. Then, BC on $Y = x_1 + x_2$ prunes 0 from x_1. rBC2-Y on the other hand creates three new variables $Y_{12} = Y_{13} = Y_{23} = x_1 + x_2$ for each of the three intersections between constraints c_1, c_2 and c_3. Then these variables are replaced in c_1, c_2, c_3, resulting in six new constraints. BC thereafter is not able to prune anything from the x_i variables. Hence, BC(rBCall-Y) $\succ$ BC(rBC2-Y). The previous example, together with the second example in Proposition 1 where rBC2 prunes value 1 from x_2 whereas rBCall-Y does nothing, prove that BC(rBCall-Y) $\not\sim$ rBC2. ■

The drawback of the reformulation approach is the extra space required for the new variables and constraints. This is prohibitive in problems with many constraints and many intersections between them. For this reason, we now present an alternative approach that is free from such requirements.

An algorithm approximating rBC2

As opposed to the reformulation approach, the algorithm we will now present does not explicitly represent the extra variables and does not introduce extra constraints to the problem. Instead, each time it revises a pair of intersecting constraints, it computes the interval of values for the intersecting variables "on the fly".

Following (Zhang and Yap 2000), we associate each variable x with the smallest interval $[x] = [lb(x), ub(x)]$ containing $D(x_i)$.

[1]For instance, $x_1 + x_2 + x_3$ and $x_1 + x_2 + x_4$ are conflicting because once $x_1+x_2+x_3+x_4 \leq b_1$ and $x_1+x_2+x_3+x_5 \leq b_2$ are reformulated as $Y_{12} = x_1+x_2+x_3, Y_{12}+x_4 \leq b_1, Y_{12}+x_5 \leq b_2$, the subexpression $x_1 + x_2 + x_4$ only belongs to the constraint $x_1 + x_2 + x_4 + x_5 \geq b_3$.

Given $[x] = [lb(x), ub(x)]$ and $[y] = [lb(y), ub(y)]$, we define the following interval operations:

$$[x] + [y] = [lb(x) + lb(y), ub(x) + ub(y)],$$

$$[x] - a = [lb(x) - a, ub(x) - a],$$

$$a[x] = \begin{cases} [a \cdot lb(x), a \cdot ub(x)], & a > 0 \\ [a \cdot ub(x), a \cdot lb(x)], & a < 0 \end{cases}$$

Given a constraint c defined by $a_1x_1 + \cdots + a_nx_n \diamond b$, with $\diamond \in \{\leq, =, \geq\}$, given intervals on all the variables in $var(c)$, we define $\Pi_{x_i}(c)$, the interval projection of c on x_i:

$$\frac{-1}{a_i}[a_1[x_1]+\cdots+a_{i-1}[x_{i-1}]+a_{i+1}[x_{i+1}]+\cdots+a_n[x_n]-b]$$

We denote by $min(x_i, c)$ and $max(x_i, c)$ the integer values $\lceil lb(\Pi_{x_i}(c)) \rceil$ and $\lfloor ub(\Pi_{x_i}(c)) \rfloor$ respectively.

We now define the function $Proj(x_i, c)$ that will be used in the algorithm as:

$$Proj(x_i, c) = \begin{cases} [min(x_i, c), max(x_i, c)], & if \diamond' \text{ is } = \\ [-\infty, max(x_i, c)], & if \diamond' \text{ is } \leq \\ [min(x_i, c), +\infty], & if \diamond' \text{ is } \geq \end{cases}$$

where $\diamond'$ is the same as $\diamond$ if a_i is positive or $\diamond$ is $=$, and $\diamond'$ swaps $\leq$ to $\geq$ or $\geq$ to $\leq$ in $\diamond$, otherwise.

We now extend the definition of the $Proj$ function to subsets of variables. Given a subset $S = \{x_i, \ldots, x_j\}$ of $var(c)$ and $Y = a_ix_i + \ldots + a_jx_j$ the part of c that involves variables in S, we can define the subset version $\Pi_Y(c)$ of the projection of c on Y as:

$$[a_1[x_1] + \cdots + a_{i-1}[x_{i-1}] + a_{j+1}[x_{j+1}] + \cdots + a_n[x_n] - b]$$

The projection function $Proj(Y, c)$ can be defined in a way similar to $Proj(x_i, c)$.

The algorithm

We now present rBC2-A (from rBC2-Approximation), a filtering algorithm for linear constraints extending the BC algorithm in (Zhang and Yap 2000).

rBC2-A uses a propagation queue Q containing the variables whose domain bounds have been modified since their last propagation. If rBC2-A is used for preprocessing, Q is initialized with all variables. During search, only the variable instantiated at a given node is put in Q (lines 1-2). Once a variable x_i is extracted from Q, all constraints c_j that include x_i are processed. This involves two phases. First, each variable x_k in $var(c_j)$ other than x_i, is revised. The revision of x_k is done in two steps. The first one (line 7) performs the basic BC operation, that is, it narrows the interval $[x_k]$ by intersecting it with $Proj(x_k, c_j)$. In the second step, rBC2-A tries to further narrow $[x_k]$ by exploiting the intersection of c_j with any other constraint c_l by calling procedure *interCheck* with x_k, c_j, and c_l as arguments (line 9).

Procedure *interCheck* takes a variable x, a constraint c, where x is involved, and another constraint c' that intersects with c. It seeks the maximal subset S of $var(c) \cap var(c')$ such that S includes more than one variable, it does not include x, and the coefficients that the variables in this subset have in c and c' are proportional (line 18). It creates a temporary variable Y that replaces variables S in c and c' in the

Algorithm 1: rBC2-A

```
1  if Preprocessing then Q ← V;
2  else Q ← {currently assigned variable};
3  while Q ≠ ∅ do
4      select and remove x_i from Q;
5      foreach c_j ∈ C, s.t. x_i ∈ var(c_j) do
6          foreach x_k ∈ var(c_j) s.t. x_k ≠ x_i do
7              [x_k] ← [x_k] ∩ Proj(x_k, c_j);
8              foreach c_l ∈ C, s.t. |var(c_j) ∩ var(c_l)| > 1 do
9                  interCheck(x_k, c_j, c_l);
10             if D(x_k) = ∅ then return FAILURE;
11             if D(x_k) has been filtered then Q ← Q ∪ {x_k};
12         foreach c_l ∈ C, s.t. |var(c_j) ∩ var(c_l)| > 1 do
13             foreach x_m ∈ var(c_l) s.t. x_m ≠ x_i do
14                 interCheck(x_m, c_l, c_j);
15                 if D(x_m) = ∅ then return FAILURE;
16                 if D(x_m) has been filtered then
                       Q ← Q ∪ {x_m};
17 return SUCCESS;

procedure interCheck(x, c, c');
18 S ← maximal subset of var(c) ∩ var(c'), s.t. |S| > 1
       AND x ∉ S AND the coefficients for S in c and c' are
       proportional;
19 if S ≠ ∅ then
20     [Y] ← Σ_{x_t ∈ S} a_t[x_t];
21     [Y] ← [Y] ∩ Proj(Y, c');
22     [x] ← [x] ∩ Proj(x, c_[Y]);
```

same way as in the reformulation approach and it computes its interval projection $[Y]$ (line 20). $[Y]$ is narrowed through constraint c' (line 21) and then it is used to further narrow $[x]$ through constraint c where variables in S are replaced by Y (line 22). Regarding the first condition in line 18, if the subset S contains only one variable then it cannot trigger any extra pruning on $[x]$ compared to BC. Regarding the second condition, we require that S does not contain x so that $[Y]$ is part of $\Pi_x(c)$ and therefore by narrowing it we can further narrow $[x]$. Regarding the third condition, the requirement for proportional coefficients (e.g. $a_1, \ldots, a_n$ and $k(a_1, \ldots, a_n)$) is a restriction, but they are not uncommon in practice. For instance, unit coefficients occur quite often. The conditions in line 18 need not be evaluated during the algorithm's execution. They can easily be precomputed in a preprocessing step.

Coming back to the description of rBC2-A, the second phase of processing constraint c_j iterates over all constraints that intersect with c_j and for each variable x_m involved in such a constraint c_l, *interCheck* is called (line 14). The reason for performing these calls to *interCheck* is the following: Since x_i has been filtered, some bound-supports on c_j may have been lost and as a result existing bound-supports in c_l may no longer be extendable to c_j. Hence, checking this may result in extra pruning for the variables involved in c_l. However, experiments have showed that a weaker version of the algorithm which does not perform the loop in lines 12-16) is more competitive in cpu times. We call this algorithm rBC2-wA (from weaker Approximation).

The following example illustrates the algorithm.

Example 1 Consider a problem with variables $x_1, \ldots, x_4$, domains $D(x_1) = \{0, \ldots, 4\}$, $D(x_2) = \{0, \ldots, 3\}$, $D(x_3) = \{0, \ldots, 2\}$, $D(x_4) = \{-1\}$, and constraints $c_1 : x_1 \leq x_2 - x_3$ and $c_2 : x_3 - x_2 \geq x_4$. The interval representation of the variable domains is as follows: $[x_1] = [0, 4]$, $[x_2] = [0, 3]$, $[x_3] = [0, 2]$, $[x_4] = [-1, -1]$. First the interval projection of c_1 on x_1 is computed: $\Pi_{x_1}(c_1) = [[x_2] - [x_3]] = [-2, 3]$, which means that $Proj(x_1, c_1) = [-\infty, 3]$. $[x_1]$ will be narrowed in line 7. The new interval will be $[x_1] = [x_1] \cap Proj(x_1, c_1) = [0, 3]$. Now the algorithm will call procedure *interCheck* to further narrow $[x_1]$ by considering the intersection of c_1 with c_2. First a maximal subset of $var(c_1) \cap var(c_2)$ will be sought. Such a subset exists and it is $S = \{x_2, x_3\}$. The interval projection of c_2 on the associated interval Y is: $\Pi_Y(c_2) = [x_4] = [-1, -1]$, which means that $Proj(Y, c_2) = [-1, \infty]$ and $[Y]$ is $[[0, 2] - [0, 3]] = [-3, 2]$. Therefore in line 21 $[Y]$ will be narrowed to $[Y] \cap Proj(Y, c_2) = [-1, 2]$. The next step will be to recompute the interval of x_1 using the updated interval for Y. The interval projection of c_1 on x_1 now is: $\Pi_{x_1}(c_1) = [[x_2] - [x_3]] = [-Y] = [-2, 1]$, which means that $Proj(x_1, c_1) = [-\infty, 1]$. Hence, in line 22 we get $[x_1] = [x_1] \cap Proj(x_1, c_1) = [0, 3] \cap [-\infty, 1] = [0, 1]$. After the processing of x_1 has finished, it will be inserted in Q since its domain has been filtered. and rBC2-A will move on to revise x_2.

Proposition 9 *The worst-case time complexity of algorithm rBC2-A is* $O(e^2 n^3 d)$.

In practice we expect the algorithm to have lower cost since usually a constraint neither includes the entire set of variables nor does it intersect with all other constraints. Also, a variable rarely belongs to all constraints.

Interestingly, rBC2-A constitutes a polynomial filtering algorithm for linear equations which achieves stronger filtering than the existing BC algorithm, while being incomparable in terms of filtering, to the exponential GAC algorithm.

Experiments

We evaluated the practical potential of our approaches on on randomly generated problems and on problems coming from two applications.

Random Problems

We ran experiments on randomly generated problems. All instances have 30 variables, domains being the interval $[0..5]$, and constraints of arity 6 and 9. The constraints are of the form $x_1 + ... + x_k \diamond b$, where $\diamond \in \{<, >\}$ (inequalities) or $\diamond \in \{=\}$ (equations). The parameter b belongs to a randomly selected value in $[10..20]$ and $[15..30]$ for arities 6 and 9 respectively. We defined four classes, ineq-6, ineq-9, eq-6, and eq-9, where ineq (resp.eq) means that all constraints are inequalities (resp. equations), and 6 or 9 represent the fixed arity of all constraints. The number of constraints (30 for ineq-6, 25 for ineq-9, 15 for eq-6, and 12

Table 1: Problems of with inequalities (top) and equations (bottom): Mean CPU time in seconds and mean node visits.

Class		BC	SBC	rBC2-wA	rBC2-A	rBC2-Y	rBCall-Y23
ineq-6	time	187.9	25.2	10.3	10.6	**6.1**	8.8
	#nodes	8,791,903	171,214	130,465	69,820	—	—
ineq-9	time	82.3	78.8	20.2	23.9	**9.8**	31.1
	#nodes	7,490,045	176,400	38,113	26,454	—	—
eq-6	time	271.6	75.5	31.1	21.2	**9.8**	12.0
	#nodes	4,760,257	80,079	1,213,789	529,677	—	—
eq-9	time	512.3	469.1	84.0	83.6	**40.4**	65.7
	#nodes	17,605,430	525,504	344,831	168,472	—	—

for eq-9) corresponds to the cross-over point (center of the phase transition), where the instances are the most difficult to solve. We solved 30 instances per class, with a cutoff limit of 1800 seconds. We compare BC, SBC, rBC2-wA, rBC2-A, rBC2-Y and rBCall-Y23. The latter reformulation uses extra variables to capture common subexpressions between pairs and between triplets of constraints. The pruning achieved on it is in between rBC2-Y and rBCall-Y.

Table 1 reports the results in cpu time and in number of nodes. On problems with inequalities, our proposed techniques are the most efficient in time both on ineq-6 and on ineq-9, sometimes being more than one order of magnitude faster than BC. Comparing our techniques, the reformulation rBC2-Y is the most efficient. The two algorithms display similar performance. It is notable that as the arity increases to arity 9, resulting in more intersections, the stronger reformulation and the stronger algorithm become worse than their respective weak versions. This is due to the cost of the extra variables and constraints that rBCall-Y23 builds and the extra calls to the *interCheck* function that slows down the performance of rBC2-A. Regarding SBC, its performance deteriorates significantly on problems of arity 9.

On problems with equations, the performance of the various techniques is similar to that on inequalities. Our techniques are again one order of magnitude faster than BC. SBC is not competitive at all. Among our techniques, rBC2-Y is the winning one, rBCall-Y23 is a close second, and the algorithms follow, being around two times slower.

When comparing the number of nodes, we observe that BC is the one that explores the largest search tree, up to two orders of magnitude more nodes than our algorithms. SBC is incomparable to our algorithms, sometimes exploring more nodes, sometimes less. This is consistent with our theoretical comparison between rBC2 and SBC. Concerning the reformulation techniques, they use extra variables, which makes a comparison of the search trees irrelevant. We thus did not report their number of nodes.

We also explored problems where our reformulation techniques face a space limitation. rBC2-Y (resp. rBCall-Y23) cannot construct the reformulation when we impose over 220 (resp. 120) and 145 (resp. 60) constraints on problems with arity 6 and 9 respectively. In contrast, the algorithms had no problem dealing with such problems and significantly outperformed BC.

Finally, we compared rBC2-wA to BC and GAC under a similar generation model as here, but this time we randomly created holes in domains. Results, which are not detailed for space reasons, showed that rBC2-wA was by far more efficient than BC and especially GAC, which reached the cutoff limit very often.

Web Services

In our second experiment, we compare the different techniques on large real instances from Web Services. Ruiz Cortés and Martín-Díaz used CP for solving matchmaking problems in Web Services. Their problems have $n = 100$ to 1500 variables, domains size from 0 to 255, and constraints of arity $k = 3$ to 20. Results showed that CP solvers were unable to solve some instances when the size increases (Ruiz Cortés et al. 2005; Kritikos and Plexousakis 2009). We reproduced their experiments.

Figure 2 displays the performance of BC, SBC, rBC2-A, rBC2-wA, and rBC2-Y, as n and k increase. The left (resp. right) plot presents the results by increasing arity (resp. increasing number of variables). We omit rBCall-Y because there is no gain compared to rBC2-Y, whereas memory is exhausted faster.

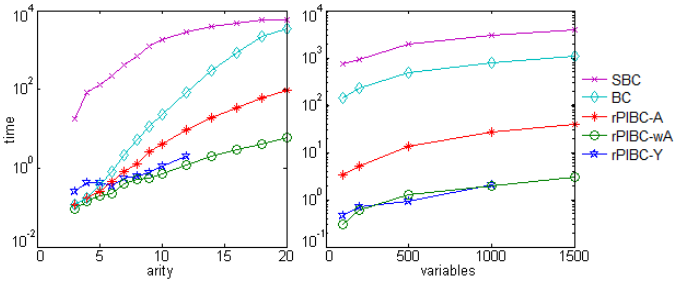

Figure 2: Web Services: CPU time in sec. (log scale).

We observe that, except on small arity constraints, BC is always orders of magnitude slower than rBC2-A, rBC2-wA, and rBC2-Y. SBC is consistently the worst. Comparing our techniques, we see that when arity increases, the performance of rBC2-A deteriorates compared to rBC2-wA. This is because the satisfiable instances appear to be very easy, so rBC2-A suffers from many useless calls to function interCheck. On small problems rBC2-Y and rBC2-wA are the two winners, but when the size increases (arity or number of variables) rBC2-Y exhausts memory.

Computing Lower Bounds in Planning

As a last experiment, we solve CSPs whose solutions provide lower bounds to the number of moves in a 15-puzzle. Using lower bounds as heuristics is often exploited in planning problems (Korf 1985). The example of 15-puzzle was presented by Malte Helmert in his ECAI 2014 invited talk. We define the CSP exactly as Helmert did. Variables U_i, D_i, L_i, R_i represent the minimal number of up, down, left and right moves the tile i must do to reach its final location. Constraints express minimal number of moves for subsets of variables. These constraints are derived from distances to final location or from conflicting positions between tiles. A global constraint forces the sum of all the 60 variables to be less than a given upper bound M. As it is an optimization problem, the CSP is considered as solved once we have found M such that $M-1$ leads to inconsistency and M to a solution.

Table 2 gives average CPU times for 20 instances. As in the two previous experiments, we observe that BC is significantly outperformed by our techniques (up to factor 8). Among our techniques, rBC2-wA is the best, almost twice faster than rBC2-A. As opposed to the previous experiments, the reformulations rBC2-Y and rBCall-Y23 are both slower than the algorithms rBC2-wA and rBC2-A (factor from 1.6 to 3.3). As usual, SBC is by far the worst technique (more than one order of magnitude slower than rBC2-wA and rBC2-A).

Table 2: Lower bounds for the 15-puzzle: CPU time in sec.

BC	SBC	rBC2-wA	rBC2-A	rBC2-Y	rBCall-Y23
39.5	122.5	**4.9**	7.8	12.4	16.4

Discussion

These experiments show the efficiency of our techniques to solve CSPs with equations or inequalities that share common variables. Our techniques significantly outperform existing alternatives such as BC or SBC. These results also show the usefulness of the reformulation technique on sparse problems, and at the same time the practical need for the proposed algorithm that overcomes the space requirements of reformulations when the size of the problem increases.

Conclusion

We have proposed rBC2 and rBCall, two new strong local consistencies that extend BC by taking into account combinations of constraints. We gave a theoretical comparison of the propagation strength of the new consistencies with respect to existing ones. We have shown that these consistencies are intractable to enforce, even on linear constraints. Hence, we proposed two techniques to approximate rBC2 and rBCall in the case of linear constraints. The first one is a reformulation with extra variables. The second is a polynomial filtering algorithm. Both our two techniques achieve stronger pruning than BC. Experimental results demonstrated the potential of strong consistencies that reason on domain bounds.

Acknowledgments

We thank Malte Helmert for the useful input he provided on heuristics for solving the 15-puzzle.

References

Apt, K., and Zoeteweij, P. 2007. An analysis of arithmetic constraints on integer intervals. *Constraints* 12(4):429–468.

Araya, I.; Neveu, B.; and Trombettoni, G. 2008. Exploiting common subexpressions in numerical csps. In *Proceedings of CP'08*, 342–357.

Bessiere, C.; Stergiou, K.; and Walsh, T. 2008. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence* 172(6-7):800–822.

Bessiere, C. 2006. Chapter 3 Constraint Propagation. In Francesca Rossi, P. v. B., and Walsh, T., eds., *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier. 29 – 83.

Bordeaux, L.; Katsirelos, G.; Narodytska, N.; and Vardi, M. 2011. The Ccomplexity of Integer Bound Propagation. *JAIR* 40:657–676.

Choi, C.; Harvey, W.; Lee, J.; and Stuckey, P. 2006. Finite Domain Bounds Consistency Revisited. In *Proceedings of the Australian Conference on AI*, 49–58.

Debruyne, R., and Bessière, C. 2001. Domain Filtering Consistencies. *JAIR* 14:205–230.

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

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.* 27(1):97–109.

Kritikos, K., and Plexousakis, D. 2009. Mixed-Integer Programming for QoS-Based Web Service Matchmaking. *IEEE T. Services Computing* 2(2):122–139.

Lecoutre, C., and Prosser, P. 2006. Maintaining Singleton Arc Consistency. In *3rd International Workshop on Constraint Propagation And Implementation (CPAI'2006)*, 47–61.

Lhomme, O. 1993. Consistency techniques for numeric csps. In *Proceedings of IJCAI-93*, 232–238.

Nightingale, P.; Akgün, Ö.; Gent, I. P.; Jefferson, C.; and Miguel, I. 2014. Automatically improving constraint models in savile row through associative-commutative common subexpression elimination. In *Proceedings of CP'14*, 590–605.

Ruiz Cortés, A.; Martín-Díaz, O.; Durán, A.; and Toro, M. 2005. Improving the Automatic Procurement of Web Services Using Constraint Programming. *International Journal of Cooperative Information Systems* 14(4):439–468.

Trick, M. 2003. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Anals OR* 118:73–84.

Zhang, Y., and Yap, R. H. C. 2000. Arc consistency on n-ary monotonic and linear constraints. In *Proceedings of CP'00*, 470–483.