World Scientific
www.worldscientific.com

STRONG DOMAIN FILTERING CONSISTENCIES FOR NON-BINARY CONSTRAINT SATISFACTION PROBLEMS

KOSTAS STERGIOU

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
konsterg@aegean.gr

Domain filtering local consistencies, such as inverse consistencies, that only delete values and do not add new constraints are particularly useful in Constraint Programming. Although many such consistencies for binary constraints have been proposed and evaluated, the situation with non-binary constraints is quite different. Only very recently have domain filtering consistencies stronger than GAC started to attract interest. Following this line of research, we define a number of strong domain filtering consistencies for non-binary constraints and theoretically compare their pruning power. We prove that three of these consistencies are equivalent to maxRPC in binary CSPs while another is equivalent to PIC. We also describe a generic algorithm for domain filtering consistencies in non-binary CSPs. We show how this algorithm can be instantiated to enforce some of the proposed consistencies and analyze the worst-case complexities of the resulting algorithms. Finally, we make a preliminary empirical study.

1. Introduction

One of the great strengths of Constraint Programming is the exploitation of local consistency techniques to prune inconsistent values from the domains of variables and thus avoid fruitless exploration of the search tree. The most widely studied and used local consistency is generalized arc consistency (GAC). It is widely accepted that "relation filtering" consistencies which alter the structure of the constraint graph or the constraints' relations (e.g. path consistency) tend to be less practical than "domain filtering" consistencies which only remove values from the domains of the variables. As a result, many strong domain filtering consistencies for binary constraints have been proposed and evaluated. For example, inverse and singleton consistencies.[8,5,16] In contrast, little work had been done on such consistencies for non-binary constraints until very recently, whereas a number of consistencies that are stronger than GAC, but not domain filtering, have been developed. For example, pairwise consistency,[10] hyper-m-consistency,[12] relational consistency,[15] and ω-consistency.[13] However, these consistencies are rarely used in practice, mainly because they have a high space complexity.

Very recently, three domain filtering consistencies for non-binary CSPs were introduced and evaluated theoretically and empirically. These are relational path inverse consistency (rPIC), restricted pairwise consistency (RPWC), and max restricted pairwise

consistency (maxRPWC).[14,3,a] All these are stronger than GAC and display promising performance on certain non-binary problems with maxRPWC being the most efficient of the three.

Continuing along the same lines of work, we propose a number of strong domain filtering consistencies for non-binary constraints and study them theoretically and empirically. These new consistencies are the following: max restricted 3-wise consistency and the parametrized max restricted k-wise consistency, relational neighborhood inverse consistency, inverse ω-consistency and extended inverse ω-consistency. To derive these consistencies we are mainly inspired by known relation-filtering consistencies for non-binary problems. In our theoretical study we compare the pruning power of these consistencies, most of which are stronger than maxRPWC, and show what they correspond to when restricted to binary constraints. We prove that three of these consistencies are equivalent to max restricted path consistency (maxRPC) in binary CSPs while another is equivalent to path inverse consistency (PIC). We also describe a generic algorithm that can be used to apply any of the proposed domain filtering consistencies. We show how this algorithm can be instantiated to enforce some of these consistencies and analyze the worst-case complexities of the resulting algorithms. Finally, we give some preliminary experimental results.

2. Background

A *Constraint Satisfaction Problem* (CSP) P is defined as a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. For each variable $x_i \in X$, $D(x_i)$ is the finite domain of its possible values. Each constraint $c_i \in C$ is defined as a pair $(var(c_i), rel(c_i))$, where $var(c_i) = \{x_{j_1}, \ldots, x_{j_k}\}$ is an ordered subset of X called the *scope* of c_i, and $rel(c_i)$ is a subset of the *Cartesian* product $D(x_{j_1}) \mathrm{x} \ldots \mathrm{x} D(x_{j_k})$ that specifies the allowed combinations of values for the variables in $var(c_i)$. Each tuple $\tau \in rel(c_i)$ is an ordered list of values $(a_1, \ldots, a_k)$. A tuple is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable. A constraint c_i can be either defined *extensionally* by explicitly giving relation $rel(c_i)$, or (usually) *intensionally* by implicitly specifying $rel(c_i)$ through a predicate or arithmetic function. For any two constraints c_i and c_j, the set of variables that are involved in both constraints is denoted by $var(c_i) \cap var(c_j)$. If this set is not empty, the constraints *intersect*. We denote by p the maximum number of variables involved in two constraints that intersect. Also, for all triangles of constraints (i.e. sets of three constraints such that any of the three intersects with any other) we denote by p' the maximum number of variables that are involved in one constraint but are not involved in any of the other two.

A binary CSP can be represented by a graph (called constraint graph) where nodes correspond to variables and edges correspond to constraints. A non-binary CSP can be represented by a constraint hypergraph where the constraints correspond to hyperedges connecting two or more nodes.

---

[a]maxRPWC was called pairwise inverse consistency in Ref. 14.

The assignment of value a to variable x_i is denoted by (x_i, a). Any tuple $\tau = (a_1, \ldots, a_k)$ can be viewed as a set of value to variable assignments $\{(x_1, a_1), \ldots, (x_k, a_k)\}$. In this way, an assignment of values to a set of variables $X' \subseteq X$ is a tuple over X'. The set of variables over which a tuple τ is defined is $var(\tau)$. For any subset var' of $var(\tau)$, $\tau[var']$ is the sub-tuple of τ that includes only assignments to the variables in var'. Any two tuples τ and τ' of $rel(c_i)$ can be ordered by the lexicographic ordering $<_l$. In this ordering, $\tau <_l \tau'$ iff there a exists a subset $\{x_1, \ldots, x_j\}$ of c_i such that $\tau[x_1, \ldots, x_j] = \tau'[x_1, \ldots, x_j]$ and $\tau[x_{j+1}] <_l \tau'[x_{j+1}]$. A tuple τ is *consistent*, iff it is valid and for all constraints c_i, where $var(c_i) \subseteq var(\tau)$, $\tau[var(c_i)] \in rel(c_i)$. A *solution* to a CSP (X, D, C) is a consistent tuple assigning all variables in X.

A value $a \in D(x_i)$ is consistent with a constraint c_j, where $x_i \in var(c_j)$, iff $\exists \tau \in rel(c_j)$ such that $\tau[x_i] = a$ and τ is valid. In this case, we say that τ is a GAC-support of (x_i, a) in c_j. A constraint c_j is *Generalized Arc Consistent* (GAC) iff $\forall x_i \in var(c_j)$, $\forall a \in D(x_i)$, there exists a GAC-support for a in c_j. A problem is GAC iff there is no empty domain in D and all the constraints in C are GAC. In binary CSPs, GAC is referred to as *arc consistency* (AC).

Since the allowed tuples of constraints are defined as relations, standard relational operators can be used. The *projection* $\Pi_{var'}\tau$ of a tuple $\tau \in rel(c_i)$ on var' is the subtuple $\tau[var']$. Accordingly, the projection of a constraint c_i on a set of variables var', where $var' \subseteq var(c_i)$ is a new constraint c' where $var(c') = var'$ and $rel(c') = \Pi_{var'}rel(c_i)$. The *join* of two constraints c_i and c_j is a new constraint, denoted by $c_i \bowtie c_j$, where $var(c_i \bowtie c_j) = var(c_i) \cup var(c_j)$ and $rel(c_i \bowtie c_j) = rel(c_i) \bowtie rel(c_j)$. Accordingly, the *join* of two tuples $\tau \in rel(c_i)$ and $\tau' \in rel(c_j)$, denoted by $\tau \bowtie \tau'$, is a tuple such that $(\tau \bowtie \tau')[var(c_i)] = \tau$ and $(\tau \bowtie \tau')[var(c_j)] = \tau'$.

2.1. *Local consistencies*

We now briefly review the most common local consistencies for binary and non-binary CSPs. We assume that any given CSP is *normalized*. That is, multiple constraints on the same variables are combined into one.

2.1.1. *Binary constraints*

A binary problem is (i, j) *consistent* iff it has non-empty domains and any consistent instantiation of i variables can be extended to a consistent instantiation involving j additional variables.[7] A problem is *strong (i, j)-consistent* iff it is (k, j) consistent for all $k \leq i$. Following the definition of (i, j)-consistency, arc consistency is equivalent to $(1, 1)$-consistency. A problem is *path consistent* (PC) iff it is $(2, 1)$-consistent. A problem is *k-consistent* iff it is $(k, 1)$-consistent. A problem is *path inverse consistent* (PIC) iff it is $(1, 2)$-consistent.[8] A problem is *max restricted path consistent* (maxRPC) iff it is $(1, 1)$-consistent and for each value (x_i, a) and variable x_j constrained with x_i, there exists a value $b \in D(x_j)$ that is an AC-support of (x_i, a) and this pair of values is path consistent (i.e. it can be consistently extended to any third variable).[4] A problem is *inverse m-consistent* iff it is $(1, m)$ consistent. A problem is *neighborhood inverse consistent* (NIC)

iff any consistent instantiation of a variable x_i can be extended to a consistent instantiation of all the variables in x_i's neighborhood.[8,b] A problem P is *singleton arc consistent* (SAC)[5] iff it has non-empty domains and for any instantiation (x_i, a) of a variable $x_i \in X$, the resulting subproblem can be made AC.

2.1.2. *Non-binary constraints*

Some local consistencies for binary CSPs can be easily extended to non-binary problems. For example, SAC has been extended to SGAC. However, for other consistencies (e.g. PIC and maxRPC) this extension is not straightforward. In the case of NIC there are two alternative extensions to non-binary constraints. To determine if a value $a \in D(x_i)$ is NIC, we can consider the subproblem consisting of the set of variables $neigh(x_i) = \{x_{i_1}, \ldots, x_{i_m}\}$ involved in a constraint with x_i and the constraints that only include variables from $neigh(x_i)$. Alternatively, we can consider the subproblem consisting of variables $neigh(x_i)$ and all the constraints that include any of these variables (and possibly other variables as well). In the rest of this paper we follow the first definition of NIC for non-binary constraints.

A problem is *relationally arc consistent* (rel AC) iff any consistent instantiation for all but one of the variables in a constraint can be extended to the final variable so as to satisfy the constraint.[15,6] A problem is *relationally path-consistent* (rel PC) iff any consistent instantiation for all but one of the variables in a pair of constraints can be extended to the final variable so as to satisfy both constraints. A problem is *relationally m-consistent* iff any consistent instantiation for all but one of the variables in a set of m distinct constraints can be extended to the final variable so as to satisfy all m constraints. A problem is *relationally (i, m)-consistent* iff any consistent instantiation for i of the variables in a set of m constraints can be extended to all the variables in the set. A problem is *strongly relationally (i, m)-consistent* iff is relationally (j, m)-consistent for every $j \leq i$.

A non-binary problem is *pairwise consistent* (PWC)[12]) iff it has non-empty relations and any consistent tuple in a constraint c_i can be consistently extended to any other constraint.[10] As shown in Ref. 10, applying PWC in a non-binary CSP is equivalent to applying AC in the dual encoding of the problem. PWC has been generalized to *k-wise consistency*[9,11] and *hyper-m-consistency*.[12] A problem is k-wise consistent iff any consistent tuple for a constraint can be consistently extended to any $k-1$ other constraints. A problem is hyper-m-consistent iff any consistent combination of tuples for m-1 constraints can be consistently extended to any mth constraint. As noted in Ref. 12, hyper-m-consistency on a non-binary problem is equivalent to m-consistency on the dual encoding of the problem.

A problem is *ω-consistent* iff any tuple in a constraint c_i can be consistently extended to any other constraint c_j and to all constraints c_k such that $var(c_k) \subseteq var(c_i) \cup var(c_j)$.[13] A problem is *generalized dual arc consistent* (GDAC) iff any tuple in a constraint c_i can be consistently extended to any other constraint c_j and at the same time satisfy all constraints c_k such that $var(c_k) \cap (var(c_i) \cup var(c_j)) \neq \emptyset$.[13]

---

[b]The neighborhood of a variable consists of all variables that are constrained with it.

Following,[5] we call a consistency property A stronger than B iff in any problem in which A holds then B holds, and strictly stronger (written $A \rightarrow B$) iff it is stronger and there is at least one problem in which B holds but A does not. We call a local consistency property A incomparable with B (written $A \otimes B$) iff A is not stronger than B nor vice versa. Finally, we call a local consistency property A equivalent to B (written $A \leftrightarrow B$) iff A is stronger than B and vice versa. Note that relationships $\rightarrow$ and $\leftrightarrow$ are transitive.

3. Strong Domain Filtering Consistencies

In practice, most of the strong local consistency techniques discussed in the previous section have prohibitive space and time complexities. Freuder proposed inverse consistencies as a way to overcome the space problem.[8] Such consistencies require limited space as they only prune domains. When an inverse local consistency is enforced, it removes from the domain of a variable the values that cannot be consistently extended to some additional variables. For example, when enforcing PIC we remove values that cannot be consistently extended to any set of two other variables.

Until the very recent introduction of rPIC, RPWC, and maxRPWC, the study of domain filtering consistencies had been restricted to binary constraints, with the exception of GAC. Experimental results demonstrated that maxRPWC, which is the strongest, is also the most efficient among these three consistencies.[14,3] We will now define a number of new domain filtering consistencies for non-binary problems. These are all strictly stronger than GAC. That is, if applied, they will remove any value that is not GAC. Also, each consistency may remove some additional values according to the property it enforces. For any consistency IC, we say that a variable x_i is IC iff any value $a \in D(x_i)$ is IC. A CSP is IC iff there is no empty domain and all variables are IC. The following definitions specify when a value is IC for a number of different domain filtering consistencies. We first recall the definitions of rPIC and maxRPWC.

Definition 3.1.[15,14] A value $a \in D(x_i)$ is *relational Path Inverse Consistent* (rPIC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, and for each $c_k \in C$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ and a valid tuple $\tau' \in rel(c_k)$ such that $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$.

If rPIC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to a valid tuple in some other constraint c_k that intersects with c_j. Note that if the two constraints do not intersect then any valid tuple in $rel(c_j)$ can be extended to any valid tuple in $rel(c_k)$. Apart from rPIC we can consider other, stronger, inverse relational consistencies such as relational $(1, 3)$-consistency and relational NIC which are defined further below.

Definition 3.2.[3] A value $a \in D(x_i)$ is *max Restricted Pairwise Consistent* (maxRPWC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ s.t. $\forall c_k \in C$, there exists a PW-support τ' of τ in $rel(c_k)$. A tuple τ' is a PW-support of τ iff it is valid and $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$.

If maxRPWC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to a valid tuple in every other constraint (intersecting c_j).

3.1. *Extending rPIC and maxRPWC*

The definition of both rPIC and maxRPWC can be generalized to derive domain filtering consistencies by considering the extensions of a constraint c_j to sets of constraints of various size. To illustrate this we first define relational $(1, 3)$ consistency, as proposed by van Beek and Dechter, and the new consistency maxR3WC. Then we present two general parameterized definitions. The former is the definition of relational $(1, k)$ consistency given in Ref. 6 while the latter introduces a family of domain filtering consistencies for non-binary constraints inspired by the concept of k-wise consistency.

Definition 3.3.[15] A value $a \in D(x_i)$ is *relational* $(1, 3)$-*Consistent* (r$(1,3)$C) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, and for each pair of constraints $c_k, c_l \in C$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ and valid tuples $\tau' \in rel(c_k), \tau'' \in rel(c_l)$ s.t. $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$, $\tau[var(c_j) \cap var(c_l)] = \tau''[var(c_j) \cap var(c_l)]$, $\tau'[var(c_k) \cap var(c_l)] = \tau''[var(c_k) \cap var(c_l)]$.

If r$(1,3)$C is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to valid tuples in some pair of extra constraints.

Definition 3.4. A value $a \in D(x_i)$ is *max Restricted 3-wise Consistent* (maxR3WC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ s.t. $\forall c_k, c_l \in C$ there exist valid tuples $\tau' \in rel(c_k), \tau'' \in rel(c_l)$ s.t. $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$, $\tau[var(c_j) \cap var(c_l)] = \tau''[var(c_j) \cap var(c_l)]$, $\tau'[var(c_k) \cap var(c_l)] = \tau''[var(c_k) \cap var(c_l)]$.

If maxR3WC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to valid tuples in every pair of other constraints.

Definition 3.5.[15] A value $a \in D(x_i)$ is *relational* $(1, m)$-*Consistent* (r$(1,m)$C) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, and for each set of additional $k - 1$ constraints $c_1, \ldots, c_{k-1}$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ s.t. τ can be extended to a valid instantiation on variables $\bigcup_{m=1}^{k-1} var(c_m)$ that satisfies each c_m for $m = 1, \ldots, k - 1$.

If r$(1,k)$C is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to valid tuples in some set of $k - 1$ extra constraints.

Definition 3.6. A value $a \in D(x_i)$ is *max Restricted k-wise Consistent* (maxRkWC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ that is k-wise consistent. That is, iff for any set of additional $k - 1$ constraints $c_1, \ldots, c_{k-1}$, τ can

be extended to a valid instantiation on variables $\bigcup_{m=1}^{k-1} var(c_m)$ that satisfies each c_m for $m = 1, \ldots, k-1$.

If maxRkWC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to valid tuples in every set of $k-1$ extra constraints.

3.2. *Other domain filtering consistencies*

We now introduce three new domain filtering consistencies which are inspired by NIC and ω-consistency.

Definition 3.7. A value $a \in D(x_i)$ is *relational Neighborhood Inverse Consistent* (rNIC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ that can be extended to a solution of the subproblem consisting of the set of variables $X_j = \{var(c_j) \cup var(c_{j_1}) \cup \ldots \cup var(c_{j_m})\}$, where $c_{j_1}, \ldots, c_{j_m}$ are the constraints that intersect with c_j.

If rNIC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to a consistent instantiation of all variables involved in a constraint that intersects with c_j so that all constraints between these variables are satisfied.

Definition 3.8. A value $a \in D(x_i)$ is *inverse ω-consistent* (IωC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ s.t. $\forall c_k \in C$, there exists an ω-support τ' of τ in $rel(c_k)$. A tuple τ' is an ω-support of τ iff it is a PW-support of τ and $\forall c_l \in C$, where $var(c_l) \subseteq var(c_j) \cup var(c_k), (\tau \bowtie \tau')[var(c_l)] \in rel(c_l)$.

If IωC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to a valid tuple in every constraint c_k that intersects with c_j and, at the same time, satisfy all constraints defined on variables $var(c_j) \cup var(c_k)$.

Definition 3.9. A value $a \in D(x_i)$ is *extended inverse ω-consistent* (EIωC) iff $\forall c_j \in C$, where $x_i \in var(c_j)$, there exists a GAC-support τ of (x_i, a) in $rel(c_j)$ s.t. $\forall c_k \in C$, there exists an extended ω-support τ' of τ in $rel(c_k)$. A tuple τ' is an *extended ω-support of τ* iff it is a PW-support of τ and $\forall c_l \in C$, where $var(c_j) \cap var(c_l) \neq \emptyset$ and $var(c_k) \cap var(c_l) \neq \emptyset$, $\Pi_{var(c_l) \cap (var(c_j) \cup var(c_k))}(\tau \bowtie \tau') \in \Pi_{var(c_l) \cap (var(c_j) \cup var(c_k))} rel(c_l)$ and can be extended to a valid tuple in $rel(c_l)$.

If EIωC is applied on a variable x_i it will remove any value $a \in D(x_i)$ such that for some constraint c_j where x_i participates, no GAC-support of (x_i, a) can be extended to a valid tuple in each constraint c_k that intersects with c_j and, at the same time, satisfy all constraints that intersect with both c_j and c_k. The difference between IωC and EIωC is that the former considers a constraint c_l only if it includes variables among $var(c_j) \cup var(c_k)$, while the latter also considers some constraints that include variables among $var(c_j) \cup var(c_k)$ and other variables as well.

c1 : x1,x2,x3,x4

0	0	0	0
0	0	0	1
1	1	0	0
2	0	0	0

c2 : x1,x2,x5

0	1	0
1	1	0
2	0	1

c3 : x3,x5,x6

0	0	0
1	1	0
1	1	1

c4 : x4,x5

0	1
1	0

a)

c1: x1,x2,x3

0	0	0
1	0	1
1	1	0

c2: x2,x3,x4,x5

0	0	0	0
0	1	1	0
1	0	0	1

c3: x4,x5,x6
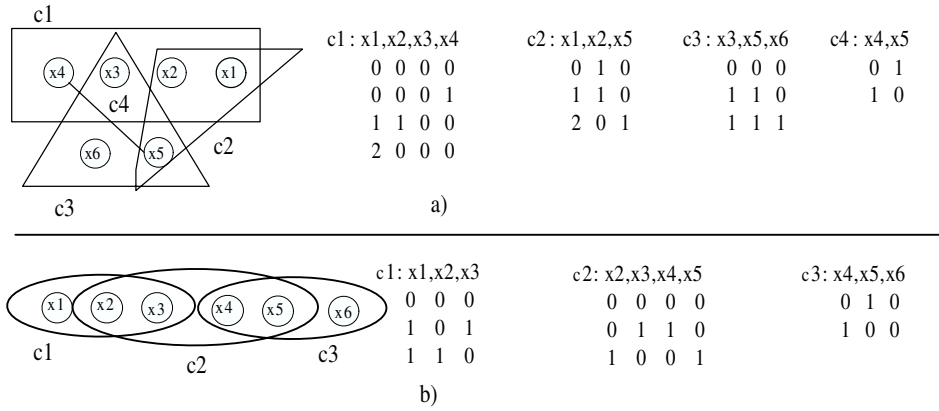
0	1	0
1	0	0

b)

Fig. 1.   Applying domain filtering consistencies on non-binary problems.

4. Theoretical Study

To clarify the definitions of the above domain filtering consistencies, we we first give an example that demonstrates which values are deleted by the application of these consistencies. We then compare the pruning power of the various consistencies. Finally, we consider the special case where the problem consists of binary constraints.

Example 4.1. Figure 1a shows a problem with 6 variables and 4 constraints with the given allowed tuples. All domains are $\{0,1\}$ except $D(x_1)$ which is $\{0,1,2\}$. Assume that we are trying to apply a given domain filtering consistency on variable x_1. All values of x_1 are GAC as they are GAC-supported in both c_1 and c_2. Value 0 is not rPIC (and thus not maxRPWC) as none of its GAC-supports in c_1 can be consistently extended to c_2. Values 1 and 2 are maxRPWC as their GAC supports take the same values in the variables shared by c_1 and c_2. But value 1 is not IωC as its GAC supports in c_1 and c_2 do not satisfy constraint c_4. Value 2 is IωC but it is not EIωC as its GAC-supports satisfy c_4 but do not satisfy constraint c_3. No value of x_1 is rNIC as in this problem where c_1 intersects with all other constraints, rNIC requires that these values participate in a solution.

Now consider the problem depicted in Figure 1b with five 0-1 variables and one variable (x_6) with domain $\{0\}$. Value 0 of x_1 has tuple $(0,0,0)$ as GAC-support in c_1. This tuple can be extended to tuple $(0,0,0,0)$ in c_2 and there are no constraints that intersect with both c_1 and c_2. Therefore $(x_1,0)$ is EIωC. However, the GAC support of 0 cannot be consistently extended to the pair of constraints c_2,c_3 since tuple $(0,0,0,0)$ of c_2 has no PW-support in c_3. Hence, $(x_1,0)$ is not maxR3WC (or r$(1,3)$C).

Theorem 4.1. On problems with non-binary constraints the following relationships hold:

(1) EIωC → IωC → maxRPWC → rPIC → GAC
(2) maxR3WC → maxRPWC and EIω ⊗ maxR3WC ⊗ IωC
(3) r$(1,3)$C is incomparable to maxRPWC, IωC, EIωC, and maxR3WC → r$(1,3)$C → rPIC

(4) NIC is incomparable to rPIC, $r(1,3)$C, maxRPWC, IωC, EIωC, maxR3WC and rNIC
$\rightarrow$ NIC

(5) rNIC $\rightarrow$ EIωC and maxR3WC $\otimes$ rNIC $\otimes$ $r(1,3)$C

Proof. (1) By definition, the "stronger than" relationship holds between EIωC, IωC, maxRPWC, and rPIC. To show EIωC $\rightarrow$ IωC $\rightarrow$ maxRPWC, consider the problems in Example 4.1. The relationship between maxRPC, rPIC and GAC was proved in Ref. 14.

(2) By definition, maxR3WC is stronger than maxRPWC. For strictness consider the problem in Example 4.1b which is maxRPWC but not maxR3WC. To show that maxR3WC is incomparable to EIωC and IωC first consider the same problem which is EIωC (and IωC). Now consider the problem of Figure 2a with 5 0-1 variables. This is maxR3WC but not IωC.

(3) To show that $r(1,3)$C is incomparable to EIωC, IωC and maxRPWC, it suffices to show that $r(1,3)$C can be stronger than EIωC and weaker than maxRPWC. First consider the second problem in Example 4.1. This problem is EIωC but it is not $r(1,3)$C. Now consider the problem in Figure 2b with 6 variables and 4 constraints intersecting on variables x_2 and x_3. Value 0 of x_2 is $r(1,3)$C but it is not maxRPWC. By definition, maxR3WC is stronger than $r(1,3)$C. To show strictness consider the example of Figure 2b where value 0 of x_2 is $r(1,3)$C but it is not maxR3WC. By definition, $r(1,3)$C is stronger than rPIC. To show strictness consider the second problem in Example 4.1. This problem is rPIC but it is not $r(1,3)$C.

(4) To prove that NIC is incomparable to rPIC, $r(1,3)$C, maxRPWC, IωC, EIωC, and maxR3WC it suffices to show that NIC can be weaker than rPIC and stronger than maxR3WC. To show the former, consider a problem with two constraints c_1, c_2, where $var(c_1) = \{x_1, x_2, x_3\}$ and $rel(c_1) = \{(0,0,0), (1,1,0), (0,1,1)\}$, $var(c_2) = \{x_1, x_2, x_4\}$ and $rel(c_2) = \{(0,0,0), (1,1,0), (1,0,1)\}$. This problem is NIC but it is not
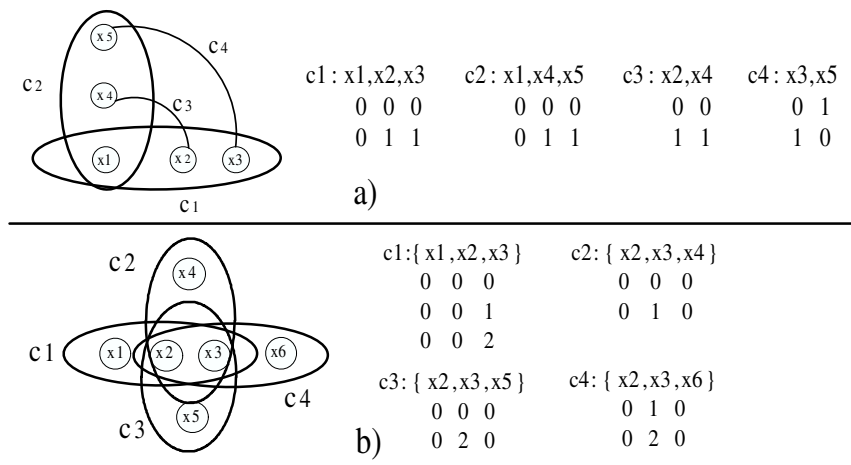

Fig. 2.   A problem that is maxR3WC but not IωC (a). A problem that is $r(1,3)$C but not maxRPWC (b).

rPIC. To show the latter, consider a clique of six variables where all constraints are binary $\neq$ constraints and all domains are $\{0, \ldots, 4\}$. This problem is maxR3WC but not NIC.

To prove that rNIC $\rightarrow$ NIC consider a problem that is rNIC. Any assignment of a variable x_i has a GAC-support τ in each constraint c_j which involves x_i that can be consistently extended to all variables involved in constraints intersecting with c_j. Therefore, τ can be consistently extended to all variables involved in a constraint with x_i, as these constraints intersect (on at least x_i) with c_j. Hence, the problem is NIC. To show strictness, consider the previous example with the two constraints c_1 and c_2. This is NIC but not rNIC.

(5) To prove that rNIC is incomparable to maxR3WC and $r(1,3)$C first consider again the binary problem with a clique of six variables. This is maxR3WC but not rNIC. Now consider the second problem in Example 4.1. This is rNIC but not $r(1,3)$C.

To prove rNIC $\rightarrow$ EIωC consider a problem that is rNIC. Any assignment of a variable x_i has a GAC-support τ in each constraint c_j which involves x_i that can be consistently extended to all variables involved in constraints intersecting with c_j. Therefore, τ can be extended to any constraint c_k intersecting with c_j s.t. all constraints that intersect with both c_j and c_k are satisfied. Hence, the problem is EIωC. To show strictness, consider again the binary problem with a clique of six variables. This is EIωC but not rNIC. $\square$

Figure 3 summarizes the relationships between the various consistencies. For clarity of presentation, the relationships between $r(1,3)$C and NIC, rNIC are not shown.
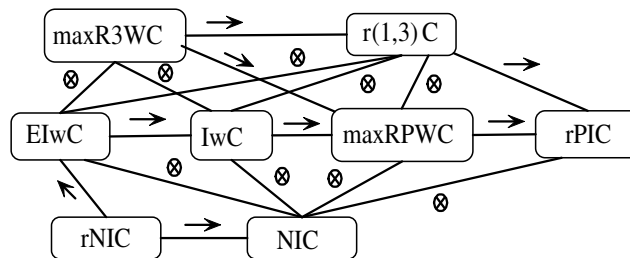

Fig. 3.   Relationships between domain filtering consistencies for non-binary CSPs.

4.1. *Binary constraints*

A natural question is what the aforementioned domain filtering consistencies correspond to in binary CSPs. In Ref. 14 it was shown that rPIC and maxRPWC are equivalent to GAC when all constraints intersect on at most one variable. Since we deal with normalized constraints, as it is usually assumed, then this is the case with binary constraints. Therefore, in normalized binary problems rPIC and maxRPWC reduce to AC. We now show that when restricted to normalized binary constraints, maxR3WC, IωC and EIωC are equivalent to maxRPC while $r(1,3)$C is equivalent to PIC.

Theorem 4.2. On binary CSPs we have maxR3WC $\leftrightarrow$ EIωC $\leftrightarrow$ IωC $\leftrightarrow$ maxRPC and r$(1, 3)$C $\leftrightarrow$ PIC.

Proof. To show IωC $\leftrightarrow$ maxRPC it suffices to show that if a value is deleted by maxRPC then it is also deleted by IωC, and vice versa. Consider a value $a \in D(x_i)$ that is removed by maxRPC. Value a is removed because it is either not AC or because there exists a variable x_j constrained with x_i for which there is no value $b \in D(x_j)$ such that the pair $\langle a, b \rangle$ is path consistent. In the former case, a will be removed by IωC since IωC is stronger than GAC (i.e. AC in binary CSPs). In the latter case, take any AC-support $b \in D(x_j)$ of (x_i, a). Since the pair $\langle a, b \rangle$ is not path consistent there must be a variable x_l such that no value in $D(x_l)$ is compatible with both (x_i, a) and (x_j, b). Assume that c is the constraint between x_i and x_j and c' is the constraint between x_i and x_l. We cannot find AC-supports for a in $D(x_j)$ and $D(x_l)$ so that these supports satisfy the constraints on $var(c) \cup var(c')$, i.e. the constraint between x_j and x_l. Hence, value a is not IωC.

Now consider a value $a \in D(x_i)$ that is deleted by IωC. If a is deleted because it is not AC then maxRPC will obviously delete it. Otherwise, there must be a constraint c involving x_i and a variable x_j such that no AC-support of (x_i, a) in $D(x_j)$ can be consistently extended to any constraint c' that intersects with c so that the constraints on $var(c) \cup var(c')$ are satisfied. Take such a constraint c' and, without loss of generality, assume that $var(c') = \{x_j, x_l\}$. As we only have binary constraints, the only other constraint that can exist among variables $var(c) \cup var(c')$ is the one between x_i and x_l. Value (x_i, a) cannot be be consistently extended to x_j and x_l so that all constraints between the three variables are satisfied. Hence, a is not maxRPC.

We now show that in binary problems EIωC and maxR3WC are equivalent to IωC. Assume that a binary problem is IωC. Then any assignment (x_i, a) can be consistently extended to any constraint c that includes x_i and any other constraint c' that intersects with c so that all constraints between variables $var(c) \cup var(c')$ are satisfied. Since there is no constraint that intersects with both c and c' and includes additional variables (as all constraints are binary), (x_i, a) is also EIωC. Now consider any third constraint c''. If this intersects with both c and c' then, since (x_i, a) is IωC, there exists an AC-support of (x_i, a) in c that can be consistently extended to both c' and c''. If c'' intersects only with one of c, c' (say c') then any valid tuple of c' can be consistently extended to c'' since the problem is IωC, and hence AC. Therefore, in any case, (x_i, a) is maxR3WC.

We now show that r$(1, 3)$C is equivalent to PIC. Consider a value $a \in D(x_i)$ that is removed by PIC. It is removed either because it is not AC or because it cannot be extended to some pair of variables x_j and x_l so that the constraints between all three variables are satisfied. In the former case, a will be removed by r$(1, 3)$C since r$(1, 3)$C is stronger than GAC. In the latter case no AC-support of a in $D(x_j)$ can be consistently extended to a value in $D(x_l)$ so that the constraint between x_i and x_l is satisfied. Hence, value a is not r$(1, 3)$C. Now consider a value $a \in D(x_i)$ that is deleted by r$(1, 3)$C. There must be a constraint c involving x_i and some other variable x_j such that no AC-support of a in $D(x_j)$ can be consistently extended to some pair of constraints c' and c''. There are two cases depending on whether the three constraints form a triangle (i.e. they are the three

constraints involving x_i, x_j and a third variable x_l). If they do not form a triangle then a is removed because it is not AC, in which case PIC will also remove it. If the constraints form a triangle then a cannot be be consistently extended to x_j and a third variable x_l so that all constraints between the three variables are satisfied. Hence, a is not PIC.   □

5. An Algorithm for Domain Filtering Consistencies

A generic AC-7 based algorithm for inverse local consistencies in binary CSPs was proposed in Ref. 16. This algorithm can be relatively easily adapted to apply certain domain filtering consistencies in non-binary problems (e.g. rPIC), but for other consistencies (e.g. maxRPWC) this is much more involved. A generic GAC-3 based algorithm for domain filtering consistencies in non-binary CSPs was given in Refs. 14 and 3. Also, instantiations of this algorithm that can be used to apply maxRPWC, rPIC and RPWC were presented. Here we recall the generic algorithm using a slightly different description (Figure 4) and show how it can be instantiated to apply maxRPWC, IωC, EIωC, and maxR3WC. Similar algorithms can be used to apply rPIC (see Ref. 3) and r(1, 3)C. Algorithms for NIC and rNIC in general require search, as the neighborhood of a variable or a constraint can be very large.

function `DFCons(P,DFC)`
1: put all constraints in Q;
2: **while** Q is not empty
3:     pop constraint c_j from Q;
4:     **for** each variable $x_i \in var(c_j)$
5:         **if** `Revise`$(x_i, c_j, $`DFC`$) > 0$ **then**
6:             **if** $D(x_i)$ is empty **then return** INCONSISTENCY;
8:             `Enqueue`(x_i, c_j);
10: **return** CONSISTENCY;

function `Revise`$(x_i, c_j,$DFC$)$
1: **for** each value $a \in D(x_i)$
2:     PW $\leftarrow$ FALSE;
3:     **for** each valid $\tau(\in rel(c_j)) \geq_l lastGAC_{x_i,a,c_j}$, s.t. $\tau[x_i] = a$
4:         **if** `Seek_Support`$(x_i, c_j, \tau,$DFC$)$ **then**
5:             $lastGAC_{x_i,a,c_j} \leftarrow \tau$;
6:             PW$\leftarrow$ TRUE; **break**;
7:     **if** ¬PW **then** remove a from $D(x_i)$;
8: **return** number of deleted values;

procedure `Enqueue`(x_j, c_i)
1: **for** each c_m such that $x_j \in var(c_m)$
2:     put in Q each c_l $(\neq c_i)$ such that $|var(c_l) \cap var(c_m)| > 1$;
3:     **if** $c_m \neq c_i$ put c_m in Q;

Fig. 4.   A generic algorithm for domain filtering consistencies.

Algorithm DFCons takes as input a (non-binary) CSP P and a specified domain filtering consistency DFC, and enforces DFC on P. DFCons uses a list Q of constraints to propagate value deletions, and works as follows. Initially, all constraints are added to Q. Then constraints are sequentially removed from Q and the domains of the variables involved in these constraints are revised. For each such constraint c_j and variable x_i, the revision is performed using function Revise(x_i, c_j, DFC). If after the revision the domain of x_i becomes empty then the algorithm detects the inconsistency and terminates. Otherwise, if the domain of x_i is pruned then each constraint c_k involving x_i and each constraint intersecting with c_k will be put in Q. Note that in the case of maxRPWC the intersection must be on more than one variable. If Q becomes empty, the algorithm terminates having successfully enforced DFC on P.

In function Revise, for each value a in $D(x_i)$, we first look for a GAC-support in $rel(c_j)$ (line 3). Following GAC2001/3.1,[2] for each constraint c_j and each $a \in D(x_i)$, where $x_i \in var(c_j)$, we keep a pointer $lastGAC_{x_i,a,c_j}$ (initialized to the first tuple in $rel(c_j)$). This is now the most recently discovered tuple in $rel(c_j)$ that GAC-supports (x_i, a) **and**, depending on DFC, has some extra property. For instance, if DFC is maxRPWC (resp. IωC) then $lastGAC_{x_i,a,c_j}$ must have PW-supports (resp. ω-supports) in all constraints that intersect with c_j. If $lastGAC_{x_i,a,c_j}$ is valid then we know that a is GAC-supported. Otherwise, we look for a new GAC-support starting from the tuple immediately after $lastGAC_{x_i,a,c_j}$ in the lexicographic order. If $lastGAC_{x_i,a,c_j}$ is valid or a new GAC-support is found then function Seek_Support is called to check if this GAC-support (tuple τ) satisfies the extra property of DFC.

5.1. *maxRPWC, IωC, EIωC*

The implementation of Seek_Support depends on the consistency being enforced. For maxRPWC (Figure 5), IωC (Figure 6), and EIωC (Figure 7), Seek_Support iterates over each constraint c_k that intersects with c_j.[c] For each such constraint it searches for a tuple τ' that is a PW-support, IωC-support, or extended IωC-support, respectively, of τ. This is explained in more detail below. If such tuples are found for all intersecting constraints then Seek_Support returns TRUE and $lastGAC_{x_i,a,c_j}$ is updated. If no DFC-support τ' is found on some intersecting constraint, indicated by τ' becoming NIL, then Seek_Support returns FALSE and the algorithm looks for a new GAC-support in function Revise. If no GAC-support that satisfies the property of DFC is found, a is removed from $D(x_i)$.

The implementation of line 6 for IωC involves three operations:

- A join of the two tuples τ and τ'.
- A projection of the joined tuple over the variables in $var(c_l)$.
- And a constraint check to determine if the derived tuple satisfies constraint c_l.

---

[c]In the case of maxRPWC only constraints intersecting on more than one variable are considered, since for constraints intersecting on one variable maxRPWC offers no more pruning than GAC.

function `Seek_Support`$(x_i, c_j, \tau, \text{maxRPWC})$
1: **for** each $c_k \in C$ s.t. $|var(c_j) \cap var(c_k)| > 1$
2:    **for** each $\tau'(\in rel(c_k))$
3:       **if** τ' is valid **and** $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$
4:          **then break**;
5:    **if** $\tau' = NIL$ **then return** FALSE;
6: **return** TRUE;

Fig. 5.   Function `Seek_Support` for maxRPWC.

function `Seek_Support`$(x_i, c_j, \tau, \text{I}\omega\text{C})$
1: **for** each $c_k \in C$ s.t. $|var(c_j) \cap var(c_k)| > 0$
2:    **for** each $\tau'(\in rel(c_k))$
3:       **if** τ' is valid **and** $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$
4:          ωC$\leftarrow$ TRUE;
5:          **for** each $c_l \in C$, s.t. $var(c_l) \subseteq var(c_j) \cup var(c_k)$
6:             **if** $(\tau \bowtie \tau')[var(c_l)] \notin rel(c_l)$
7:             **then** ωC$\leftarrow$ FALSE; **break**;
8:          **if** ωC **then break**;
9:    **if** $\tau' = NIL$ **then return** FALSE;
10: **return** TRUE;

Fig. 6.   Function `Seek_Support` for IωC.

function `Seek_Support`$(x_i, c_j, \tau, \text{EI}\omega\text{C})$
1: **for** each $c_k \in C$ s.t. $|var(c_j) \cap var(c_k)| > 0$
2:    **for** each $\tau'(\in rel(c_k))$
3:       **if** τ' is valid **and** $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$
4:          EωC$\leftarrow$ TRUE;
5:          **for** each $c_l \in C$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset \wedge var(c_k) \cap var(c_l) \neq \emptyset$
6:             **if** $\Pi_{var(c_l) \cap (var(c_j) \cup var(c_k))}(\tau \bowtie \tau')$
                 cannot be extended to a valid tuple in $rel(c_l)$
7:             **then** EωC$\leftarrow$ FALSE; **break**;
8:          **if** EωC **then break**;
9:    **if** $\tau' = NIL$ **then return** FALSE;
10: **return** TRUE;

Fig. 7.   Function `Seek_Support` for EIωC.

In contrast, the implementation of line 6 for EIωC is more complex and expensive as it involves searching in $rel(c_l)$. To be precise, the following operations take place:

- A join of the two tuples τ and τ'.
- A projection over the variables in $var(c_j) \cup var(c_k)$ that also appear in $var(c_l)$.
- A search in $rel(c_l)$ to determine if the derived sub-tuple can be extended to a valid tuple.

function Seek_Support(x_i,c_j,τ,maxR3WC)
1: **for** each $c_k \in C$ s.t. $|var(c_j) \cap var(c_k)| > 0$
2:     **for** each valid $\tau'(\in rel(c_k))$
            s.t. $\tau[var(c_j) \cap var(c_k)] = \tau'[var(c_j) \cap var(c_k)]$
3:        3W$\leftarrow$ TRUE;
4:        **for** each $c_l \in C$, s.t. $|var(c_j) \cap var(c_l)| > 0 \vee |var(c_k) \cap var(c_l)| > 0$
5:            **if** $\nexists$ valid $\tau''(\in rel(c_l))$ such that
                 $\tau[var(c_j) \cap var(c_l)] = \tau''[var(c_j) \cap var(c_l)]$ and
                 $\tau'[var(c_k) \cap var(c_l)] = \tau''[var(c_k) \cap var(c_l)]$
6:            **then**
7:                **if** $|var(c_k) \cap var(c_l)| = 0$ **then return** FALSE;
8:                **else** 3W$\leftarrow$ FALSE; **break**;
9:        **if** 3W **then break**;
10:   **if** $\tau' = NIL$ **then return** FALSE;
11: **return** TRUE;

<div align="center">Fig. 8.   Function Seek_Support for maxR3WC.</div>

5.2. *maxR3WC*

In the case of maxR3WC (Figure 8), Seek_Support iterates over each constraint c_k that intersects with c_j and searches for a PW-support of τ in $rel(c_k)$. If such a tuple τ' is found, the algorithm iterates over each constraint c_l that intersects with c_j or c_k (or both) and searches for a tuple $\tau'' \in rel(c_l)$ that is a PW-support of both τ and τ'. This is explained in more detail below. In case c_l does not intersect with c_j (resp. c_k) then obviously any valid $\tau'' \in rel(c_l)$ is a PW-support of τ (resp. τ'). If such a pair of tuples is found for all pairs of constraints c_k and c_l then Seek_Support returns TRUE and $lastGAC_{x_i,a,c_j}$ is updated. Otherwise Seek_Support returns FALSE and a new GAC-support is sought in function Revise.

Depending on how the three constraints intersect, the search for tuple τ'' (line 5) is executed as follows:

- If c_l intersects with c_j but not with c_k then we simply look for a PW-support of τ in $rel(c_l)$ without considering constraint c_k. If no such support is found, Seek_Support returns FALSE (line 7) so that new GAC-support for (x_i, a) in $rel(c_j)$ is sought in Revise. Note that in this case a constraint c_l is considered only if it intersects with c_j on more than one variable since, for constraints intersecting on one variable, the propagation achieved cannot be greater than that achieved by GAC.
- If c_l intersects with c_k but not with c_j then we look for a PW-support of τ'' in $rel(c_l)$ without considering constraint c_j. If no such support is found, then the algorithm immediately (line 8) moves to look for a new PW-support τ' of τ in $rel(c_k)$. As in the previous case, a constraint c_l is considered only if it intersects with c_k on more than one variable.
- Finally, if c_l intersects with both c_j and c_k then we look for a tuple τ'' in $rel(c_l)$ that is a PW-support of both τ and τ'. This is done as in Seek_Support for IωC or EIωC, depending on whether all variables in $var(c_l)$ appear also in $var(c_j) \cup var(c_k)$ or not.

If no such support is found, then the algorithm moves to look for a new PW-support τ' of τ in $rel(c_k)$.

5.3. *Complexities*

We now analyze the worst-case time and space complexity of algorithm `DFCons` when instantiated to apply IωC, EIωC, and maxR3WC. The worst-case complexity of an algorithm for rNIC is exponential in n as any constraint may intersect with all other constraints. In such an extreme case applying rNIC is essentially at least as hard as solving the problem.

First, we recall the complexities of `DFCons` when instantiated to apply maxRPWC or rPIC. The resulting algorithms are called maxRPWC-1 and rPIC-1 in Ref. 3. Following this naming convention, we denote algorithm `DFCons(P,DFC)` as `DFC-1`.

Proposition 5.1.[3] *The worst-case time complexity of algorithms* `maxRPWC-1` *and* `rPIC-1` *is* $O(e^2 k^2 d^p)$, *where p is the maximum number of variables involved in two constraints that share at least two variables.*

As discussed in Ref. 3, the space complexity of `maxRPWC-1` is $O(ekd)$ for extensional constraints and $O(ek^2 d)$ for intensional ones. Accordingly, the space complexity of `rPIC-1` is $O(e^2 kd)$ for extensional constraints and $O(e^2 k^2 d)$ for intensional ones.

Proposition 5.2. *The worst-case time complexity of algorithm* `IωC-1` *is* $O(e^3 k^3 d^p)$.

Proof. Let us denote by k_j the number of variables involved in c_j and by p_{jk} the total number of variables involved in the two constraints c_j and c_k. The complexity is determined by the number of constraint checks performed in total, in all calls to function `Revise` and `Seek_Support`.

We first analyze the cost of `Seek_Support`(x_i,c_j,τ,IωC). The inner loop of `Seek_Support` (lines 5-7) iterates through the, at most $e-2$, constraints c_l, s.t. $var(c_l) \subseteq var(c_j) \cup var(c_k)$. For any such constraint it verifies if the projection over $var(c_l)$ of the join of tuples τ and τ' satisfies constraint c_l. This costs $O(k)$, assuming that the cost of a constraint check is linear to the arity of the constraint. Therefore the cost of the inner loop is $O(ek)$. In the outer loop of `Seek_Support` the algorithm iterates through the constraints that intersect c_j. For each such constraint c_k, the second loop searches for a tuple τ' that is an ω-support of $lastGAC_{x_i,a,c_j}$ (i.e. τ). There are at most $d^{p_{kj}-k_j}$ tuples to be searched, i.e. those that take the same values in variables $var(c_j) \cap var(c_k)$ as in τ. The cost of each such check is $O(k)$. If a tuple τ' that is valid and takes the same values as τ on the intersecting constraints is found then the inner loop is executed to verify if τ' is an ω-support of τ. Therefore, the cost of the second loop is $O(k \times d^{p_{kj}-k_j} \times ek)$. As there are at most $e-1$ constraints intersecting c_j, the cost of `Seek_Support` is bounded above by $K_{ija} = \sum_{c_k \in C \setminus \{c_j\}} ek^2 d^{p_{jk}-k_j}$.

Now let us consider the number of calls to `Seek_Support` in function `Revise`. Given a variable x_i and a constraint c_j, `Seek_Support` is called for each value $a \in D(x_i)$ each time function `Revise`(x_i, c_j, IωC) is called or each time a *new* GAC-support

$lastGAC_{x_i,a,c_j}$ is found (line 3 of $\texttt{Revise}$). $\texttt{Revise}(x_j, c_i, \text{I}\omega\text{C})$ can be called at most nd times. This is because every one of the n variables may either belong to $var(c_j)$ or participate in a constraint that intersects with c_j. In this case every deletion of a value from a variable will force $\texttt{Enqueue}$ to add c_j to Q and subsequently cause a call to $\texttt{Revise}$. $lastGAC_{x_i,a,c_j}$ cannot change more than d^{k_j-1} times because IωC-1 only checks the tuples that contain the assignment (x_i, a) and it only checks tuples that have not been checked before. So, $\texttt{Seek\_Support}$ is called at most $L_{ija} = nd + d^{k_j-1}$ times for each variable x_i, value a, and constraint c_j. L_{ija} is also the number of times a tuple can be checked as GAC-support for (x_i, a) on c_j at a cost $O(k)$ (line 3 of $\texttt{Revise}$). Thus, for a variable x_j, value a, and constraint c_i, the complexity is bounded above by $M_{jia} = L_{ija} \times (k + K_{ija}) = (nd + d^{k_j-1}) \times (k + \sum_{c_k \in C \setminus \{c_j\}} ek^2 d^{p_{jk}-k_j})$. Assuming that $d^{k-1} > nd$, this gives a complexity in $O(e^2 k^2 d^{p-1})$. Since there are at most d values in $D(x_i)$, k variables in $var(c_j)$, and e constraints in C, the total complexity is bounded above by $ekd \times e^2 k^2 d^{p-1}$. This gives a time complexity in $O(e^3 k^3 d^p)$.   □

The space complexity of IωC-1 is determined by the space required for the $lastGAC$ data structure. If the constraints are given in extension, in which case we can use pointers of constant size, then the size of $lastGAC$ is $O(ekd)$. If the constraints are intensionally specified then the size of $lastGAC$ is $O(ek^2d)$, since in this case each pointer is of size k.

Proposition 5.3. *The worst-case time complexity of algorithm* $\texttt{EI}\omega\texttt{C-1}$ *is* $O(e^3 k^3 d^{p+p'})$.

Proof. The proof is similar to that for IωC-1 given above. Note that the two algorithms only differ in the implementation of the inner loop in function $\texttt{Seek\_Support}$. The inner loop of $\texttt{Seek\_Support}$ for $\texttt{EI}\omega\texttt{C-1}$ iterates through the, at most $e - 2$, constraints c_l, s.t. $var(c_j) \cap var(c_l) \neq \emptyset$ and $var(c_k) \cap var(c_l) \neq \emptyset$. Let us denote by p_{ljk} the number of variables involved in c_l but not in c_j or c_k. For each constraint c_l the algorithm searches for a valid tuple in $rel(c_l)$ that takes the same values as τ on variables $var(c_j) \cap var(c_l)$ and the same values as τ' on variables $var(c_k) \cap var(c_l)$. There are at most $d^{p_{ljk}}$ such tuples to be searched and the cost of each check is in $O(k)$. Therefore, the cost of the inner loop is bounded above by $K_{jk} = \sum_{c_l \in C \setminus \{c_j, c_k\}} kd^{p_{ljk}}$. As the rest of algorithm $\texttt{EI}\omega\texttt{C-1}$ is identical to $\texttt{I}\omega\texttt{C-1}$, following the analysis of Proposition 5.2 we can conclude that the the cost of $\texttt{Seek\_Support}$ is bounded above by $K_{ija} = \sum_{c_k \in C \setminus \{c_j\}} (kd^{p_{jk}-k_j} \times K_{jk})$. Therefore, for a variable x_j, value a, and constraint c_i, the complexity is bounded above by $(nd + d^{k_j-1}) \times (k + \sum_{c_k \in C \setminus \{c_j\}} (kd^{p_{jk}-k_j} \times \sum_{c_l \in C \setminus \{c_j, c_k\}} kd^{p_{ljk}}))$. Assuming that $d^{k-1} > nd$, this gives a complexity in $O(d^{k_j-1} \times ekd^{p-k_j} \times ekd^{p'}) = O(e^2 k^2 d^{p+p'-1})$ Therefore, the complexity of $\texttt{EI}\omega\texttt{C-1}$ is in $O(e^3 k^3 d^{p+p'})$.   □

The space complexity of $\texttt{EI}\omega\texttt{C-1}$ is the same as $\texttt{I}\omega\texttt{C-1}$ since they use the same data structures (i.e. $lastGAC$).

Proposition 5.4. *The worst-case time complexity of algorithm* $\texttt{maxR3WC-1}$ *is* $O(e^3 k^3 d^{p+p'})$.

Proof. The proof is similar to that for IωC-1 and EIωC-1. Note that maxR3WC-1 only differs from the other two algorithms in the implementation of the inner loop in function `Seek_Support`. The inner loop of `Seek_Support` for `maxR3WC-1` iterates through the, at most $e - 2$, constraints c_l that intersect with c_j or c_k. For each constraint c_l the algorithm searches for a valid tuple in $rel(c_l)$ that takes the same values as τ on variables $var(c_j) \cap var(c_l)$ and the same values as τ' on variables $var(c_k) \cap var(c_l)$. As discussed, there are three cases depending on whether c_l intersects only with c_j, only with c_k, or with both. The first two are similar in terms of their effect on the cost. Therefore, to simplify the analysis we combine these cases into one and assume that c_l intersects with c_k when intersecting with only one of the two. Let us denote by r_k the number of variables involved in c_k and by p_{kl} the total number of variables involved in the two constraints c_k and c_l. In this case there are at most $d^{p_{kl} - r_k}$ tuples to be searched, i.e. those that take the same values in variables $var(c_k) \cap var(c_l)$ as in τ', with cost $O(k)$ for each one. Now if c_l intersects with both c_j and c_k then let us denote by p_{ljk} the number of variables involved in c_l but not in c_j or c_k. In this case there are at most $d^{p_{ljk}}$ tuples to be searched with cost $O(k)$ for each one. Putting things together, the cost of a single iteration of the inner loop is $O(k \times max(d^{p_{kl} - r_k}, d^{p_{ljk}})) = O(kd^{p_{ljk}})$.[d] Therefore, the cost of the inner loop is bounded above by $K_{jk} = \sum_{c_l \in C \setminus \{c_j, c_k\}} kd^{p_{ljk}}$.

As the rest of algorithm `EI`ω`C-1` is similar to `I`ω`C-1` and `EI`ω`C-1`, following the analysis of Propositions 5.2 and 5.3 we can conclude that the the cost of `Seek_Support` is bounded above by $K_{ija} = \sum_{c_k \in C \setminus \{c_j\}} (kd^{p_{jk} - k_j} \times K_{jk})$. Therefore, for a variable x_j, value a, and constraint c_i, the complexity is bounded above by $(nd + d^{k_j - 1}) \times (k + \sum_{c_k \in C \setminus \{c_j\}} (kd^{p_{jk} - k_j} \times \sum_{c_l \in C \setminus \{c_j, c_k\}} kd^{p_{ljk}})$. Assuming that $d^{k-1} > nd$, this gives a complexity in $O(d^{k_j - 1} \times ekd^{p - k_j} \times ekd^{p'}) = O(e^2 k^2 d^{p + p' - 1})$ Therefore, the complexity of `EI`ω`C-1` is in $O(e^3 k^3 d^{p + p'})$. $\qquad\square$

The space complexity of `maxR3WC-1` is the same as `I`ω`C-1` and `EI`ω`C-1` since they all use the same data structures (i.e. $lastGAC$).

6. Experimental Results

We compared IωC and EIωC to maxRPWC on random problems generated using the extended *model B*.[1] According to this model, a random non-binary CSP is defined by the input parameters $\langle n, d, k, p(e), q \rangle$, where n is the number of variables, d the uniform domain size, k the uniform arity of the constraints, p the density of the problem (i.e. the ratio between the e constraints in the problem and the number of possible constraints involving k variables), and q the uniform looseness of the constraints. The constraints and the allowed tuples were generated following a uniform distribution. We made sure that the generated graphs were connected. All algorithms were implemented in C and the experiments were run on a 3.06 GHz Pentium PC with 1 GB RAM.

---

[d]Asymptotically $d^{p_{kl} - r_k}$ is in $O(d^{p_{ljk}})$ and vice versa.
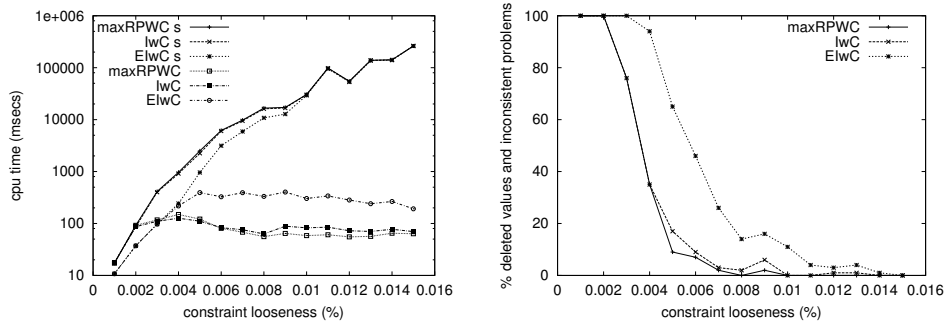
Fig. 9. Cpu times (left) and percentage of inconsistent problems detected (right).

We first compare the pruning power of the three consistencies and their effect as pre-processing tools. Results show that EIωC and, on denser problems, IωC can achieve considerably more pruning than maxRPWC and thus are useful for preprocessing. Then we compare algorithms that maintain the consistencies throughout search. Results show that IωC, and especially EIωC, can be too expensive to maintain on soluble instances, but they can offer speed-ups on insoluble instances.

Figure 9 (left) shows average CPU times for the three consistencies on 100 instances of class $\langle 30, 20, 4, 0.001(27), q \rangle$. We show both the time needed to enforce the consistencies and the time required to solve the instances with an algorithm that maintains maxRPC during search after they have been preprocessed by each of the three consistencies (suffix s). The right figure shows the average percentage of instances proved to be inconsistent by the three consistencies. The value of q is varied along the x-axis.

IωC displays similar performance to maxRPWC in cpu times and inconsistency detection. This is not surprising given that this is a sparse class where all constraints are 4-ary. As a result, for any pair of intersecting constraints c_j, c_k there is seldom the case that some other constraint exists which only involves variables from $var(c_j) \cup var(c_k)$. Note that EIωC detects many more inconsistent problems, and deletes a higher percentage of values, (for $q > 0.004$) than IωC and maxRPWC, albeit with a higher cost. However, this preprocessing cost is negligible compared to the cost of search, and as a result, the search algorithm that uses EIωC preprocessing is more efficient than the others up to the value of q where EIωC achieves a notable number of value deletions.

Table 1 gives results from problems belonging to classes $\langle 50, 10, 4, 0.001(230), q \rangle$ (class 1) and $\langle 100, 10, 4, 0.0001(392), q \rangle$ (class 2). In each line we give the number of inconsistent instances detected, the average percentage of value deletions, and the cpu time (in msecs) when each consistency is enforced for preprocessing. The first three lines in the table refer to class 1 and correspond to parameter settings such that maxRPWC determines as inconsistent almost all, around half and only a few of the instances. Accordingly for class 2 in the next three lines. EIωC proves the inconsistency of all instances and in some cases it runs up to one order of magnitude faster than the other consistencies as it

Table 1.   Average results over 100 instances on two classes of random problems.

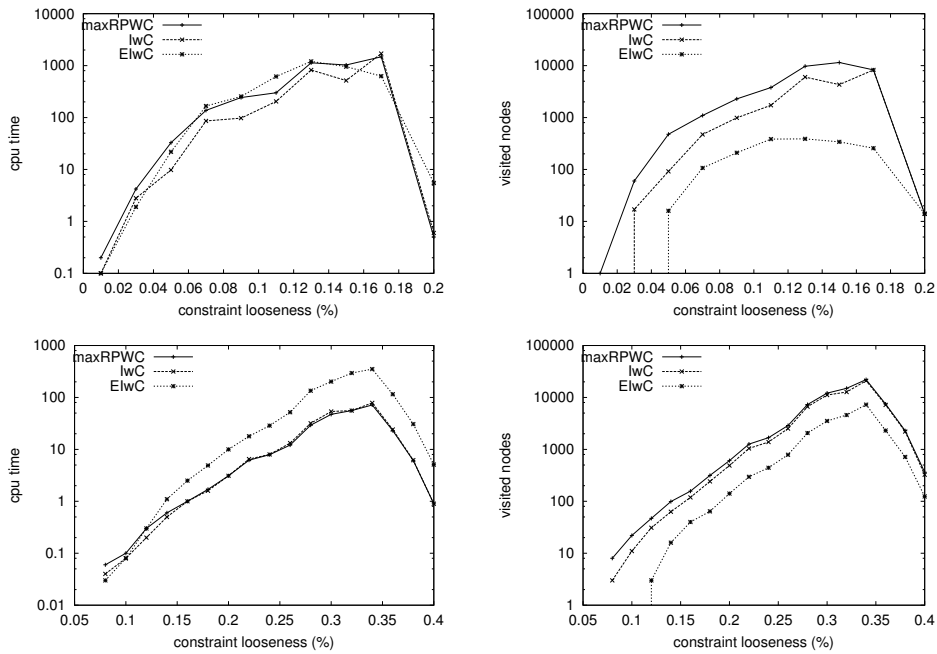| class	maxRPWC			IωC			EIωC		
	inc	%del	time	inc	%del	time	inc	%del	time
1	96	28	583	99	26	275	100	7	48
1	45	15	561	90	27	441	100	10	90
1	8	3	295	53	17	470	100	13	231
2	95	24	888	95	23	813	100	10	70
2	48	14	1488	54	16	1251	100	13	302
2	9	3	412	18	5	535	100	15	674

Fig. 10.   Cpu times and node visits for classes $\langle 15, 20, 4, 0.02(27), q \rangle$ (top) and $\langle 20, 8, 4, 0.008(38), q \rangle$ (bottom).

quickly wipes out some domain. IωC proves the inconsistency of many more instances than maxRPWC (especially in class 1) in competitive run times.

Figure 10 gives results from problems belonging to classes $\langle 15, 20, 4, 0.02(27), q \rangle$ and $\langle 20, 8, 4, 0.008(38), q \rangle$. For each data point we generated 50 instances and measured the average node visits and cpu time of algorithms that maintain maxRPWC, IωC and EIωC throughout search. These algorithms are simply denoted by the local consistency they apply. Results show that in both classes, and especially the first one, EIωC significantly reduces the size of the explored search tree (i.e. node visits) but at a high cost. IωC outperforms maxRPWC on the first class while it is competitive but not faster on the second

class. Both of the strong consistencies are more efficient on insoluble instances compared to soluble ones.

In general, these preliminary experiments indicate that IωC is better suited to denser problems with large domains where there are many intersections between constraints. On such problems it can outperform maxRPWC as it detects more inconsistencies with little extra cost. EIωC cannot be maintained throughout search in practice, but it can be used for preprocessing and perhaps it can be conservatively applied during search (e.g. on specific constraints). Of course, further experimentation is required to validate or refute these conjectures.

7. Conclusion

Although domain filtering local consistencies tend to be more practical than consistencies that change the constraint relations and the constraint graph, only few such consistencies have been proposed for non-binary constraints. In this paper, we performed a detailed study of several strong domain filtering consistencies for non-binary constraints. All these consistencies are stronger than GAC, the consistency that is predominantly used by current constraint solvers, and most are stronger than maxRPWC, a recently introduced domain filtering consistency for non-binary constraints. We proved that three of the new consistencies are equivalent to maxRPC when restricted to normalized binary CSPs while another is equivalent to PIC. We also described a generic algorithm for domain filtering consistencies in non-binary CSPs, showed how this algorithm can be instantiated to enforce some of the proposed consistencies, and analyzed the worst-case complexities of the resulting algorithms.

References

1. C. Bessière, P. Meseguer, E. Freuder, and J. Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. *Artificial Intelligence*, 141:205–224, 2002.
2. C. Bessière, J. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
3. C. Bessière, K. Stergiou, and T. Walsh. Inverse Consistencies for Non-binary Constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.
4. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings of CP-97*, pages 312–326, 1997.
5. R. Debruyne and C. Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
6. R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997.
7. E. Freuder. A Sufficient Condition for Backtrack-bounded Search. *JACM*, 32(4):755–761, 1985.
8. E. Freuder and C. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI'96*, pages 202–208, 1996.
9. M. Gyssens. On the complexity of join dependencies. *ACM Trans. Database Syst.*, 11(1):81–108, 1986.
10. P. Janssen, P. Jègou, B. Nouguier, and M. Vilarem. A filtering process for general constraint satisfaction problems: Achieving pairwise consistency using an associated binary representation. In *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

11. P. Jégou. *Contribution à l'étude des problèmes de satisfaction de contraintes: algorithmes de propagation et de résolution; propagation de contraintes dans les réseaux dynamiques*. PhD thesis, CRIM, University Montpellier II, 1991. in French.
12. P. Jègou. On the Consistency of General Constraint Satisfaction Problems. In *Proceedings of AAAI'93*, pages 114–119, 1993.
13. S. Nagarajan, S. Goodwin, and A. Sattar. Extending Dual Arc Consistency. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(5):781–815, 2003.
14. K. Stergiou and T. Walsh. Inverse Consistencies for Non-binary Constraints. In *Proceedings of ECAI-2006*, pages 153–157, 2006.
15. P. van Beek and R. Dechter. On the Minimality and Global Consistency of Row-convex Constraint Networks. *JACM*, 42(3):543–561, 1995.
16. G. Verfaillie, D. Martinez, and C. Bessière. A Generic Customizable Framework for Inverse Local Consistency. In *Proceedings of AAAI'99*, pages 169–174, 1999.