Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 172 (2008) 800–822
Artificial Intelligence
www.elsevier.com/locate/artint

# Domain filtering consistencies for non-binary constraints

Christian Bessiere [a,*], Kostas Stergiou [b], Toby Walsh [c]

[a] LIRMM (CNRS/U. Montpellier), France
[b] Department of Information & Communication Systems Engineering, University of the Aegean, Greece
[c] National ICT Australia & School of Computer Science and Engineering, University of New South Wales, Australia

Received 25 February 2007; received in revised form 5 October 2007; accepted 22 October 2007

**Abstract**

In non-binary constraint satisfaction problems, the study of local consistencies that only prune values from domains has so far been largely limited to generalized arc consistency or weaker local consistency properties. This is in contrast with binary constraints where numerous such domain filtering consistencies have been proposed. In this paper we present a detailed theoretical, algorithmic and empirical study of domain filtering consistencies for non-binary problems. We study three domain filtering consistencies that are inspired by corresponding variable based domain filtering consistencies for binary problems. These consistencies are stronger than generalized arc consistency, but weaker than pairwise consistency, which is a strong consistency that removes tuples from constraint relations. Among other theoretical results, and contrary to expectations, we prove that these new consistencies do not reduce to the variable based definitions of their counterparts on binary constraints. We propose a number of algorithms to achieve the three consistencies. One of these algorithms has a time complexity comparable to that for generalized arc consistency despite performing more pruning. Experiments demonstrate that our new consistencies are promising as they can be more efficient than generalized arc consistency on certain non-binary problems.


*Keywords:* Constraint programming; Constraint satisfaction problems; Non-binary constraints; Local consistency; Inverse consistency; Pairwise consistency

## 1. Introduction

Local consistency techniques are of great importance in constraint programming. They prune values from the domain of variables and terminate branches of the search tree, saving much fruitless exploration of the search tree. The most widely studied local consistency is (generalized) arc consistency (GAC). Local consistency techniques that only filter domains like GAC tend to be more practical than those that alter the structure of the constraint hypergraph or the constraints' relations (e.g. path consistency). For binary constraints, many domain filtering consistencies have been proposed and evaluated, including inverse and singleton consistencies [10,13,23]. The situation is very different in the case of non-binary constraints. A number of consistencies that are stronger than GAC have been developed, including relational consistency [22], pairwise consistency [15], and hyper-$m$-consistency [17]. However, these are typically not domain filtering. In addition, algorithms that enforce them typically have a high time complexity.

* Corresponding author.
*E-mail address:* bessiere@lirmm.fr (C. Bessiere).

0004-3702/$ – see front matter 
doi:10.1016/j.artint.2007.10.016

In this paper we study a number of domain filtering consistencies for non-binary constraints, inspired by corresponding variable based consistencies for binary problems. A domain filtering consistency is a local consistency that when enforced removes from the domain of a variable the values that cannot be consistently extended to some additional variables. The simplest level of domain filtering local consistency for binary constraints is arc consistency, which when enforced, removes values that cannot be consistently extended to any other variable. As another example, enforcing path inverse consistency removes values that cannot be consistently extended to any set of two other variables. Enforcing a domain filtering local consistency never infers no-goods of size more than one, and therefore it does not create new constraints or modify existing constraint definitions. On the opposite, local consistencies that do not only filter domains may infer and add no-goods of arbitrary size to the problem. This implies that either new extensional constraints may be created or the definition of existing intentional constraints may be altered to cover the recorded no-goods. In both cases the space requirements can become excessive. Also, restoration of the problem state during backtracking becomes much more involved. For these reasons domain filtering consistencies are generally considered more practical.

We study the following domain filtering consistencies: restricted pairwise consistency (RPWC), relational path inverse consistency (rPIC), and max restricted pairwise consistency (maxRPWC). All these consistencies are stronger than GAC, in the sense that they perform more domain pruning. However, they are weaker than a local consistency which applies pairwise consistency and then removes any values that are left unsupported in some constraint. Pairwise consistency (also called inter-consistency [17]) is a strong local consistency defined between constraint relations [15] whose application removes tuples from the constraints' relations.

Relational consistencies treat non-binary constraints irrespective of arity in a uniform manner [22]. However, they have rarely been used in practice as most are not domain filtering, and as most have high time complexities. As with relational consistencies, pairwise consistency, and hyper-$m$-consistency in general, has also rarely been used as it is not domain filtering and the algorithm proposed in [15] has a high time complexity and requires all constraints to be extensionally represented. The consistencies we study do not suffer from these problems. They are domain filtering and are not prohibitively expensive to enforce.

In our theoretical study of RPWC, rPIC, and maxRPWC, we compare their pruning power and also compare them to other consistencies for non-binary problems. We also consider their pruning power in the case where we only have binary constraints. Our theoretical analysis reveals some surprises. For example, although the consistencies we study are inspired by corresponding consistencies for binary problems, when restricted to binary constraints they do not reduce to the variable based definition of their counterparts.

We also propose algorithms to enforce each of the consistencies that can be applied to constraints intentionally or extensionally specified. The time complexity of the algorithms for maxRPWC and rPIC is $O(e^2k^2d^p)$, where $e$ is the number of constraints, $k$ is the maximum arity, $d$ is the maximum domain size, and $p$ is the maximum number of variables involved in two constraints that share at least two variables. The time complexity of the algorithm for RPWC is $O(ne^2k^2d^k)$, where $n$ is the number of variables. Focusing on maxRPWC, which is the most efficient among the three consistencies, we propose two alternative algorithms to apply it. One of these algorithms avoids many constraint checks and its time complexity is $O(e^2kd^k)$. This is comparable to the complexity of standard GAC algorithms, like GAC-Schema [6] and GAC2001/3.1 [8]. The time complexity of GAC2001/3.1 is $O(ek^2d^k)$, while GAC-Schema, taking advantage of multidirectionality, has time complexity of $O(ekd^k)$. However, the improvement we achieve comes at a cost as the space required is exponential in the number of *shared* variables. Our third algorithm for maxRPWC provides a balance between the other two by avoiding many constraint checks that the first one makes, but not as many as the second algorithm, while only requiring polynomial space.

Experimental results demonstrate that it is feasible to maintain strong domain filtering consistencies during search, and that they can be more cost-effective than GAC on certain problems. We show that maxRPWC, which is the strongest among the consistencies we study, is also the most promising. This consistency prunes more values than RPWC and rPIC, with small additional cost.

The rest of the paper is structured as follows. Section 2 gives necessary background and definitions. In Section 3 we define RPWC, rPIC, and maxRPWC. In Section 4 we make a theoretical study of the pruning power offered by the three consistencies. Section 5 presents algorithms for applying the consistencies. In Section 6 we give experimental results. Finally, we conclude and point to future work.

## 2. Background

In this section we first give some necessary definitions on CSPs and then review local consistencies for binary and non-binary constraints.

### 2.1. Basic definitions

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple $(X, D, C)$ where: $X = \{x_1, \ldots, x_n\}$ is a set of $n$ variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of ordered finite domains, and $C = \{c_1, \ldots, c_e\}$ is a set of $e$ constraints. Each constraint $c_i$ is a pair $(var(c_i), rel(c_i))$, where $var(c_i) = (x_{i_1}, \ldots, x_{i_k})$ is an ordered subset of $X$, and $rel(c_i)$ contains 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 $(a_{i_1}, \ldots, a_{i_k})$. A tuple $\tau \in rel(c_i)$ is *valid* iff all the values in the tuple are present (i.e. they have not been removed) in the domain of the corresponding variable. The process which verifies whether a given tuple is valid and allowed by a constraint $c$ is called a *constraint check*.

A constraint $c$ can be either defined *extensionally* by explicitly giving $rel(c)$, or (usually) *intensionally* by implicitly specifying $rel(c)$ through a predicate or arithmetic function. Any two constraints $c_i$ and $c_j$ *intersect* iff the set $var(c_i) \cap var(c_j)$ of variables involved in both constraints is not empty. We denote by $p$ the maximum number of variables involved in two constraints that intersect on at least two variables. The maximum number of variables that any two constraints intersect on is denoted by $f$.

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.

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 ordered set of variables over which a tuple $\tau$ is defined is $var(\tau)$. For any subset $var'$ of $var(\tau)$, $\tau[var']$ is the sub-tuple of $\tau$ that includes only assignments to the variables in $var'$. Any two tuples $\tau$ and $\tau'$ over $var(c_i) = (x_{i_1}, \ldots, x_{i_k})$ can be lexicographically ordered. In this ordering, $\tau <_l \tau'$ iff there a exists a subsequence $(x_{i_1}, \ldots, x_{i_j})$ of $var(c_i)$ such that $\tau[x_{i_1}, \ldots, x_{i_j}] = \tau'[x_{i_1}, \ldots, x_{i_j}]$ and $\tau[x_{i_{j+1}}] <_l \tau'[x_{i_{j+1}}]$. An assignment $\tau$ is *consistent* iff 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 is a consistent assignment to all variables.

### 2.2. Local consistencies

The concept of local consistency is central to constraint programming. Local consistencies are used prior to and during search to filter domains and discover inconsistencies early. The most widely studied local consistency is of course (G)AC. In the rest of the paper we usually write AC when it involves binary constraints and GAC otherwise. A CSP is *Generalized Arc Consistent* (GAC) iff for all $x_i \in X$, $D(x_i)$ is non-empty and for all $a \in D(x_i)$, $a$ is *GAC-supported* in each constraint $c_j$, s.t. $x_i \in var(c_j)$. A value $a \in D(x_i)$ is GAC-supported in a constraint $c_j$ iff there exists $\tau \in rel(c_j)$ such that $\tau[x_i] = a$ and $\tau$ is valid. In this case, we say that $\tau$ is a GAC-support of $a$ in $c_j$.

#### 2.2.1. Local consistencies for binary constraints

Apart from AC, numerous other local consistencies have been proposed for binary CSPs. A generic concept that captures many of them is $(i, j)$-consistency [12]. A binary CSP is $(i, j)$-*consistent* iff it has non-empty domains and any consistent assignment of $i$ variables can be extended to a consistent assignment involving $j$ additional variables. A problem is *strong* $(i, j)$-*consistent* iff it is $(k, j)$ consistent for all $k \leqslant i$. Following the definition of $(i, j)$-consistency, a problem is (strong) *path consistent* (PC) iff it is (strong) (2, 1)-consistent. A problem is (*strong*) $m$-*consistent* iff it is (strong) $(m - 1, 1)$-consistent.

Local consistencies that only prune values from domains and leave the structure of the constraint graph/hypergraph unchanged are called *domain filtering* consistencies [10]. We now recall the most important domain filtering consistencies for binary CSPs proposed in the literature. They were usually defined on *normalized* binary CSPs, that is, CSPs where any pair of variables is linked by at most one constraint [1,4]. We present their generalizations where several constraints are allowed on the same pair of variables.

A binary CSP is *Path Inverse Consistent* (PIC) [13] iff it has non-empty domains and it is $(1,2)$-consistent i.e., for all $x_i \in X$, for all $a \in D(x_i)$, for all $x_j, x_l \in X$, s.t. $x_i \neq x_j \neq x_l \neq x_i$, there exists $b \in D(x_j)$, there exists $d \in D(x_l)$, s.t. the assignments $(x_i, a)$, $(x_j, b)$ and $(x_l, d)$ satisfy the constraints between the three variables. In general, a problem is *inverse m-consistent* iff it is $(1, m)$ consistent.

A binary CSP is *Restricted Path Consistent* (RPC) [3] iff it is $(1,1)$-consistent and for all $x_i \in X$, for all $a \in D(x_i)$, for all $x_j \in X$ s.t. there is a unique value $b$ in $D(x_j)$ with $((x_i, a), (x_j, b))$ consistent, for all $x_l \in X$, there exists $d \in D(x_l)$ s.t. the 3-tuple $((x_i, a), (x_j, b), (x_l, d))$ is consistent. Informally, a problem is RPC iff it is $(1,1)$-consistent and for all values $a$ that have a single consistent extension $b$ to some variable, this pair of values is path consistent.

A binary CSP is *max Restricted Path Consistent* (maxRPC) [9] iff it is $(1,1)$-consistent and for all $x_i \in X$, for all $a \in D(x_i)$, for all $x_j \in X$ s.t. there exists $c \in C$ with $var(c) = (x_i, x_j)$, there exists $b$ in $D(x_j)$, s.t. for all $x_l \in X$, there exists $d \in D(x_l)$ s.t. the 3-tuple $((x_i, a), (x_j, b), (x_l, d))$ is consistent. Informally, a problem is maxRPC iff it is $(1,1)$-consistent and for each value $(x_i, a)$ and variable $x_j$ linked to $x_i$ by some constraint, there is a consistent extension $b$ of $a$ on $x_j$ and this pair of values is path consistent.

A binary CSP is *Singleton Arc Consistent* (SAC) [10] iff it has non-empty domains and for any assignment $(x_i, a)$ of a variable $x_i \in X$, the resulting subproblem, denoted by $P_{(x_i,a)}$, can be made AC. If $P_{(x_i,a)}$ cannot be made AC, SAC removes $a$ from $D(x_i)$.

Finally a CSP is *Neighborhood Inverse Consistent* (NIC) iff any consistent assignment $(x_i, a)$ of a variable $x_i \in X$ can be extended to a consistent assignment of all the variables in $x_i$'s neighborhood [13]. The neighborhood of a variable consists of all variables that are constrained with it.

#### 2.2.2. Local consistencies for non-binary constraints

A number of consistencies that are stronger than GAC have been developed for non-binary problems, including relational consistency [22], pairwise consistency [15], and hyper-$m$-consistency [17]. However, these are not domain filtering as they may alter some constraints' relations or add new constraints to the problem.

To define local consistency properties on non-binary constraints in an uniform manner, Dechter and van Beek introduced *relational consistency* [11]. This unifies together operations like resolution in theorem proving, joins in relational databases, and variable elimination when solving linear inequalities. By abstracting out the constraint arity, relational consistencies offer an elegant characterization of the level of local consistency needed to ensure global consistency given constraints of a given tightness, and of that guaranteed to hold given constraints of a certain looseness [11].

A problem is *relationally arc consistent* (rel AC) iff any consistent assignment for all but one of the variables in a constraint can be extended to the final variable so as to satisfy the constraint [22]. A problem is *relationally path-consistent* (rel PC) iff any consistent assignment 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.

Relational consistency can be generalized to subsets of constraints of arbitrary size. A problem is *relationally $m$-consistent* iff any consistent assignment 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 assignment 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 $j \leqslant i$.

We can construct singleton versions of all the consistencies for non-binary constraints in a straightforward manner [10]. For example, a problem is *singleton generalized arc consistent* (SGAC) iff it has non-empty domains and for any assignment of a variable, the resulting subproblem can be made GAC.

Other consistencies for non-binary constraints include pairwise consistency and hyper-$m$-consistency. A problem is *pairwise consistent* (PWC) iff it has non-empty relations and any consistent tuple of a constraint $c$ can be consistently extended to any other constraint that intersects with $c$ [15]. As shown in [15], applying PWC to a non-binary CSP is equivalent to applying AC to the dual encoding of the problem. Since PWC does not prune values from domains but instead deletes tuples from constraint relations, domains can be filtered if GAC is applied as a second step [15]. In this way, values that have lost all their GAC-supports in a constraint will be deleted. In the rest of this paper the local consistency achieved by this two-step process is simply called PWC + GAC.

PWC has been generalized to *$k$-wise consistency* [14,16] and *hyper-$m$-consistency* [17]. 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 $m$th constraint. As noted in [17], hyper-$m$-consistency on a non-binary problem is equivalent to $m$-consistency on the dual encoding of the problem.

To compare the pruning power of the various consistencies, we follow [10] and call a consistency property $A$ stronger than $B$ iff in any problem in which $A$ holds then $B$ holds, and strictly stronger (written $A \to B$) 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.

From [10] we know that RPC is strictly stronger than AC. Also, in problems with more than two variables maxRPC is strictly stronger than PIC which is strictly stronger than RPC.

## 3. Domain filtering consistencies for non-binary constraints

Many strong local consistency techniques have prohibitive space and time complexities. One way around this problem is to use domain filtering consistencies since they require limited space as they only prune domains. We will define three domain filtering consistencies for non-binary constraints inspired by the definitions of RPC, PIC and maxRPC for binary constraints. RPC, PIC and maxRPC specify that every value in the domain of a variable must allow a consistent extension on every second variable, and they also specify conditions on how this consistent pair of values can be extended to a third variable. Our generalizations to the non-binary case specify that every value in the domain of a variable must allow a GAC-support on every constraint, and they also specify conditions on how this GAC-support can be extended to another constraint. For example, when enforcing PIC we remove values that cannot be consistently extended to any set of two other variables. When enforcing rPIC (the generalization of PIC) we will remove values that cannot be extended to satisfy any set of two constraints.

*Relational path inverse consistency*

By analogy to the definition of PIC, and inverse $m$-consistency, relational $(1,2)$-consistency is called relational path inverse consistency. We now give a formal definition.

**Definition 1.** A non-binary CSP is *relational Path Inverse Consistent* (rPIC) iff $\forall x_i \in X$ and $\forall a \in D(x_i)$, $\forall c_j \in C$, where $x_i \in var(c_j)$, and $\forall c_l \in C$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset$, $\exists \tau \in rel(c_j)$ such that $\tau[x_i] = a$, $\tau$ is valid, **and** $\exists \tau' \in rel(c_l)$ such that $\tau'$ is valid and $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$.

If rPIC is applied it will enforce GAC. This is because in the above definition constraints $c_j$ and $c_l$ are not necessarily different. In addition, rPIC will remove any value $a \in D(x_i)$ such that for some constraint $c_j$ where $x_i$ participates, no GAC-support of $a$ can be extended to a valid tuple in some other constraint that intersects with $c_j$.

*Restricted pairwise consistency*

**Definition 2.** A non-binary CSP is *Restricted Pairwise Consistent* (RPWC) iff $\forall x_i \in X$, all values in $D(x_i)$ are GAC and, $\forall a \in D(x_i)$, $\forall c_j \in C$, s.t. there exists a unique valid $\tau \in rel(c_j)$ with $\tau[x_i] = a$, $\forall c_l \in C$ $(c_l \neq c_j)$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset$, $\exists \tau' \in rel(c_l)$, s.t. $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$ and $\tau'$ is valid.

As shown by the definition, RPWC is inspired by the variable-based consistency RPC. If RPWC is applied it will enforce GAC. In addition it will remove any value $a \in D(x_i)$ such that for some constraint $c_j$ where $x_i$ participates, $a$ has a single GAC-support in $rel(c_j)$ and this tuple cannot be extended to a valid tuple in some other constraint that intersects with $c_j$.

*Max restricted pairwise consistency*

**Definition 3.** A non-binary CSP is *max Restricted Pairwise Consistent* (maxRPWC) iff $\forall x_i \in X$ and $\forall a \in D(x_i)$, $\forall c_j \in C$, where $x_i \in var(c_j)$, $\exists \tau \in rel(c_j)$ such that $\tau[x_i] = a$, $\tau$ is valid, **and** $\forall c_l \in C$ $(c_l \neq c_j)$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset$, $\exists \tau' \in rel(c_l)$, s.t. $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$ and $\tau'$ is valid. In this case we say that $\tau'$ is a PW-support of $\tau$.

As shown by the definition, maxRPWC is inspired by the variable-based consistency maxRPC. If maxRPWC is applied it will enforce GAC. In addition 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 $a$ can be extended to a valid tuple in every constraint that intersects with $c_j$.

## 4. Theoretical results

We first compare the pruning power of RPWC, rPIC maxRPWC and GAC on general non-binary problems and we position the consistencies with respect to PWC+GAC and SGAC. We also consider the special case where all constraints intersect on at most one variable. We then consider the pruning power of the consistencies in problems consisting of binary constraints only. Finally, we study RPWC, rPIC and maxRPWC with respect to consistencies enforced in two well-known binary encodings of non-binary problems.

### *4.1. Non-binary constraints*

**Theorem 1.**

$$\begin{array}{ccccccccc} \text{PWC} + \text{GAC} & \to & \max \text{RPWC} & \to & \text{rPIC} & \to & \text{RPWC} & \to & \text{GAC} \\ \otimes & & \otimes & & \otimes & & \uparrow & & \\ \text{SGAC} & & \text{SGAC} & & \text{SGAC} & & \text{SGAC} & & \end{array}$$

**Proof.** By definition, PWC + GAC is stronger than maxRPWC, maxRPWC is stronger than rPIC, rPIC is stronger than RPWC, and RPWC is stronger than GAC.

To show PWC + GAC $\to$ maxRPWC, consider the problem depicted in Fig. 1a with five 0–1 variables $\{x_1, \dots, x_5\}$ and one variable ($x_6$) with domain $\{0\}$. There are three constraints, and their allowed tuples are shown in Fig. 1a. 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 therefore $(x_1, 0)$ is maxRPWC (as are all other values). However, tuple $(0, 0, 0, 0)$ of $c_2$ cannot be consistently extended to $c_3$, and therefore it is not pairwise consistent, which means that PWC will delete it. As a result, tuple $(0, 0, 0)$ in $c_1$ will be deleted and a further application of GAC will remove value 0 from $D(x_1)$.

To show maxRPWC $\to$ rPIC, consider the problem depicted in Fig. 1b with three 0–1 variables $\{x_1, x_2, x_3\}$ and two variables ($x_4$ and $x_5$) with domain $\{0\}$. There are three constraints, and their allowed tuples are shown in Fig. 1b. Value 0 of $x_1$ is rPIC as its GAC-support $(0, 0, 0)$ in $c_1$ can be extended to tuple $(0, 0, 0)$ in $c_2$ and its GAC-support $(0, 1, 1)$ in $c_1$ can be extended to tuple $(1, 1, 0)$ in $c_3$. However, this value is not maxRPWC as it does not have a GAC-support in $c_1$ that can be extended to both $c_2$ and $c_3$.

To show rPIC $\to$ RPWC, consider the problem on variables $x_1$ to $x_3$ with domain $\{0, 1, 2\}$, $x_4$ with domain $\{0, 1\}$ and two constraints $c_1 = \mathit{alldiff}(x_1, x_2, x_3)$ and $c_2 = \mathit{alldiff}(x_2, x_3, x_4)$. No value admits a single GAC-support on any of the two constraints. So, the problem is RPWC. However, $(x_1, 2)$ is not rPIC because none of its GAC-supports on $c_1$ extends consistently to $c_2$.

To show RPWC $\to$ GAC, consider the problem with constraints $\mathit{alldiff}(x_1, x_2, x_3)$ and $x_1 = x_2$. If the domains are $\{0, 1, 2\}$ then the problem is GAC but it is not RPWC.

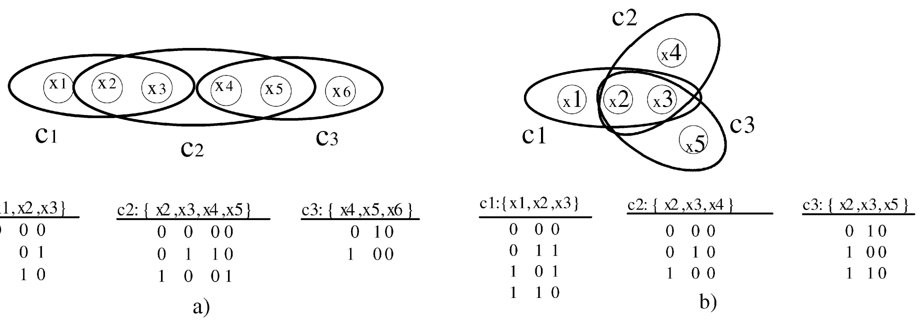


Fig. 1. a) PWC + GAC vs. maxRPWC, b) maxRPWC vs. rPIC.

We show SGAC $\to$ RPWC. To show that SGAC is stronger than RPWC consider a problem with a value $a$ for some $x_i$ which is not RPWC. Then, there exists a constraint $c_j$ such that the only GAC-support $\tau$ of $(x_i, a)$ cannot be extended to some constraint $c_k$. Thus, when $x_i$ is assigned $a$ in the problem, GAC will assign all values in $\tau$ because it is the only GAC-support of $(x_i, a)$ on $c_j$. So, GAC on $c_k$ will wipe out a domain and the problem is not SGAC. To show strictness, consider the problem $P$ with variables $x_1, x_2, x_3$ taking values in $\{0, 1\}$ and constraints $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_1$. This problem is RPWC because any support for any value can be extended to any second constraint. However, it is not SGAC because GAC fails on $P_{(x_1,0)}$.

We show that SGAC is incomparable with PWC + GAC, maxRPWC, and rPIC. Consider the problem $P'$ on variables $x_1$ to $x_4$ with domain $\{0, 1\}$ and the constraints $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_4$, $x_4 = x_1$. It is PWC + GAC, and so it is maxRPWC and rPIC. But it is not SGAC, which in this case is equivalent to SAC (see [10] Fig. 3e). Consider the problem $P''$ on variables $x_1$ to $x_3$ with domain $\{0, 1\}$ and the constraints $c_1(x_1, x_2, x_3) = \{000, 011, 100, 111\}$ and $c_2(x_1, x_2, x_3) = \{001, 010, 100, 111\}$. $(x_1, 0)$ is SGAC because $P''_{(x_1,0)}$ is GAC. All other values are SGAC because they belong to a solution of $P''$. But none of the GAC-supports of $(x_1, 0)$ on $c_1$ can satisfy $c_2$. Thus, $P''$ is not rPIC, and so not maxRPWC and not PWC + GAC. □

*Constraints intersecting on at most one variable*

Not surprisingly, when all constraints intersect on at most one variable, maxRPWC, rPIC, and RPWC collapse down to GAC.

**Theorem 2.** *On constraints that intersect on at most one variable*: $\max \text{RPWC} \leftrightarrow \text{rPIC} \leftrightarrow \text{RPWC} \leftrightarrow \text{GAC}$.

**Proof.** Suppose that the constraints intersect on at most one variable and are GAC. Consider an assignment $x_i = a$ and a constraint $c_j$ involving $x_i$. As $c_j$ is GAC, we can find a satisfying tuple $\tau$ including $(x_i, a)$. Consider any constraint $c_k$ intersecting with $c_j$ and the value $\tau[x_l]$ of the intersection variable $x_l$. As $c_k$ is GAC, we can extend $\tau[x_l]$ to satisfy $c_k$. $\tau$ can be extended similarly for any constraint intersecting with $c_j$. The problem is thus maxRPWC (and hence rPIC and RPWC). □

### 4.2. Binary constraints

Due to Theorem 2, on a normalized binary CSP, where constraints on the same pair of variables are combined into a single constraint, maxRPWC, rPIC, and RPWC are equivalent to AC. Therefore, these consistencies are strictly weaker than the corresponding variable based consistencies maxRPC, PIC, and RPC. We now analyze the pruning power of maxRPWC, rPIC, and RPWC on non-normalized binary problems. We might expect maxRPWC, rPIC, and rPWC to reduce to the corresponding variable based definitions. However, this is not the case.

**Theorem 3.** *On non-normalized binary constraints*:[1]

$$\begin{array}{ccccccccccccccc}
\max \text{RPC} & \to & \text{PIC} & \to & \text{RPC} & \to & (1,1)\text{-cons.} & \leftrightarrow & \max \text{RPWC} & \to & \text{rPIC} & \to & \text{RPWC} & \to & \text{rel AC} \\
\otimes & & \otimes & & \uparrow & & & & & & & & & & \\
\text{SAC} & & \text{SAC} & & \text{SAC} & & & & & & & & & &
\end{array}$$

**Proof.** Relations between maxRPC, PIC, RPC and $(1, 1)$-consistency come from [10]. By definition, maxRPWC is stronger than rPIC, rPIC is stronger than RPWC, and RPWC is stronger than AC.

To show maxRPWC $\to$ rPIC, consider the problem with variables $x_1, x_2$ taking values in $\{0, 1, 2\}$ and constraints $c_1 \equiv x_1 + x_2 \neq 0$, $c_2 \equiv x_1 + x_2 \neq 1$, $c_3 \equiv x_1 + x_2 \neq 2$. Value 0 of $x_1$ is rPIC because for each pair of constraints it has a GAC-support satisfying both of them. All other values are rPIC by similar reasoning. However, value 0 of $x_1$ is not maxRPWC as it does not have any GAC-support in $c_1$ that satisfies both $c_2$ and $c_3$.

[1] The relation between SAC and maxRPC and PIC is different from that in [10] where CSPs are normalized.

To show rPIC $\to$ RPWC, consider the problem with variables $x_1, x_2$ taking values in $\{0, 1, 2, 3\}$ and constraints $c_1(x_1, x_2) = \{00, 01, 10, 11, 22, 33\}$, $c_2(x_1, x_2) = \{02, 03, 10, 11, 22, 33\}$. Value 0 of $x_1$ is RPWC because for each constraint it has more than one GAC-support. All other values are RPWC as they belong to a solution. However, value 0 of $x_1$ is not rPIC as it does not have any GAC-support in $c_1$ that satisfies $c_2$.

To show RPWC $\to$ rel AC, we first observe that rel AC is equivalent to AC on binary constraints. Thus, RPWC is stronger than rel AC. Then, consider the problem with variables $x_1, x_2$ taking values in $\{0, 1\}$ and constraints $c_1 \equiv x_1 + x_2 \neq 0$, $c_2 \equiv x_1 + x_2 \neq 1$. The problem is rel AC because on each constraint each value can be extended to a GAC-support. However, value 0 of $x_1$ is not RPWC as its unique GAC-support in $c_1$ does not satisfy $c_2$.

We show that maxRPWC is stronger than $(1, 1)$-consistency. Consider a binary problem which is not $(1, 1)$-consistent. There exists a value $a$ for a variable $x_i$ that cannot be extended to some variable $x_j$. Thus, there exists a set $C_{ij}$ of constraints on $x_i, x_j$ such that any value $b$ for $x_j$ is such that $(a, b)$ violates one of the constraints in $C_{ij}$. Take any constraint $c$ from $C_{ij}$. For every GAC-support $\tau$ of $a$ on $c$, there exists a constraint $c'$ in $C_{ij}$ which rejects $\tau$. As a result, $a$ for $x_i$ is not maxRPWC.

We show that $(1, 1)$-consistency is stronger than maxRPWC. Consider a binary CSP $P$. If $P$ is not AC, it is trivially not $(1, 1)$-consistent. So, suppose $P$ is AC but value $a$ for variable $x_i$ is not maxRPWC. This means that there exists a constraint $c(x_i, x_j)$ such that for any GAC-support $\tau$ of $(x_i, a)$ on $c$, there exists a second constraint $c'$ on which $\tau$ cannot be extended. If $c'$ intersects $c$ on a single variable, AC guarantees that $\tau$ can be extended to $c'$. So all these constraints $c'$ rejecting the GAC-supports of $a$ on $c$ involve the same variables as $c$. Therefore, value $a$ is not $(1, 1)$-consistent.

To show SAC is incomparable with maxRPC and PIC, we must find a problem which is SAC but not PIC (and thus not maxRPC), and a problem which is maxRPC (and thus PIC) but not SAC. Consider the problem $P$ on variables $x_1$ with domain $\{0, 1\}$ and variables $x_2, x_3$ with domain $\{0, 1, 2\}$, and the constraints $c_1(x_1, x_2) = c_2(x_1, x_3) = \{00, 01, 02, 11, 12\}$, $c_3(x_2, x_3) = \{00, 01, 02, 10, 11, 20, 22\}$ and $c_4(x_2, x_3) = \{00, 01, 02, 10, 12, 20, 21\}$. $(x_1, 1)$ is SAC because $P_{(x_1,1)}$ is AC. All other values are SAC because they belong to a solution of $P$. But $(x_1, 1)$ is not PIC (and not maxRPC) because it cannot be extended to $x_2$ and $x_3$. On the other hand, we know from [10] that there exist networks which are PIC and maxRPC but not SAC.

To show SAC $\to$ RPC, we first show that SAC is stronger than RPC. Consider a problem $P$ with a value $a$ for some variable $x_i$ which is not RPC. Then, there is a variable $x_j$ where a value, say $b$, is the only consistent extension of $a$ on $x_j$ and there exists $x_k$ such that the pair $(a, b)$ cannot be extended to $x_k$. If we assign $x_i$ with $a$ (in $P_{(x_i,a)}$), AC will leave only $b$ in $D(x_j)$. If $\exists c \in D(x_k)$ which is AC in $P_{(x_i,a)}$, this means that it is consistent with both $a$ on $x_i$ and $b$ on $x_j$, which contradicts that RPC failed to extend $(a, b)$ to $x_k$. As for strictness, we know from [10] that there exist binary networks which are RPC and not SAC. $\square$

These results can easily be generalized to show that, on binary constraints, inverse $m$-consistency is strictly stronger than relational inverse $m$-consistency for all $m > 1$.

### 4.3. Binary encodings of non-binary constraints

One way to deal with non-binary constraints is to encode them into binary ones, and apply binary techniques (as for example in [2]). We now position RPWC, rPIC and maxRPWC with respect to other consistencies enforced in the hidden variable and dual encodings of a non-binary problem. Recall that in the hidden variable and dual encodings of a non-binary problem each constraint $c$ is turned into a hidden variable $H_c$ or a dual variable $V_c$ having as domain the valid tuples allowed by the constraint. The hidden variable encoding is defined on the original variables plus the hidden variables. Its constraints ensure that a hidden variable $H_c$ and an original variable $x_k$ involved in $c$ agree on the value taken by $x_k$. The dual encoding is defined on the dual variables only. Its constraints ensure that two variables $V_{c_i}$ and $V_{c_j}$ take values that agree on the original variables common to $c_i$ and $c_j$. Local consistency A is denoted by A when used on the original representation, $\mathrm{A}_{\mathrm{hidden}}$ on the hidden variable encoding, and $\mathrm{A}_{\mathrm{dual}}$ on the dual encoding. Each result has the precondition that the non-binary constraints are GAC so that the hidden variable or dual encoding is node consistent (and not trivially unsatisfiable).

**Theorem 4.** *On* (*non-binary*) *constraints which are* GAC:

$$
\begin{array}{lllllll}
 & & \mathrm{SAC}_{\mathrm{hidden}} & & \mathrm{SAC}_{\mathrm{hidden}} & & \mathrm{SAC}_{\mathrm{hidden}} \\
 & & \otimes & & \otimes & & \downarrow \\
\mathrm{SAC}_{\mathrm{dual}} & \rightarrow & \max \mathrm{RPC}_{\mathrm{dual}} & \rightarrow & \mathrm{PIC}_{\mathrm{dual}} & \rightarrow & \mathrm{RPC}_{\mathrm{dual}} \\
\downarrow & & \otimes & & & & \downarrow \\
\mathrm{SAC}_{\mathrm{hidden}} & \rightarrow & \mathrm{SGAC} & & \mathrm{SGAC} & & \mathrm{AC}_{\mathrm{dual}} \\
 & & \downarrow & & \otimes & & \downarrow \\
\mathrm{GAC} & \leftarrow & \mathrm{RPWC} & \leftarrow & \mathrm{rPIC} & \leftarrow & \max \mathrm{RPWC} \\
\updownarrow & & & & & & \\
\max \mathrm{RPC}_{\mathrm{hidden}} & \leftrightarrow & \mathrm{PICY}_{\mathrm{hidden}} & \leftrightarrow & \mathrm{RPC}_{\mathrm{hidden}} & \leftrightarrow & \mathrm{AC}_{\mathrm{hidden}}
\end{array}
$$

**Proof.** We show $\mathrm{SAC}_{\mathrm{hidden}}$ is incomparable with $\mathrm{PIC}_{\mathrm{dual}}$ and with $\max \mathrm{RPC}_{\mathrm{dual}}$. Consider problem $P_1$ on variables $x_1$ to $x_4$ with domain $\{0, 1\}$ and the constraints $x_1 \neq x_2$, $x_2 \neq x_3$, $x_3 \neq x_4$, $x_4 = x_1$. Its dual encoding is maxRPC (and so PIC) because it is AC and there is no triple of dual variables pairwise connected by constraints. However the hidden encoding of $P_1$ is not SAC because whatever the hidden variable we assign, AC leads to a wipe out. Consider problem $P_2$ on variables $x_1$ to $x_6$ with domain $\{0, 1\}$ and the constraints $c_1(x_1, x_2, x_3), c_2(x_2, x_5, x_6), c_3(x_3, x_5, x_6)$. $c_1 = \{001, 010, 111\}, c_2 = \{000, 011, 101, 110\}$ and $c_3 = \{001, 010, 100, 111\}$. $t = 111$ for $c_1$ is the only tuple of a constraint in $P_2$ that does not belong to any solution. If we assign $H_{c_1}$ with $t$ in the hidden encoding of $P_2$, no wipe out is detected because original variables $x_5$ and $x_6$ keep their two values. However, the dual encoding of $P_2$ is not PIC (and so not maxRPC) because $t$ for $V_{c_1}$ cannot be extended to $V_{c_2}$ and $V_{c_3}$ simultaneously.

We show $\mathrm{SAC}_{\mathrm{dual}} \rightarrow \mathrm{SAC}_{\mathrm{hidden}} \rightarrow \mathrm{SGAC}$. By definition $\mathrm{SAC}_{\mathrm{dual}}$ is stronger than $\mathrm{SAC}_{\mathrm{hidden}}$. As for strictness, we already presented problem $P_2$ which is $\mathrm{SAC}_{\mathrm{hidden}}$ but is not $\mathrm{PIC}_{\mathrm{dual}}$ and so not $\mathrm{SAC}_{\mathrm{dual}}$. The relation $\mathrm{SAC}_{\mathrm{hidden}} \rightarrow$ SGAC has been proved in [20].

We show $\mathrm{SAC}_{\mathrm{hidden}} \rightarrow \mathrm{RPC}_{\mathrm{dual}}$. To show $\mathrm{SAC}_{\mathrm{hidden}}$ is stronger than $\mathrm{RPC}_{\mathrm{dual}}$, consider a problem such that the dual encoding is not AC. Then, there exists $v$ in $D(V_{c_i})$ for some $c_i$, there exists $V_{c_j}$ for some $c_j$, such that there is no $w \in D(V_{c_j})$ compatible with $v$. Thus, when assigning $H_{c_i}$ with value $v$, AC will fail on $H_{c_j}$ in the hidden encoding. Hence, if $\mathrm{SAC}_{\mathrm{hidden}}$ is not stronger than $\mathrm{RPC}_{\mathrm{dual}}$ there necessarily exists a network such that its hidden encoding is SAC and its dual encoding is AC but not RPC. Consider such a problem. In the dual encoding, there exists $v$ in $D(V_{c_i})$ for some $c_i$, and $w$ in $D(V_{c_j})$ for some $c_j$ such that $w$ is the only value consistent with $v$ in $D(V_{c_j})$ and such that there exists $V_{c_k}$ with no value consistent with both $v$ and $w$. In the hidden variable encoding, when assigning $v$ to $H_{c_i}$, AC forces $H_{c_j}$ to take value $w$ because this is the only value consistent with $v$ and every original variable $x_p$ in the scope of $c_i$ has been forced to take as value the projection of $v$ on $x_p$. Thus, after AC, $D(H_{c_k})$ contains only values that are consistent with both $v$ and $w$. But we know that no such value exists because the dual encoding is not RPC. So, AC wipes out and $v$ is not SAC in the hidden variable encoding. As for strictness, we already showed that there exists problem $P_1$ such that the dual encoding is PIC (and so RPC) and the hidden variable encoding is not SAC.

We show $\mathrm{AC}_{\mathrm{dual}} \rightarrow \max \mathrm{RPWC}$. $\mathrm{AC}_{\mathrm{dual}}$ is known to be stronger than PWC + GAC [15]. Now, we have shown in Theorem 1 that PWC + GAC $\rightarrow$ maxRPWC.

It is known that GAC $\leftrightarrow \mathrm{AC}_{\mathrm{hidden}}$ and, due to the topology of the hidden variable encoding, $\mathrm{NIC}_{\mathrm{hidden}} \leftrightarrow \mathrm{AC}_{\mathrm{hidden}}$ [21]. Since NIC is strictly stronger than maxRPC, PIC, RPC, and AC [10], it immediately follows that $\max \mathrm{RPC}_{\mathrm{hidden}} \leftrightarrow \mathrm{PIC}_{\mathrm{hidden}} \leftrightarrow \mathrm{RPC}_{\mathrm{hidden}} \leftrightarrow \mathrm{AC}_{\mathrm{hidden}}$.

We show that $\mathrm{SAC}_{\mathrm{dual}} \rightarrow \max \mathrm{RPC}_{\mathrm{dual}} \rightarrow \mathrm{PIC}_{\mathrm{dual}} \rightarrow \mathrm{RPC}_{\mathrm{dual}} \rightarrow \mathrm{AC}_{\mathrm{dual}}$. By definition, $\mathrm{SAC}_{\mathrm{dual}}$ is stronger than $\max \mathrm{RPC}_{\mathrm{dual}}$, which is stronger than $\mathrm{PIC}_{\mathrm{dual}}$, which is stronger than $\mathrm{RPC}_{\mathrm{dual}}$, which is stronger than $\mathrm{AC}_{\mathrm{dual}}$. To show strictness, consider problem $P_1$ above. It is $\max \mathrm{RPC}_{\mathrm{dual}}$ but not $\mathrm{SAC}_{\mathrm{hidden}}$ and so not $\mathrm{SAC}_{\mathrm{dual}}$. Consider problem $P_3$ on variables $x_1$ to $x_5$ with domain $\{0, 1\}$ and the constraints $c_1(x_1, x_2, x_3)$, $c_2(x_2, x_4)$, $c_3(x_3, x_5)$, $c_4(x_1, x_4, x_5)$. $c_1 = \{000, 110, 111\}$, $c_2 = c_3 = \{00, 11\}$, and $c_4 = \{000, 100, 111\}$. $P_3$ is $\mathrm{PIC}_{\mathrm{dual}}$, but value 110 for $V_{c_1}$ is not $\max \mathrm{RPC}_{\mathrm{dual}}$. Consider problem $P_4$ on variables $x_1$ to $x_5$ with domain $\{0, 1\}$ and the constraints $c_1(x_1, x_2, x_3)$, $c_2(x_2, x_4, x_5)$, $c_3(x_3, x_4, x_5)$. $c_1 = \{000, 011, 101\}$, $c_2 = c_3 = \{000, 001, 110, 111\}$. $P_4$ is $\mathrm{RPC}_{\mathrm{dual}}$, but value 101 for $V_{c_1}$ is not $\mathrm{PIC}_{\mathrm{dual}}$. Consider problem $P_5$ on variables $x_1$ to $x_4$ with domain $\{0, 1\}$ and the constraints $c_1(x_1, x_2, x_3)$, $c_2(x_2, x_4)$, $c_3(x_3, x_4)$. $c_1 = \{000, 011, 101\}$, $c_2 = c_3 = \{00, 11\}$. $P_5$ is $\mathrm{AC}_{\mathrm{dual}}$, but value 101 for $V_{c_1}$ is not $\mathrm{RPC}_{\mathrm{dual}}$.

We still need to position SGAC. The problem $P_1$ above is not SGAC but its dual encoding is maxRPC. The problem $P''$ in Theorem 1 is SGAC but not rPIC. Therefore, consistencies $\max \mathrm{RPC}_{\mathrm{dual}}$, $\mathrm{PIC}_{\mathrm{dual}}$, $\mathrm{RPC}_{\mathrm{dual}}$, $\mathrm{AC}_{\mathrm{dual}}$, maxRPWC and rPIC are incomparable to SGAC. Finally, we know from Theorem 1 that SGAC → RPWC. This completes the picture. □

## 5. Algorithms

Verfaillie, Martinez, and Bessiere, proposed a generic AC-7 based algorithm for domain filtering local consistencies [23]. Although the algorithm was primarily intended for binary constraints, it can be easily adapted to enforce rPIC. However, adapting it to enforce RPWC and maxRPWC is much more involved. In what follows we propose a number of algorithms that are much simpler to implement compared to the generic algorithm of [23].

First we describe the schema of a simple generic AC-3 based algorithm for non-binary domain filtering consistencies, and show how this algorithm can be instantiated to apply RPWC, rPIC, and maxRPWC. Then we describe two alternative algorithms for maxRPWC, which is stronger and, as experimental results show, more cost-effective than RPWC and rPIC. The first of these algorithms has better time complexity than the generic algorithm but worse space complexity, while the second achieves a balance between the other two in terms of time and space complexity. Both these algorithms can be easily modified to apply RPWC and rPIC.

### *5.1. A generic algorithm for domain filtering consistencies*

Fig. 2 describes the framework of a generic algorithm for domain filtering consistencies. This is based on coarse-grained GAC algorithms like GAC-3 [18,19] and GAC2001/3.1 [8].

Algorithm `DFcons` takes as input a (non-binary) CSP $P$, a specified domain filtering local consistency DFC and a parameter *current-var*, and enforces DFC on $P$. The parameter *current-var* is set to $-1$ if the algorithm is used stand-alone (e.g. for preprocessing a problem with a given consistency). Otherwise if the algorithm is applied during search (e.g. to maintain a given consistency), *current-var* is the currently assigned variable. `DFcons` uses a list $Q$ (that can be implemented as a stack or as a queue) of constraints to propagate value deletions, and works as follows. If used stand-alone, the algorithm initially puts all constraints in $Q$ (line 1). Else if the algorithm is applied during search, it calls procedure `Enqueue` to initialize $Q$ with the appropriate constraints (line 2). These include each constraint $c_m$ involving *current-var* and each constraint intersecting with $c_m$ on more than one variable. We now explain why these constraints are added to $Q$ for the case where DFC is rPIC or maxRPWC. A similar explanation holds for RPWC.

A constraint $c_m$ that involves *current-var* needs to be added to $Q$ as the assignment of *current-var* effectively means that all but its given value have been removed from its domain. As a result, some variables involved in $c_m$ may have lost their GAC-support. A constraint $c_l$ that intersects with $c_m$ (on more than one variable) needs to be added to $Q$ because the following situation may occur. For some value $a \in x_i$, where $x_i \in var(c_l)$, there may now not exist any

**function** `DFcons`($P$, DFC, *current-var*)
1: **if** *current-var* $= -1$, put all constraints in $Q$;
2: **else** `Enqueue`(*current-var*, $-1$);
3: **while** $Q$ is not empty
4: pop constraint $c_i$ from $Q$;
5: **for** each unassigned variable $x_j$ where $x_j \in var(c_i)$
6: **if** `Revise`($x_j, c_i$, DFC)$> 0$
7: **if** $D(x_j)$ is empty **return** INCONSISTENCY;
8: `Enqueue`($x_j, c_i$);
9: **return** CONSISTENCY;

**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. 2. A generic algorithm for domain filtering consistencies.

tuple $\tau \in rel(c_l)$ that includes assignment $(x_i, a)$ and can be consistently extended to $c_m$. This is because tuples that previously were consistent extensions of $\tau$ may now have become invalid.

Then constraints are sequentially removed from $Q$ (line 4) and the domains of the variables involved in these constraints are revised. For each such constraint $c_i$ and variable $x_j$, the revision is performed using function `Revise`($x_j$, $c_i$, DFC). The implementation of `Revise` depends on the local consistency DFC being applied. If after the revision the domain of $x_j$ becomes empty then the algorithm detects the inconsistency and terminates (line 7). Otherwise, if the domain of $x_j$ is pruned then constraints are added to $Q$ for further propagation using procedure `Enqueue`($x_j$) (line 8). In this procedure, each constraint $c_m$ involving $x_j$ (except $c_i$) and each constraint intersecting with $c_m$ on more than one variable (and that is not already in $Q$) will be put in $Q$. If at some point $Q$ becomes empty, the algorithm terminates having successfully enforced DFC on $P$.

We now show how we can derive algorithms for maxRPWC, rPIC, and RPWC by instantiating function `Revise` appropriately.

### 5.2. maxRPWC-1: An algorithm for maxRPWC

From the definition of maxRPWC we can immediately derive a simple algorithm by extending a GAC algorithm so that whenever it finds a GAC-supporting tuple for a value in a constraint $c_i$, it also checks if this tuple can be extended to a valid tuple in all constraints intersecting with $c_i$. Fig. 3 gives function `Revise` for maxRPWC-1, an algorithm for maxRPWC derived from `DFcons`.

In function `Revise` of maxRPWC-1, for each value $a$ in $D(x_j)$, we first look for a tuple in $rel(c_i)$ that GAC-supports it. As in GAC2001/3.1, for each constraint $c_i$ and each $a \in D(x_j)$, where $x_j \in var(c_i)$, we keep a pointer $lastGAC_{x_j,a,c_i}$ (initialized to the first tuple in $rel(c_i)$). This is now the most recently discovered tuple in $rel(c_i)$ that GAC-supports value $a$ of variable $x_j$ **and** can be extended to a valid tuple in all constraints that intersect with $c_i$. If this tuple is valid then we know that $a$ is GAC-supported. Otherwise, we look for a new GAC-support starting from the tuple immediately after $lastGAC_{x_j,a,c_i}$ in the lexicographic order (line 3). If $lastGAC_{x_j,a,c_i}$ is valid or a new GAC-support is found then the algorithm checks if the GAC-support (tuple $\tau$) can be extended to all intersecting constraints. Note that this check must be performed in the case where $lastGAC_{x_j,a,c_i}$ is valid, since this tuple may have lost its PW-supports on some of $c_i$'s intersecting constraints.

To check if $\tau$ has PW-supports, maxRPWC-1 iterates over each constraint $c_m$ that intersects with $c_i$ on more than one variable. Constraints intersecting on one variable are not considered because maxRPWC offers here no more pruning than GAC. The algorithm checks if there is a tuple $\tau' \in rel(c_m)$ such that $\tau'$ is a PW-support of $\tau$ (lines 5–6). If such tuples are found for all intersecting constraints then $lastGAC_{x_j,a,c_i}$ is updated (line 8). If no PW-support is found on some intersecting constraint, then the iteration stops (line 7) and the algorithm looks for a new GAC-support. If no GAC-support that can be extended to all intersecting constraints is found, $a$ is removed from $D(x_j)$ (line 9). In this case, each constraint $c_m$ involving $D(x_j)$ and each constraint intersecting with $c_m$ (that is not already in $Q$) will be put in $Q$ using procedure `Enqueue`. The algorithm terminates if a domain is wiped out, in which case the problem cannot be made maxRPWC, or if $Q$ becomes empty, in which case the problem is maxRPWC.

**function** `Revise`($x_j$, $c_i$, maxRPWC)
1: **for** each value $a \in D(x_j)$
2: PW ← FALSE;
3: **for** each valid $\tau (\in rel(c_i)) \geqslant_l lastGAC_{x_j,a,c_i}$, such that $\tau[x_j] = a$
4: PW ← TRUE;
5: **for** each $c_m \neq c_i$ such that $|var(c_i) \cap var(c_m)| > 1$
6: **if** $\nexists$ valid $\tau' (\in rel(c_m))$ such that
$\tau[var(c_i) \cap var(c_m)] = \tau'[var(c_i) \cap var(c_m)]$
7: PW ← FALSE; **break**;
8: **if** PW=TRUE $lastGAC_{x_j,a,c_i} \leftarrow \tau$; **break**;
9: **if** PW=FALSE remove $a$ from $D(x_j)$;
10: **return** number of deleted values;

Fig. 3. Function `Revise` of algorithm maxRPWC-1.

Finally, one extra feature of the algorithm that is not shown in Figs. 2 and 3, to avoid complicating the pseudocode, is the following. During each revision of a constraint $c_i$, once we have established that a tuple $\tau$ is a GAC-support for a value $a \in D(x_j)$ that has PW-supports in all intersecting constraints, we do not check if the values in $\tau$ of the other variables involved in $c_i$ are maxRPWC, since it is certain that they are. In this way we take advantage of the multi-directionality of supports to avoid some redundant constraint checks.

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

**Proof.** Let us denote by $k_i$ the number of variables involved in $c_i$ and by $p_{im}$ the total number of variables involved in the two constraints $c_i$ and $c_m$. The complexity is determined by the number of constraint checks performed in total, in all calls to function `Revise`. In the inner loop of `Revise` (lines 5–6), maxRPWC-1 verifies if $lastGAC_{x_j,a,c_i}$ has PW-supports in the at most $e-1$ constraints intersecting $c_i$ on at least two variables. For each such constraint $c_m$ the algorithm checks at most $d^{p_{im}-k_i}$ tuples, i.e. those that take the same values in variables $var(c_i) \cap var(c_m)$ as in $lastGAC_{x_j,a,c_i}$. The cost of each such check is $O(k)$ if we assume that the cost of a constraint check is linear to the arity of the constraint. Therefore, the cost of lines 5–6 is bounded by $K_{jia} = \sum_{c_m \in C \setminus \{c_i\}} kd^{p_{im}-k_i}$. Given a variable $x_j$ and a constraint $c_i$, these two lines are performed for each value $a$ each time function `Revise`($x_j$, $c_i$, maxRPWC) is called or each time a *new* GAC-support $lastGAC_{x_j,a,c_i}$ is found. `Revise`($x_j$, $c_i$, maxRPWC) can be called at most $nd$ times. This is because every one of the $n$ variables may either belong to $var(c_i)$ or participate in a constraint that intersects with $c_i$. In this case every deletion of a value from a variable will force `Enqueue` to add $c_i$ to $Q$ and subsequently cause a call to `Revise`. $lastGAC_{x_j,a,c_i}$ cannot change more than $d^{k_i-1}$ times because maxRPWC-1 only checks the tuples that contain the assignment $(x_j, a)$ and it only checks tuples that have not been checked before. So, lines 5–6 are performed at most $L_{jia} = nd + d^{k_i-1}$ times for each variable $x_j$, value $a$, and constraint $c_i$. $L_{jia}$ is also the number of times a tuple can be checked as GAC-support for $(x_j, a)$ on $c_i$ at a cost $O(k)$ (line 3). Thus, for a variable $x_j$, value $a$, and constraint $c_i$, the complexity is bounded above by $M_{jia} = L_{jia} \cdot (k + K_{jia}) = (nd + d^{k_i-1}) \cdot (k + \sum_{c_m \in C \setminus \{c_i\}} kd^{p_{im}-k_i})$. Assuming that $d^{k-1} > nd$, this gives a complexity in $O(ekd^{p-1})$. Since there are at most $d$ values in $D(x_j)$, $k$ variables in $var(c_i)$, and $e$ constraints in $C$, the total complexity is bounded above by $ekd \cdot ekd^{p-1}$. This gives a time complexity in $O(e^2k^2d^p)$, assuming that $d^{k-1} > nd$. □

The space complexity of maxRPWC-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 $k$.

### *5.3. rPIC-1: An algorithm for rPIC*

Fig. 4 gives function `Revise` for rPIC-1, an algorithm for rPIC derived from `DFcons`. The algorithm is similar to maxRPWC-1 and works as follows.

In each call to `Revise`($x_j$, $c_i$, rPIC), for each value $a$ of $D(x_j)$ we iterate over each constraint $c_m$ that intersects with $c_i$[2] (line 2) to look for a tuple in $rel(c_i)$ that GAC-supports $a$ and can be extended to a valid tuple in $rel(c_m)$. To do this we store a pointer $lastGAC_{x_j,a,c_i,c_m}$ for each constraint $c_m$ that intersects with $c_i$. This is now the most recently discovered tuple in $rel(c_i)$ that GAC-supports value $a$ of variable $x_j$ **and** can be extended to a valid tuple in $c_m$. If this tuple is valid then we know that $a$ is GAC-supported. Otherwise, we look for a new GAC-support starting from the tuple immediately "after" $lastGAC_{x_j,a,c_i,c_m}$ (line 4). If $lastGAC_{x_j,a,c_i,c_m}$ is valid or a new GAC-support is found then the algorithm checks if the GAC-support (tuple $\tau$) can be extended to $c_m$. To do this, rPIC-1 checks if there is a tuple $\tau' \in rel(c_m)$ such that $\tau'$ is a PW-support of $\tau$ (line 5). If such a tuple is found then $lastGAC_{x_j,a,c_i,c_m}$ is updated (line 6). If no such tuple is found, the algorithm looks for a new GAC-support in $c_i$. The process is repeated for all constraints intersecting with $c_i$. If no GAC-support that can be extended to some intersecting constraint is found, $a$ is removed from $D(x_j)$ (line 8). In this case, each constraint $c_m$ involving $D(x_j)$ and each constraint intersecting

[2] As in maxRPWC-1, we only consider constraints intersecting on more than one variable with $c_i$.

**function** Revise($x_j$, $c_i$, rPIC)
1: **for** each value $a \in D(x_j)$
2: **for** each $c_m$ such that $|var(c_i) \cap var(c_m)| > 1$
3: PW $\leftarrow$ FALSE;
4: **for** each valid $\tau(\in\ rel(c_i)) \geqslant_l lastGAC_{x_j,a,c_i,c_m}$, such that $\tau[x_j] = a$
5: **if** $\exists$ valid $\tau'(\in\ rel(c_m))$ such that
$\tau[var(c_i) \cap var(c_m)] = \tau'[var(c_i) \cap var(c_m)]$
6: $lastGAC_{x_j,a,c_i,c_m} \leftarrow \tau$;
7: PW $\leftarrow$ TRUE; **break**;
8: **if** PW=FALSE remove $a$ from $D(x_j)$; **break**;
9: **return** number of deleted values;

Fig. 4. Function Revise of algorithm rPIC-1.

with $c_m$ (that is not already in $Q$) will be put in $Q$ using procedure Enqueue. The algorithm terminates if a domain is wiped out, in which case the problem cannot be made rPIC, or if $Q$ becomes empty, in which case the problem is rPIC.

**Proposition 2.** *The worst-case time complexity of algorithm* rPIC-1 *is* $O(e^2k^2d^p)$.

**Proof.** As in maxRPWC-1, the complexity is determined by the number of constraint checks performed in total, in all calls to function Revise. In the inner loop of Revise (line 5), rPIC-1 verifies if $lastGAC_{x_j,a,c_i,c_m}$ has PW-support in $c_m$. For such a constraint $c_m$ the algorithm checks at most $d^{p_{im}-k_i}$ tuples, i.e. those that take the same values in variables $var(c_i) \cap var(c_m)$ as in $lastGAC_{x_j,a,c_i,c_m}$. The cost of each such check is $O(k)$. Therefore, the cost of lines 5–7 is bounded by $K_{jima} = kd^{p_{im}-k_i}$. For a variable $x_j$, a value $a$, and constraints $c_i$, $c_m$, these lines are performed each time function Revise($x_j$, $c_i$, rPIC) is called or each time $lastGAC_{x_j,a,c_i,c_m}$ is modified. As in maxRPWC-1, Revise($x_j$, $c_i$, rPIC) can be called at most $nd$ times. $lastGAC_{x_j,a,c_i,c_m}$ cannot change more than $d^{k_i-1}$ times because rPIC-1 only checks the tuples that contain the assignment $(x_j, a)$ and it only checks tuples that have not been checked before. So, lines 5–7 are performed at most $L_{jima} = nd + d^{k_i-1}$ times for each variable $x_j$, value $a$, constraint $c_i$ that involves $x_j$ and any other constraint $c_m$ intersecting $c_i$. $L_{jima}$ is also the number of times a tuple can be checked as GAC-support for $(x_j, a)$ on $c_i$ (with respect to $c_m$) at a cost $O(k)$ (line 4). Therefore, the cost is bounded by $L_{jima} \cdot (k + K_{jima})$. For all the constraints intersecting $c_i$ the cost is bounded by $\sum_{c_m \in C} L_{jima} \cdot (k + K_{jima}) = \sum_{c_m \in C}(nd + d^{k_i-1}) \cdot (k + kd^{p_{im}-k_i})$. Assuming that $d^{k-1} > nd$, this gives a complexity in $O(ekd^{p-1})$. Since there are at most $d$ values in $D(x_j)$, $k$ variables in $var(c_i)$, and $e$ constraints in $C$ that involve $x_j$, the total complexity is bounded above by $ekd \cdot ekd^{p-1}$. This gives a time complexity in $O(e^2k^2d^p)$, assuming that $d^{k-1} > nd$. □

The space complexity of rPIC-1, determined again by the space required for the *lastGAC* data structure, is $O(e^2kd)$ for extensional constraints and $O(e^2k^2d)$ for intensional ones.

*5.4. RPWC-1: An algorithm for RPWC*

Fig. 5 gives function Revise for RPWC-1, an algorithm for RPWC derived from DFcons. The main idea of algorithm RPWC-1 is the following. In each call to Revise($x_j$, $c_i$, RPWC) of RPWC-1, for each value $a$ of $D(x_j)$, we look for two tuples in $rel(c_i)$ that GAC-support it. If we only find one GAC-support then we have to check if it can be extended to all constraints intersecting with $c_i$.

To implement the above idea, for each constraint $c_i$ and each $a \in D(x_j)$, where $x_j \in var(c_i)$, we keep two pointers $lastGAC1_{x_j,a,c_i}$ (initialized to the first tuple in $rel(c_i)$) and $lastGAC2_{x_j,a,c_i}$ (initialized to the second tuple in $rel(c_i)$). $lastGAC1_{x_j,a,c_i}$ will always represent the smallest GAC-support and $lastGAC2_{x_j,a,c_i}$ the second smallest. The value *NIL* for $lastGAC2_{x_j,a,c_i}$ means that all possible extensions for $(x_j, a)$ on $c_i$ have been explored. If $lastGAC2_{x_j,a,c_i}$ is not valid (and not *NIL*), we look for a new GAC-support starting immediately after $lastGAC2_{x_j,a,c_i}$ (lines 2–3). If we find one, we update $lastGAC2_{x_j,a,c_i}$, otherwise we set $lastGAC2_{x_j,a,c_i}$ to *NIL* (lines 4–5). We now check the validity of

**function** `Revise`($x_j$, $c_i$, RPWC)
1: **for** each value $a \in D(x_j)$
2: **if** $lastGAC2_{x_j,a,c_i} \neq NIL$ and $lastGAC2_{x_j,a,c_i}$ is not valid
3: **if** $\exists$ valid $\tau(\in rel(c_i)) >_l lastGAC2_{x_j,a,c_i}$ such that $\tau[x_j] = a$
4: $lastGAC2_{x_j,a,c_i} \leftarrow \tau$;
5: **else** $lastGAC2_{x_j,a,c_i} \leftarrow NIL$;
6: **if** $lastGAC2_{x_j,a,c_i} \neq NIL$ and $lastGAC1_{x_j,a,c_i}$ is not valid
7: $lastGAC1_{x_j,a,c_i} \leftarrow lastGAC2_{x_j,a,c_i}$;
8: **if** $\exists$ valid $\tau(\in rel(c_i)) >_l lastGAC2_{x_j,a,c_i}$ such that $\tau[x_j] = a$
9: $lastGAC2_{x_j,a,c_i} \leftarrow \tau$;
10: **else** $lastGAC2_{x_j,a,c_i} \leftarrow NIL$;
11: **if** $lastGAC1_{x_j,a,c_i}$ is not valid PW $\leftarrow$ FALSE;
12: **else if** $lastGAC2_{x_j,a,c_i} = NIL$
13: PW $\leftarrow$ `FindPWsupports`($c_i$, $lastGAC1_{x_j,a,c_i}$);
14: **else** PW=TRUE;
15: **if** PW=FALSE remove $a$ from $D(x_j)$;
16: **return** number of deleted values;

**function** `FindPWsupports`($c_i$, $\tau$);
1: PW $\leftarrow$ TRUE;
2: **for** each $c_m \neq c_i$ such that $|var(c_i) \cap var(c_m)| > 1$
3: **if** $\nexists$ valid $\tau'(\in rel(c_m))$ such that $\tau[var(c_i) \cap var(c_m)] = \tau'[var(c_i) \cap var(c_m)]$
4: PW $\leftarrow$ FALSE; **break**;
5: **return** PW;

Fig. 5. Function `Revise` for algorithm RPWC-1.

$lastGAC1_{x_j,a,c_i}$. If search is not exhausted and $lastGAC1_{x_j,a,c_i}$ is not valid (line 6), we assign $lastGAC1_{x_j,a,c_i}$ with the tuple stored in $lastGAC2_{x_j,a,c_i}$ (necessarily valid), and we search for a new GAC-support greater than $lastGAC2_{x_j,a,c_i}$ (lines 7–8). If we find one, we update $lastGAC2_{x_j,a,c_i}$, otherwise we set $lastGAC2_{x_j,a,c_i}$ to *NIL* (lines 9–10).

At this point, if $lastGAC1_{x_j,a,c_i}$ is not a valid GAC-support, $(x_j, a)$ has no GAC-support on $c_i$ and thus $a$ must be removed from $D(x_j)$ (lines 11 and 15). Otherwise, if $lastGAC2_{x_j,a,c_i}$ is *NIL* (line 12), this means that $(x_j, a)$ has a single GAC-support on $c_i$ and we must check if this only support ($lastGAC1_{x_j,a,c_i}$) has PW-supports on all intersecting constraints. (This is done by means of function `FindPWsupports` in line 13). If this is not the case then $a$ will be deleted from $D(x_j)$ (line 15). Otherwise, if both pointers are valid, nothing is done (line 14).

In case of value deletion, each constraint $c_m$ involving $D(x_j)$ and each constraint intersecting with $c_m$ (that are not already in $Q$) will be put in $Q$ using procedure `Enqueue` in the main algorithm (Fig. 2). The algorithm terminates if a domain is wiped out, in which case the problem cannot be made RPWC, or if $Q$ becomes empty, in which case the problem is RPWC.

**Proposition 3.** *The worst-case time complexity of algorithm* RPWC-1 *is* $O(ne^2k^2d^k)$.

**Proof.** In each call to `Revise`($x_j$, $c_i$, RPWC), for each $a \in D(x_j)$, lines 2–10 look for a new GAC-support for $a$ on $c_i$ if one of $lastGAC1_{x_j,a,c_i}$ and $lastGAC2_{x_j,a,c_i}$ is non-valid. The search for new supports always starts from the tuple after $lastGAC2_{x_j,a,c_i}$, which is the greatest of the two. So, for each value $a \in D(x_j)$, each tuple will be visited at most once. Therefore, for a variable $x_j$, a value $a$, and a constraint $c_i$, the total cost of lines 2–10 of `Revise` on all its calls is bounded above by $K_{jia} = kd^{k_i-1}$, where $d^{k_i-1}$ is the number of tuples in $c_i$ containing $a$ for $x_j$ to be checked at cost $O(k)$. In each call to `Revise`, the cost of lines 11–15 is bounded above by the cost of `FindPWsupports` which looks for a PW-support for $lastGAC1_{x_j,a,c_i}$ on the at most $e-1$ constraints intersecting $c_i$ on at least two variables. For each such constraint $c_m$, we check at most $d^{p_{im}-k_i}$ tuples, with $O(k)$ cost for each one. So, the cost of lines 11–15 is bounded above by $L_{jia} = \sum_{c_m \in C \setminus \{c_i\}} kd^{p_{im}-k_i}$. In the worst case, given $x_j, a, c_i$, lines 11–15 will be repeated in each of the *nd* possible calls to `Revise`. This is because we may repeatedly discover that the single GAC-support is valid, and each time we have to search for PW-supports for it. Thus, for each value $a$ of $x_j$ and each constraint $c_i$, the asymptotic cost of `Revise` is bounded above by $K_{jia} + nd \cdot L_{jia} = kd^{k_i - 1} + nd \sum_{c_m \in C \setminus \{c_i\}} kd^{p_{im} - k_i}$. This gives a complexity in $O(kd^{k-1} + nekd^{g+1})$ where $g = max_{c_i, c_m \in C}(p_{im} - k_i)$. Since there are at most $d$ values in $D(x_j)$, $k$ variables in $var(c_i)$, and $e$ constraints in $C$, the worst-case time complexity of RPWC-1 is in $O(ek^2d^k + ne^2k^2d^{g+2})$. In the worst case, $g$ can be equal to $k - 2$ because the algorithm only considers pairs of constraints sharing at least two variables. Therefore, the time complexity of RPWC-1 is in $O(ne^2k^2d^k)$. □

Note that if the number of intersecting variables is large, in which case $g$ is small, the worst-case time complexity of RPWC-1 is $O(ek^2d^k)$, the same as GAC2001/3.1. The space complexity of RPWC-1 is $O(ekd)$ for extensional constraints and $O(ek^2d)$ for intensional ones. This is determined by the space required for the GAC-supports.

### *5.5. maxRPWC-2: An improved algorithm for maxRPWC*

Although the asymptotic time complexities of the above algorithms are lower than that of the generic algorithm of [23], they can still be prohibitive in practice. We will now present maxRPWC-2, an alternative algorithm for maxRPWC which offers a significant improvement in terms of time complexity, but with an increase in the space complexity. Note that it is easy to design similar algorithms for RPWC and rPIC by appropriately modifying their `Revise` function.

The major bottleneck for maxRPWC-1 is that each time a GAC-support $\tau$ for a value $a \in D(x_j)$ is found in $rel(c_i)$, it has to check if $\tau$ can be extended to all constraints that intersect with $c_i$. This is done by iterating through all constraints that intersect with $c_i$. Assuming the intersection is on $f$ variables, in each such iteration the algorithm may check **all** the $d^{k-f}$ sub-tuples that include the assignment $\tau[var(c_i) \cap var(c_m)]$. This process is repeated each time $c_i$ is revised. To overcome this problem, for each constraint $c_i$, algorithm maxRPWC-2 keeps a set of $d^f$ pointers for every constraint $c_m$ intersecting with $c_i$. Each such pointer $lastPW_{c_i,c_m,s}$ corresponds to the sub-tuple $s$ among the $d^f$ sub-tuples for variables $var(c_i) \cap var(c_m)$. These pointers are initialized to the first combination of values for variables $var(c_m) \setminus (var(c_i) \cap var(c_m))$. During the execution of the algorithm, each pointer points to the most recently discovered valid tuple in $rel(c_m)$ that extends sub-tuple $s$.

Fig. 6 gives function `Revise` of maxRPWC-2. During each revision, for each value $a$ of $D(x_j)$ we first look for a tuple in $rel(c_i)$ that GAC-supports it, in the same way as in maxRPWC-1 (line 3). If such a tuple $\tau$ is found then the algorithm checks if $\tau$ has PW-supports in all constraints intersecting with $c_i$. For each such constraint $c_m$, maxRPWC-2 first checks whether all possible extensions of $s$, where $s$ is the sub-tuple $\tau[var(c_i) \cap var(c_m)]$, have been searched

**function** `Revise`($x_j$, $c_i$, maxRPWC)
1: **for** each value $a \in D(x_j)$
2: PW ← FALSE;
3: **for** each valid $\tau(\in rel(c_i)) \geqslant_l lastGAC_{x_j,a,c_i}$, such that $\tau[x_j] = a$
4: PW ← TRUE;
5: **for** each $c_m \neq c_i$ such that $|var(c_i) \cap var(c_m)| > 1$
6: $s \leftarrow \tau[var(c_i) \cap var(c_m)]$;
7: **if** $lastPW_{c_i,c_m,s} \neq NIL$
8: **if** $\exists$ valid $\tau'(\in rel(c_m)) \geqslant_l lastPW_{c_i,c_m,s}$
and $\tau'[var(c_i) \cap var(c_m)] = s$
9: $lastPW_{c_i,c_m,s} \leftarrow \tau'$;
10: **else**
11: $lastPW_{c_i,c_m,s} \leftarrow NIL$;
12: PW ← FALSE; **break**;
13: **else** PW ← FALSE; **break**;
14: **if** PW=TRUE, $lastGAC_{x_j,a,c_i} \leftarrow \tau$; **break**;
15: **if** PW=FALSE, remove $a$ from $D(x_j)$;
16: **return** number of deleted values;

Fig. 6. Function `Revise` of maxRPWC-2.

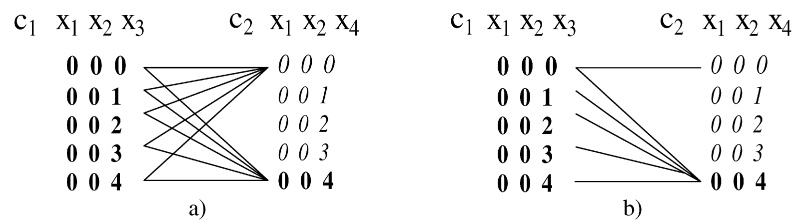


Fig. 7. Applying maxRPWC-1 and maxRPWC-2 on a non-binary problem.

before (line 7). Such a situation, indicated by $lastPW_{c_i,c_m,s}$ pointing to a virtual tuple *NIL*, means that there is no PW-support available for $\tau$, and therefore we move on to look for another GAC-support (line 13). If $lastPW_{c_i,c_m,s}$ does not point to *NIL*, the algorithm looks for a PW-support starting from $lastPW_{c_i,c_m,s}$ in the lexicographic order (line 8). If such a tuple $\tau'$ is found, $lastPW_{c_i,c_m,s}$ is updated (line 9). Otherwise, it is set to *NIL* (line 11). If no tuple is found in $rel(c_i)$ that is both a GAC-support for $a$ and has PW-supports in all intersecting constraints, then $a$ is removed from $D(x_j)$ (line 15).

The following example demonstrates the savings in constraint checks that maxRPWC-2 achieves compared to maxRPWC-1.

**Example 1.** Consider the problem of Fig. 7. There are four variables $\{x_1, \ldots, x_4\}$ and two constraints that intersect on $x_1$ and $x_2$. Tuples in bold are allowed by the constraints and are valid, while tuples in italics are not allowed by the constraints. Assume we wish to determine if the values of $x_3$ are maxRPWC. maxRPWC-1 checks all 5 tuples of $c_2$ for each value of x3, as depicted in Fig. 7a (for each tuple $\tau$ in $c_1$, all tuples of $c_2$ between the pair of edges starting from $\tau$ are checked against $\tau$). maxRPWC-2 checks all 5 tuples only for value 0 of x3. After locating tuple $\{0, 0, 4\}$ of $c_2$ as a PW-support for tuple $\{0, 0, 0\}$ of $c_1$, $lastPW_{c_1,c_2,s}$, where $s = \{0, 0\}$, points to tuple $\{0, 0, 4\}$. For the rest of x3's values, maxRPWC-2 only checks this tuple, (as depicted in Fig. 7b).

When checking if a tuple $\tau$ of a constraint $c_i$ has PW-supports, maxRPWC-2 updates $lastPW_{c_i,c_j,s}$ whenever a PW-support for $\tau$ on a constraint $c_j$ is found. This is done irrespective if $\tau$ has PW-supports in all intersecting constraints or not. For example, assume that PW-supports for $\tau$ are discovered on $q-1$ constraints and there is no PW-support on the $q$-th intersecting constraint. In this case the algorithm will stop looking for PW-supports and move on to search for a new GAC-support in the next iteration of the for loop of line 3. However, the $lastPW_{c_i,c_j,s}$ pointers for the $q-1$ constraints will not be restored. They will keep pointing to the tuples discovered when $\tau$ was checked. It is easy to see that this is correct since all tuples before $lastPW_{c_i,c_j,s}$ are either disallowed or invalid. Therefore, there is no point in restoring the pointers to their previous positions.

**Proposition 4.** *The worst-case time complexity of algorithm* maxRPWC-2 *is* $O(e^2kd^k)$.

**Proof.** When looking for a GAC-support within `Revise`, maxRPWC-2 is identical to maxRPWC-1 and therefore the two algorithms have the same cost for this part of their operation. That is, there are $O(nd)$ validity checks and $O(d^{k-1})$ visits to tuples (at cost $O(k)$) in order to make a value $a \in D(x_j)$ GAC for all calls to `Revise`($x_j$, $c_i$, maxRPWC). The difference is in the cost of extending a GAC-supporting tuple to all intersecting constraints. Once such a tuple $\tau$ is found, maxRPWC-2 iterates over the constraints that intersect with $c_i$. Assuming that $\tau[var(c_i) \cap var(c_m)] = s_m$, for any intersecting constraint $c_m$ maxRPWC-2 searches through the tuples that include assignment $s_m$. However, these tuples are not searched from scratch every time. Since the pointer $lastPW_{c_i,c_m,s_m}$ is used, the tuples that include assignment $s_m$ are searched from $lastPW_{c_i,c_m,s_m}$ since the tuples before $lastPW_{c_i,c_m,s_m}$ have already been searched. As a result, in all the calls to `Revise` on $c_i$, each tuple of each intersecting constraint is checked at most once (with each check costing $O(k)$) and there are at most $d^k$ such tuples per constraint. Thus, the cost on all calls to `Revise`($x_j$, $c_i$, maxRPWC) is $O(knd + kd^{k-1})$ for each variable-value pair, plus $O(ekd^k)$. For $kd$ values in $c_i$ this gives an asymptotic cost of $O(k^2nd^2 + k^2d^k + ekd^k) = O(k^2d^k + ekd^k)$ for sufficiently large $k$ and $d$. For $e$ constraints, the worst-case time complexity of maxRPWC-2 is $O(e^2kd^k)$ if $k < e$. □

For any constraint $c_i$ maxRPWC-2 requires $O(ekd^f)$ space to store the *lastPW* pointers of size $k$ for all constraints that intersect with $c_i$. Therefore, the space complexity of maxRPWC-2 is $O(e^2kd^f)$. This means that maxRPWC-2 is not practical for constraints of large arity sharing many variables. Note that, even when constraints share just two variables (and the space complexity is $O(e^2kd^2)$), maxRPWC is still stronger than GAC.

### *5.6. maxRPWC-3: A third algorithm for maxRPWC*

The asymptotic time complexity of maxRPWC-2 is significantly lower than that of maxRPWC-1, but to achieve it, the algorithm requires space exponential in the number of intersecting variables. Therefore, in problems where this number is large, the memory requirements can prohibit the use of maxRPWC-2. We will now describe maxRPWC-3; a third algorithm for maxRPWC that can save constraint checks compared to maxRPWC-1 (though not as many as maxRPWC-2), but only requires polynomial space.

In addition to the *lastGAC* pointers, algorithm maxRPWC-3 stores a pointer $lastPW_{x_j,a,c_i,c_m}$ for each pair of intersecting constraints $c_i$ and $c_m$ and each $a \in D(x_j)$, where $x_j \in var(c_i)$. Such a pointer is now the smallest (i.e., most recently discovered) PW-support of $lastGAC_{x_j,a,c_i}$ in $rel(c_m)$.

Fig. 8 gives function `Revise` of maxRPWC-3 (the rest of the algorithm is the same as maxRPWC-1 and maxRPWC-2). When $(x_j, a)$ is revised on $c_i$, maxRPWC-3 first checks if $lastGAC_{x_j,a,c_i}$ is valid (as do the other algorithms). If $lastGAC_{x_j,a,c_i}$ is still valid, maxRPWC-3 checks the validity of all $lastPW_{x_j,a,c_i,c_m}$, for all $c_m$ intersecting with $c_i$. If for some $c_m$, $lastPW_{x_j,a,c_i,c_m}$ is no longer valid, the algorithm searches for a new PW-support for $lastGAC_{x_j,a,c_i}$ starting from the tuple immediately after $lastPW_{x_j,a,c_i,c_m}$. In this way some constraint checks are avoided, in the spirit of maxRPWC-2. In case $lastGAC_{x_j,a,c_i}$ is no longer valid or does not have PW-support on some intersecting constraint, maxRPWC-3 looks for a new $lastGAC_{x_j,a,c_i}$, as do the other algorithms. However, in this case, for each candidate $lastGAC_{x_j,a,c_i}$ the search for a PW-support, i.e. a new $lastPW_{x_j,a,c_i,c_m}$, starts from scratch in each intersecting constraint $c_m$, in the spirit of maxRPWC-1. This is necessary in order to avoid missing any PW-supports. If no tuple is found in $rel(c_i)$ that is both a GAC-support for $a$ and has PW-supports in all intersecting constraints, then $a$ is removed from $D(x_j)$.

The following example demonstrates the savings in constraint checks that maxRPWC-3 achieves compared to maxRPWC-1.

**Example 2.** Consider the problem of Fig. 9. There are five variables $\{x_1, \ldots, x_5\}$ and two constraints that intersect on $x_1$ and $x_2$. Tuples in bold are allowed by the constraints and are valid, tuples in italics are not allowed by the constraints, and the rest of the tuples (i.e. the "deleted" ones) are allowed but not valid. Assume we wish to determine if the values of $x_4$ are maxRPWC. Initially both maxRPWC-1 and maxRPWC-3 find the GAC-supports $\{0, 0, 0, 0\}$ and $\{0, 0, 0, 1\}$ in $rel(c_1)$ for values 0 and 1 of $x_4$, respectively. Then, for each of these tuples both algorithms check

**function** `Revise`($x_j$, $c_i$, maxRPWC)
  **for** each value $a \in D(x_j)$
    PW $\leftarrow$ FALSE;
    **for** each valid $\tau(\in rel(c_i)) \geqslant_l lastGAC_{x_j,a,c_i}$, such that $\tau[x_j] = a$
      PW $\leftarrow$ TRUE;
      **for** each $c_m \neq c_i$ such that $|var(c_i) \cap var(c_m)| > 1$
        **if** $\tau = lastGAC_{x_j,a,c_i}$, $t \leftarrow lastPW_{x_j,a,c_i,c_m}$;
        **else** $t \leftarrow$ first tuple in $rel(c_m)$;
        **if** $\nexists$ valid $\tau'(\in rel(c_m)) \geqslant_l t$ such that
          $\tau[var(c_i) \cap var(c_m)] = \tau'[var(c_i) \cap var(c_m)]$
          PW $\leftarrow$ FALSE; **break**;
        **else** $lastPW_{x_j,a,c_i,c_m} \leftarrow \tau'$;
      **if** PW=TRUE $lastGAC_{x_j,a,c_i} \leftarrow \tau$; **break**;
    **if** PW=FALSE remove $a$ from $D(x_j)$;
  **return** number of deleted values;

Fig. 8. Function `Revise` of maxRPWC-3.

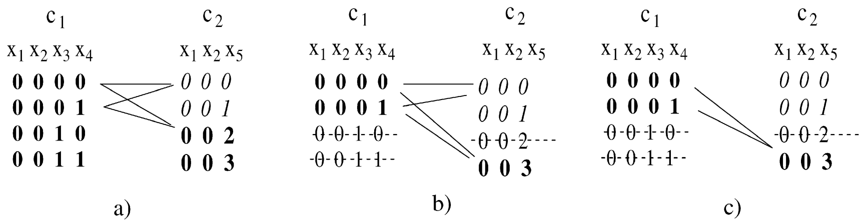


Fig. 9. Applying maxRPWC-1 and maxRPWC-3 on a non-binary problem.

3 tuples of $c_2$, as depicted in Fig. 9a, before discovering that tuple $\{0, 0, 2\}$ is a PW-support. Assume that later value 1 of $x_3$ is deleted. As a result we need to establish if the values of $x_4$ are still maxRPWC. Since the GAC-supports of $x_4$'s values discovered earlier are still valid, both maxRPWC-1 and maxRPWC-3 will try to extend them to $c_2$. Assume that tuple $\{0, 0, 2\}$ of $c_2$ is no longer valid (because value 2 of $x_5$ has been deleted). For both values of $x_4$, maxRPWC-1 will check the tuples of $c_2$ from scratch, as depicted in Fig. 9b, before discovering that tuple $\{0, 0, 3\}$ is a PW-support. On the other hand, as depicted in Fig. 9c, maxRPWC-3 will start searching immediately after tuple $\{0, 0, 2\}$ and will thus only check tuple $\{0, 0, 3\}$. This will be made possible because pointers $lastPW_{x_4,0,c_1,c_2}$ and $lastPW_{x_4,1,c_1,c_2}$ will point to tuple $\{0, 0, 2\}$.

**Proposition 5.** *The worst-case time complexity of algorithm* maxRPWC-3 *is* $O(e^2k^2d^p)$.

**Proof.** In each call to `Revise`($x_j$, $c_i$, maxRPWC) and for each $a \in D(x_j)$, maxRPWC-3 operates in an identical way to maxRPWC-1, except when $lastGAC_{x_j,a,c_i}$ is valid. In this case it may avoid some constraint checks through the use of the *lastPW* pointers. In the worst case, pointer $lastGAC_{x_j,a,c_i}$ will be assigned all $d^{k_i-1}$ extensions of $a$, which means that maxRPWC-3 will behave like maxRPWC-1. Therefore, they have the same worst-case time complexity of $O(e^2k^2d^p)$. □

The space complexity of maxRPWC-3 is determined by the space required for the *lastGAC* and *lastPW* pointers. Therefore, it is $O(ekd + e^2kd) = O(e^2kd)$ for extensional constraints and $O(ek^2d + e^2k^2d) = O(e^2k^2d)$ for intensional ones.

## 6. Experiments

To compare the efficiency of the consistencies, we ran experiments on random and benchmark configuration problems. We compared algorithms that maintain GAC2001/3.1, RPWC-1, maxRPWC-1, and rPIC-1 throughout search. For simplicity we refer to these search algorithms by the consistency they maintain. We also compared the three different algorithms for maxRPWC. In all experiments all algorithms used the dom/deg variable ordering heuristic [7] for dynamic variable ordering, and lexicographic value ordering. The experiments were run on a 3.06 GHz Pentium PC with 1 GB RAM.

### 6.1. Random problems

We first compared the different local consistencies and different algorithms for maxRPWC on randomly generated problems. Random problems allow us to relate the performance of the algorithms to certain parameters, such as tightness, constraint graph density, and domain size.

Random instances were generated using the extended *model B* [5]. According to this model, a random non-binary CSP is defined by the following five input parameters:

- $n$ number of variables;
- $d$ uniform domain size;
- $k$ uniform arity of the constraints;
- $p$ density, i.e. the ratio between the number of constraints in the problem and the number of possible constraints involving $k$ variables;

$q$ uniform looseness of the constraints, i.e. the ratio between allowed tuples and the $d^k$ total tuples of a constraint.

The constraints and the allowed tuples were generated following a uniform distribution. We made sure that the generated graphs were connected. In the following, a class of non-binary CSPs will be denoted by a tuple of the form $\langle n, d, k, p(e), q\rangle$, where $e$ denotes the actual number of constraints in the class.

*6.1.1. Comparing different consistencies*

Fig. 10 shows average CPU times and node visits from 50 instances of class $\langle 20, 10, 4, 0.004(19), q\rangle$. The value of $q$ is varied along the x-axis. On this sparse class of problems, maxRPWC is clearly the most efficient algorithm in terms of run times. RPWC and rPIC are close to the performance of GAC despite visiting many fewer nodes. rPIC in particular displays a bad CPU time per node performance as its node visits are close to those of maxRPWC but its CPU times are close to but sometimes worse than GAC.

Fig. 11 shows average CPU times and node visits from 50 instances of class $\langle 20, 20, 4, 0.004(19), q\rangle$. This is the same class as before with the difference that the domain size of the variables has been doubled. This makes the problems much harder, especially for GAC. The difference in run times between GAC and maxRPWC is now more than one order of magnitude for most values of $q$. Also, both RPWC and rPIC are more efficient than GAC, but remain significantly slower than maxRPWC.

Table 1 gives results from hard problems of five classes with different characteristics. Class 1 ($\langle 14, 8, 4, 0.1(100), 0.4\rangle$) is dense, classes 2 and 3 ($\langle 20, 10, 4, 0.04(193), 0.4\rangle$ and $\langle 15, 15, 4, 0.05(68), 0.2\rangle$) are of medium density, with class 3 having large domain sizes, class 4 ($\langle 50, 5, 4, 0.0002(46), 0.185\rangle$) is sparse and has small domain sizes, and class 5 ($\langle 30, 15, 4, 0.001(27), 0.05\rangle$) is sparse. GAC is the most efficient algorithm in classes 1 and 4, it is competitive with maxRPWC in class 2, and it is inefficient in classes 3 and 5. RPWC is worse than maxRPWC in all five classes and worse than GAC in all but classes 3 and 5. rPIC displays very bad performance in all but the last class where it is significantly better than GAC and RPWC but worse than maxRPWC. Finally, maxRPWC is worse than GAC in

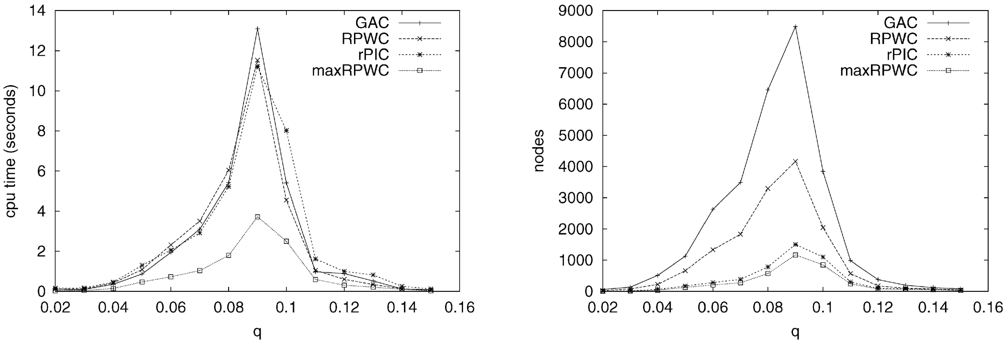


Fig. 10. CPU times (left) and node visits (right) on class $\langle 20, 10, 4, 0.004(19), q\rangle$.

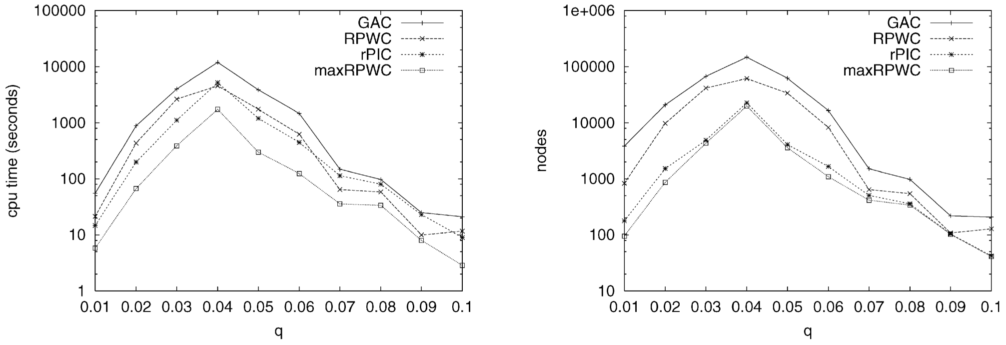


Fig. 11. CPU times (left) and node visits (right) on class $\langle 20, 20, 4, 0.004(19), q\rangle$.

Table 1
Results on four classes of random problems. CPU times and node visits are averages over 50 hard instances for each class

| Class | GAC nodes–time | RPWC nodes–time | rPIC nodes–time | maxRPWC nodes–time |
|---|---|---|---|---|
| 1: $\langle 14, 8, 4, 0.1(100), 0.4\rangle$ | 1298–4.64 | 672–7.70 | 515–41.11 | 114–7.80 |
| 2: $\langle 20, 10, 4, 0.04(193), 0.4\rangle$ | 16,883–421.83 | 9380–532.89 | 5335–5,822.93 | 1433–409.14 |
| 3: $\langle 15, 15, 4, 0.05(68), 0.2\rangle$ | 14,560–1,615.21 | 7276–707.69 | 3589–3,750.32 | 1799–321.94 |
| 4: $\langle 50, 5, 4, 0.0002(46), 0.185\rangle$ | 14,292–1.76 | 8700–6.93 | 4553–14.74 | 4342–3.81 |
| 5: $\langle 30, 15, 4, 0.001(27), 0.05\rangle$ | 534,899–8,560.73 | 175,404–3,137.56 | 12,454–616.54 | 10,134–283.25 |

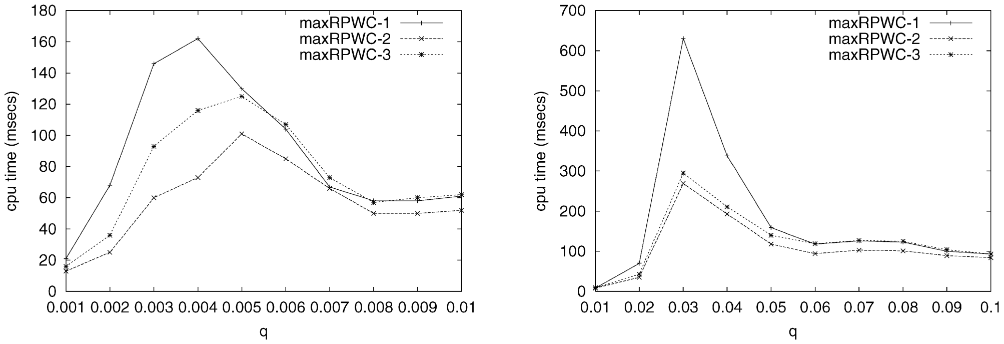


Fig. 12. CPU times of the three maxRPWC algorithms on classes $\langle 30, 20, 4, 0.001(27), q\rangle$ (left) and $\langle 50, 10, 4, 0.001(230), q\rangle$ (right).

classes 1 and 4, slightly better in class 2, five times better in class 3, and more than one order of magnitude better in class 4.

From these experiments and also experiments with other parameters, we conjecture that maxRPWC is more efficient than GAC on sparse problems, especially when the domain size is large. On the other hand, maxRPWC is too expensive on problems of relatively high density or when the domain sizes are very small. Although it always reduces the visited nodes, in such problems it is outperformed by GAC in CPU time. In problems of medium density there is no clear winner. When domain sizes are large maxRPWC outperforms GAC whereas when domain sizes are small the opposite holds. RPWC is outperformed by maxRPWC in almost all parameter settings. However, it is better than GAC in sparse problems with relatively large domain sizes. Finally, rPIC is always worse than maxRPWC and displays bad performance in all but the very sparse classes.

The reason for rPIC's bad performance is the amount of redundant constraint checks it performs at each node. Recall that in order to establish that a value $a$ of a variable $x$ in a constraint $c$ is rPIC, for each constraint $c'$ intersecting $c$ algorithm rPIC-1 searches for a tuple in $c$ that GAC-supports $a$ and can be extended to a tuple in $c'$. Since this process is repeated for each intersecting constraint, it may involve repeated searches for GAC-supports in $c$. So, although for each pair $c$,$c'$ each tuple of $c$ will be checked only once for GAC-support, overall it may end up being checked many times.

### 6.1.2. *Comparing algorithms for maxRPWC*

Fig. 12 shows average CPU times (in msecs) of the three maxRPWC algorithms on two classes of random problems. The three algorithms were run stand-alone to preprocess the generated instances. Algorithm maxRPWC-2 is the fastest of the three, followed by maxRPWC-3. Both these algorithms are considerably faster than maxRPWC-1 (up to 2.5 times) for small values of $q$ where problems are either not maxRPWC or the application of maxRPWC results in many value deletions. As $q$ grows and maxRPWC results in no deletions, i.e. it becomes useless for preprocessing, the performance of the three algorithms evens out.

Fig. 13 shows average CPU times when maintaining the three algorithms during search on problems of class $\langle 20, 10, 4, 0.004(19), q\rangle$. As we can see, again maxRPWC-2 and maxRPWC-3 are faster than maxRPWC-1, but the differences are now marginal. This is due to the cost of updating the data structures used by maxRPWC-2 and maxRPWC-3 upon backtracking. Similar results were observed with other problem classes.

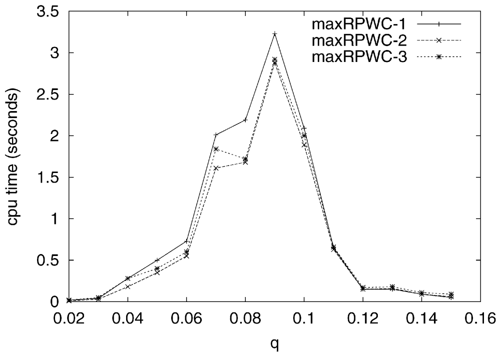


Fig. 13. CPU times when maintaining the three maxRPWC algorithms on class $\langle 20, 10, 4, 0.004(19), q\rangle$.

### 6.2. Configuration

We ran experiments on some problems from the CLib configuration benchmark library (see www.itu.dk/doi/VeCoS/clib). Since these problems are very easy (when looking for one solution), each problem was slightly altered by adding a few variables (5–6) and constraints (8–10) randomly in the following way. The added constraints were of arity 3 or 4 (chosen at random) and the variables on which they were posted were selected at random, making sure that the resulting graph was connected. The looseness of each added constraint was also set at random, and finally, the allowed tuples of each constraint were chosen at random according to its looseness. In this way we were able to obtain a large number of hard instances.

Table 2 gives results for algorithms that maintain GAC, RPWC, rPIC, and maxRPWC during search. The corresponding algorithms for each local consistency are again GAC2001/3.1, RPWC-1, rPIC-1 and maxRPWC-1.

Algorithm maxRPWC is considerably more efficient than all the other ones in these problems. rPIC and RPWC are also faster than GAC, in some cases by a large margin. This is due to the additional pruning they perform (as clearly demonstrated by the numbers of node visits).

Apart from the problems of Table 2, we also ran experiments on 50 instances created following the same methodology as above and using the Renault configuration problem as basis. This is a problem with 101 variables and 134 constraints, where the largest domain has 42 values and there are constraints with arity as high as 10. The instances created from the Renault problem are very hard. For example, the second instance could not be solved by GAC within four days of CPU time. We set a time limit of 20 minutes within which GAC solved 10 instances. Algorithms RPWC, rPIC and maxRPWC solved 15, 16 and 27 instances respectively. Note that the second instance was solved within the time limit only by maxRPWC (in a few seconds). The results from the Renault problem are summarized in Table 3.

There were 7 instances (all soluble) that were solved by all algorithms within the time limit and 21 instances on which all algorithms timed out. On the 7 instances solved by all algorithms, RPWC displayed by far the best performance, as shown in Table 3. There were 2 instances which GAC and RPWC were able to solve whereas maxRPWC and rPIC timed out. Also, there was 1 instance which GAC, RPWC, and maxRPWC were able to solve whereas rPIC timed out. From the remaining 19 instances all were solved by maxRPWC, 5 by RPWC, 9 by rPIC, and none by GAC. Overall, maxRPWC was the most efficient algorithm in 17 instances and RPWC in 12.

Table 2
Configuration problems. $n$ and $e$ are the numbers of variables and constraints in the modified instances. $k$ and $d$ are the maximum arity and domain size respectively. Run times are in seconds. Averages are over 50 hard instances created from each benchmark

| Problem | $n$ | $e$ | $k$ | $d$ | GAC nodes–time | RPWC nodes–time | rPIC nodes–time | maxRPWC nodes–time |
|---|---|---|---|---|---|---|---|---|
| machine | 30 | 22 | 4 | 9 | 535,874–10.27 | 95,459–4.63 | 36,046–6.48 | 8263–0.84 |
| fx | 24 | 21 | 5 | 44 | 193,372–3.43 | 54,915–1.65 | 17,755–2.18 | 864–0.06 |
| fs | 29 | 18 | 6 | 51 | 618,654–10.65 | 150,544–4.11 | 1443–0.31 | 19–0.06 |
| esvs | 33 | 20 | 5 | 61 | 6,179,966–124.39 | 333,036–10.78 | 32,548–3.79 | 2612–0.07 |

Table 3
Results from the Renault problem. Nodes and CPU times (in seconds) are averages are over the 7 instances solved by all algorithms. The last column gives the number of instances on which each algorithm was the most efficient of the four

| Algorithm | Nodes | Time | # instances solved | # instances solved fastest |
|---|---|---|---|---|
| GAC | 583,355 | 146.47 | 10 | 0 |
| RPWC | 462 | 28.97 | 15 | 12 |
| rPIC | 456 | 189.28 | 16 | 0 |
| maxRPWC | 214 | 127.28 | 27 | 17 |

## 7. Conclusion

Although domain filtering consistencies tend to be more practical than consistencies that change the constraint relations and the constraint graph, very few such consistencies have been proposed for non-binary constraints. In this paper, we performed a detailed theoretical, algorithmic, and empirical study of three such consistencies, RPWC, rPIC and maxRPWC. In our theoretical study we showed that the pruning power of these consistencies on non-binary problems lays between PWC + GAC and GAC, while rPIC and maxRPWC are incomparable to SGAC. Some surprising results were also revealed. For example, rPIC and maxRPWC are weaker than RPC when restricted to binary constraints.

We also described algorithms that can be used to achieve RPWC, maxRPWC and rPIC. In addition we proposed two alternative algorithms for maxRPWC, which is the most efficient among the three consistencies. One of them has a particularly good time complexity, competitive with GAC algorithms, though with a higher space cost. Experiments demonstrated the potential of the new consistencies, and especially maxRPWC and RPWC, as an alternative or complement to GAC. As future work, we will investigate ways to combine inverse consistencies with specialized GAC propagators. Also, we intend to further develop some of the algorithms presented here. Finally, new domain filtering consistencies for non-binary constraints can be developed and studied.

## Acknowledgements

We would like to thank the anonymous reviewers for their very useful comments and suggestions that helped improve this paper.

## References

[1] K.R. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.
[2] F. Bacchus, P. Chen, X. van Beek, T. Walsh, Binary vs. non-binary CSPs, Artif. Intell. 140 (2002) 1–37.
[3] P. Berlandier, Improving domain filtering using restricted path consistency, in: Proceedings of IEEE CAIA-95, 1995, pp. 32–37.
[4] C. Bessiere, Constraint propagation, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006 (Chapter 3).
[5] C. Bessière, P. Meseguer, E.C. Freuder, J. Larrosa, On forward checking for non-binary constraint satisfaction, Artif. Intell. 141 (2002) 205–224.
[6] C. Bessière, J.C. Régin, Arc consistency for general constraint networks: Preliminary results, in: Proceedings of IJCAI'97, 1996, pp. 398–404.
[7] C. Bessière, J.C. Régin, MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems, in: Proceedings of CP'96, 1996, pp. 61–75.
[8] C. Bessière, J.C. Régin, R. Yap, Y. Zhang, An optimal coarse-grained arc consistency algorithm, Artif. Intell. 165 (2) (2005) 165–185.
[9] R. Debruyne, C. Bessière, From restricted path consistency to max-restricted path consistency, in: Proceedings of CP-97, 1997, pp. 312–326.
[10] R. Debruyne, C. Bessière, Domain filtering consistencies, J. Artif. Intell. Res. 14 (2001) 205–230.
[11] R. Dechter, P. van Beek, Local and global relational consistency, Theoret. Comput. Sci. 173 (1997) 283–308.
[12] E. Freuder, A sufficient condition for backtrack-bounded search, JACM 32 (4) (1985) 755–761.
[13] E. Freuder, C. Elfe, Neighborhood inverse consistency preprocessing, in: Proceedings of AAAI'96, 1996, pp. 202–208.
[14] M. Gyssens, On the complexity of join dependencies, ACM Trans. Database Syst. 11 (1) (1986) 81–108.
[15] P. Janssen, P. Jégou, B. Nouguier, M.C. 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, 1989, pp. 420–427.
[16] 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).
[17] P. Jégou, On the consistency of general constraint satisfaction problems, in: Proceedings of AAAI'93, 1993, pp. 114–119.
[18] A. Mackworth, Consistency in networks of relations, in: Artificial Intelligence, 1977, pp. 99–118.

[19] A.K. Mackworth, On reading sketch maps, in: Proceedings IJCAI'77, 1977, pp. 598–606.
[20] N. Mamoulis, K. Stergiou, Solving non-binary CSPs using the hidden variable encoding, in: Proceedings of CP-2001, 2001, pp. 168–182.
[21] K. Stergiou, T. Walsh, Encodings of non-binary constraint satisfaction problems, in: Proceedings of AAAI'99, 1999, pp. 163–168.
[22] P. van Beek, R. Dechter, On the minimality and global consistency of row-convex constraint networks, JACM 42 (3) (1995) 543–561.
[23] G. Verfaillie, D. Martinez, C. Bessière, A generic customizable framework for inverse local consistency, in: Proceedings of AAAI'99, 1999, pp. 169–174.