Inverse Consistencies for Non-binary Constraints

Kostas Stergiou[1] and **Toby Walsh**[2]

Abstract. We present a detailed study of two inverse consistencies for non-binary constraints: relational path inverse consistency (rel PIC) and pairwise inverse consistency (PWIC). These are stronger than generalized arc consistency (GAC), even though they also only prune domain values. We propose algorithms to achieve rel PIC and PWIC, that have a time complexity better than the previous generic algorithm for inverse consistencies. One of our algorithms for PWIC has a complexity comparable to that for GAC despite doing more pruning. Our experiments demonstrate that inverse consistencies can be more efficient than GAC on a range of non-binary problems.

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. Local consistency techniques that only filter domains like (generalized) arc consistency (GAC) tend to be more practical than those that alter the structure of the constraint graph 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 [8, 6, 13]. The situation is very different for non-binary constraints. A number of consistencies that are stronger than GAC have been developed, including relational consistency [12] and pairwise consistency [9]. However, these are typically not domain filtering.

We study here two promising domain filtering techniques for non-binary constraints: relational path inverse consistency (rel PIC) and, the stronger, pairwise inverse consistency (PWIC). Relational consistencies [12] have rarely been used in practice as most are not domain filtering, and have high time complexities. Pairwise consistency [9] (also called inter-consistency [10]) has also rarely been used as it is not domain filtering and the algorithm proposed in [9] has a high time complexity and requires all constraints to be extensional. Rel PIC and PWIC are local consistency techniques that do not suffer from these problems. They are domain filtering and are not prohibitively expensive to enforce. Our theoretical analysis reveals some surprises. For example, when restricted to binary constraints, rel PIC does not reduce to the variable based definition of path inverse consistency.

We propose algorithms to enforce rel PIC and PWIC that can be applied to constraints intentionally or extensionally specified. Their time complexity is better than the complexity of the generic algorithm for inverse consistencies given in [13]. The time complexity of the second algorithm for PWIC is $O(e^2k^2d^k)$ where e is the number of constraints, k is the maximum arity and d is the maximum domain size. This is significantly lower than the complexity of the generic algorithm, and is comparable to the complexity of standard GAC algorithms, like GAC-Schema ($O(ekd^k)$) [3] and GAC2001/3.1 ($O(ek^2d^k)$) [5]. However, this improvement comes at a cost as the space required is exponential in the number of *shared* variables. Experimental results demonstrate that it is feasible to maintain rel PIC and PWIC during search, and they can be more cost-effective than GAC on certain problems.

[1] Department of Information & Communication Systems Engineering University of the Aegean, Greece.

[2] National ICT Australia & University of New South Wales, Australia.

2 BACKGROUND

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple (X, D, C) where: X is a set of n variables, D is a set of domains, and C is a set of e constraints. Each constraint c_i is a pair $(var(c_i), rel(c_i))$, where $var(c_i) = \{x_{j_1}, \ldots, x_{j_k}\}$ is an ordered subset of X, and $rel(c_i)$ is a subset of the *Cartesian* product $D(x_{j_1})\text{x}\ldots\text{x}D(x_{j_k})$. A tuple $\tau \in rel(c_i)$ is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable. 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 assume that the number of variables that any two constraints have in common (denoted by f) is uniform. This assumption is made to simplify the description and complexity analysis of the algorithms, and can be easily lifted.

The assignment of value a to variable x_i is denoted by (x_i, a). The set of variables over which a tuple τ is defined is $var(\tau)$. For any subset var' of $var(\tau)$, $\tau[var']$ is the sub-tuple of τ that includes only assignments to the variables in var'. Any two tuples τ and τ' of $rel(c_i)$ can be lexicographically ordered. In this ordering, $\tau <_l \tau'$ iff there a exists a subset $\{x_1, \ldots, x_j\}$ of $var(c_i)$ such that $\tau[x_1, \ldots, x_j] = \tau'[x_1, \ldots, x_j]$ and $\tau[x_{j+1}] <_l \tau'[x_{j+1}]$. An assignment τ 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* is a consistent assignment to all variables. A value $a \in D(x_j)$ is consistent with a constraint c_i, where $x_j \in var(c_i)$, iff $\exists \tau \in rel(c_i)$ such that $\tau[x_j] = a$ and τ is valid. In this case, we say that τ is a GAC-support of a in c_i. A constraint c_i is *Generalized Arc Consistent* (GAC) iff $\forall\, x_j \in var(c_i)$, $\forall\, a \in D(x_j)$, there exists a GAC-support for a in c_i. A problem is GAC iff domains are non-empty and all constraints are GAC.

A binary problem is (i, j) *consistent* iff it has non-empty domains and any consistent instantiation of i variables can be extended to a consistent instantiation involving j additional variables [7]. A problem is *strong* (i, j)*-consistent* iff it is (k, j) consistent for all $k \leq i$. A problem is *arc consistent* (AC) iff it is $(1, 1)$-consistent. A problem is (strong) *path consistent* (PC) iff it is (strong) $(2, 1)$-consistent. A problem is *(strong)* m*-consistent* iff it is (strong) $(m, 1)$-consistent. A problem is *path inverse consistent* (PIC) iff it is $(1, 2)$-consistent.

A problem is *relationally arc consistent* (rel AC) iff any consistent instantiation for all but one of the variables in a constraint can be extended to the final variable so as to satisfy the constraint [12]. A problem is *strongly relationally arc consistent* (strong rel AC) iff any consistent instantiation of a subset of the variables in a constraint can be extended to all the variables in the constraint. A problem is *relationally* (i, m)*-consistent* iff any consistent instantiation for i of the variables in a set of m constraints can be extended to all the variables in the set. We can construct singleton versions of all the relational consistencies in a straightforward manner [6]. For example, a problem is *relationally singleton arc consistent* (rel SAC) iff it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made relationally AC. A problem is *pairwise consistent* (PWC) iff it has non-empty relations and any consistent tuple in a constraint c can be consistently extended to any other constraint that intersects with c [9]. As shown in [9], applying PWC to a non-binary CSP is equivalent to applying AC to the dual encoding.

Following [6], we call a consistency property A stronger than B iff in any problem in which A holds then B holds, and strictly stronger (written $A \rightarrow B$) iff it is stronger and there is at least one problem in which B holds but A does not. We call a local consistency property A incomparable with B (written $A \otimes B$) iff A is not stronger than B nor vice versa. Finally, we call a local consistency property A equivalent to B (written $A \leftrightarrow B$) iff A is stronger than B and vice versa.

3 INVERSE CONSISTENCIES

In practice, most strong local consistency techniques have prohibitive space and time complexities. One way around this problem is to use inverse consistencies [8]. These require limited space as they only prune domains. When an inverse local consistency is enforced, it removes from the domain of a variable the values that cannot be consistently extended to some additional variables. For example, when enforcing PIC we remove values that cannot be consistently extended to any set of two other variables. By analogy with the definition of PIC, relational $(1, 2)$ consistency is called *relational path inverse consistency* (rel PIC). We also define *pairwise inverse consistency* (PWIC), an inverse version of PWC. A value $a \in D(x_j)$ is PWIC iff $\forall$ constraints c_i, where $x_j \in var(c_i)$, $\exists \tau \in rel(c_i)$ such that $\tau[x_j] = a$ and τ is valid **and** $\forall c_l$, s.t. $var(c_i) \cap var(c_l) \neq \emptyset, \exists \tau' \in rel(c_l)$, s.t. $\tau[var(c_i) \cap var(c_j)] = \tau'[var(c_i) \cap var(c_j)]$ and τ' is valid. In this case we say that τ' is a PW-support of τ. A constraint c_i is PWIC iff $\forall\, x_j \in var(c_i), \forall\, a \in D(x_j)$, a is PWIC. A problem is PWIC iff domains are non-empty and all constraints are PWIC.

We first compare rel PIC and PWIC with GAC, the most popular local consistency for non-binary constraints.

Theorem 1 *PWIC → rel PIC → GAC*

Proof: By definition, PWIC is stronger than rel PIC which is stronger than GAC. To show strictness, consider: *alldiff*(x_1, x_2, x_3) and $x_1 = x_2$. If domains are $\{0, 1, 2\}$ then the problem is GAC but is not rel PIC. Now consider a constraint c_1 over variables x_1, x_2, x_3 and two other constraints c_2 and c_3 involving x_2 and x_3 (and other variables). Assume that value 0 of x_1 can be extended to tuples $(0, 0, 0)$ and $(0, 1, 1)$ in c_1 but only the first tuple can be extended to a consistent tuple in c_2, while only the second tuple can be extended to a consistent tuple in c_3. The problem is rel PIC but not PWIC. □

Not surprisingly, when all constraints intersect on at most one variable, PWIC and rel PIC collapse down to GAC.

Theorem 2 *On constraints that intersect on at most one variable: PWIC ↔ rel PIC ↔ GAC*

Proof: Suppose that the constraints intersect on at most one variable and are GAC. Consider an assignment for a variable and the set of constraints involving this variable. Take one of these constraints. As this is GAC, we can find a satisfying tuple including this assignment. Consider all the intersecting constraints and the value of the intersection variable in our tuple. As all these constraints are GAC, we can extend the tuple to satisfy them. Hence, we can extend any assignment to a tuple that can be extended to satisfy all intersecting constraints. The problem is thus PWIC (and hence rel PIC). □

When restricted to binary constraints, PWIC collapses down to rel PIC. We might expect rel PIC to reduce to the corresponding variable based definition. Surprisingly, this is not the case.

Theorem 3 *On binary constraints:*

$$\begin{array}{ccccc}
\textit{PWIC} & \leftrightarrow & \textit{rel PIC} & \rightarrow & \textit{rel AC} \\
 & & \uparrow & & \updownarrow \\
 & & \textit{PIC} & \rightarrow & \textit{AC}
\end{array}$$

Proof: We first show PWIC ↔ rel PIC. Since binary constraints intersect on at most one variable unless they involve the same two variables, consider such a case with two variables x_1, x_2, where all constraints are rel PIC. Take any two constraints. Since the constraints are rel PIC, any instantiation (x_1, a) can be extended to an instantiation (x_2, b) that satisfies both constraints. Since this holds for all pairs of constraints that involve x_1 and x_2, these instantiations are consistent with all such constraints. Therefore, the problem is PWIC.

To show PIC → rel PIC, consider the constraints: $x_1 \neq x_2, x_1 \neq x_3$ and $x_2 \neq x_3$. If all variables are 0/1 then the problem is rel PIC but not PIC. The other relations follow in a straightforward way. □

Non-binary constraints can sometimes be decomposed into binary constraints on the same variables. For example, an all-different constraint can be decomposed into binary not-equals constraints. On such constraints, we can directly compare relational consistencies with binary consistencies on the decomposition.

Theorem 4 *On decomposable non-binary constraints:*

$$\begin{array}{ccccc}
 & & \textit{strong PC} & & \\
 & \otimes & & \otimes & \\
\textit{PWIC} & \rightarrow & \textit{rel PIC} & \rightarrow & \textit{AC} \\
\otimes\ \otimes & & & \otimes\ \otimes & \\
\textit{SAC PIC} & & & \textit{SAC PIC} &
\end{array}$$

Proof: To show rel PIC → AC on the decomposition, consider a problem that is rel PIC. Given any two constraints, any instantiation for a variable can be extended to all the variables in the two constraints. Hence, it can be extended to at least one other variable. Thus, the decomposition is AC. To show strictness, consider the constraints: *alldiff*(x_1, x_2, x_3) and $x_3 = x_4$. If all variables are 0/1, then the problem is not rel PIC but its decomposition is AC.

To show the other relationships, consider the constraints: *alldiff*(x_1, x_2, x_3, x_4) and $x_4 = x_5$. If domains are $\{0, 1, 2\}$, then the problem is not rel PIC (and hence not PWIC). However, its decomposition is strong PC (and thus SAC and PIC). For the reverse, consider the constraints: $x_1 \neq x_2$, $x_1 \neq x_3$, and $x_2 \neq x_3$. If all variables are 0/1, then the problem is rel PIC and PWIC, but its decomposition is not PIC (and hence not SAC nor strong PC). □

One way to deal with non-binary constraints is to encode them into binary ones, and apply binary techniques (as for example in [1]). We now position rel PIC and PWIC with respect to other relational consistencies and consistencies enforced in the hidden variable and dual encodings of a non-binary problem. 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 5 *On (non-binary) constraints which are GAC:*

$$\begin{array}{ccccc}
\textit{SAC}_{\text{hidden}} & \rightarrow & \textit{PIC}_{\text{hidden}} & \leftrightarrow & \textit{AC}_{\text{hidden}} \\
\downarrow & & \uparrow & & \\
\textit{rel SAC} & \rightarrow & \textit{rel PIC} & \otimes & \textit{(strong) rel AC} \\
\uparrow & & \uparrow & & \otimes \\
\textit{SAC}_{\text{dual}} & \rightarrow & \textit{PIC}_{\text{dual}} & \rightarrow & \textit{AC}_{\text{dual}}\ \textit{(PWC)} \leftrightarrow \textit{PWIC}
\end{array}$$

Proof: Due to space limitations we only give proofs of $\text{AC}_{\text{dual}} \leftrightarrow$ PWIC, $\text{PIC}_{\text{dual}} \rightarrow$ rel PIC and rel PIC $\otimes$ (strong) rel AC.

To compare PWIC to AC_{dual} (and PWC), we only consider the values that are pruned by these consistencies. For AC_{dual} (and PWC), which delete tuples from constraints, this can be done if GAC is applied as a second step. In this case, a value is deleted iff all the tuples in some dual variable x, that include that value, are deleted. A tuple is deleted iff it has no support in some other dual variable (i.e. it is not pairwise consistent). In the non-binary problem, PWIC deletes the same values because it finds that all their extensions (i.e. tuples) in the constraint corresponding to x are not pairwise consistent. Therefore PWIC achieves the same pruning as AC_{dual}.

To show $\text{PIC}_{\text{dual}} \rightarrow$ rel PIC, consider a problem whose dual encoding is PIC. Take any pair of constraints. There are two cases. In the first case, the two constraints are disjoint. As each constraint is GAC, we can extend any assignment for a variable in one of the constraints to satisfy all the variables in both constraints. In the second case, the two constraints overlap. As each constraint is GAC, we can extend any assignment for a variable in one of the constraints to satisfy all the variables in the constraint. As the dual encoding is PIC, we can extend this assignment for the dual variable associated with this constraint, to the dual variable associated with the second constraint, and any other third dual variable. That is, we can extend the assignment to satisfy all the variable in both the constraints. In both cases, the problem is rel PIC. To show strictness, consider the constraints: $x_1 \neq x_2, x_1 \neq x_3$ and $x_2 \neq x_3$. If all variables are 0/1 then the dual encoding is not PIC, whilst the original problem is rel PIC.

To show rel PIC $\otimes$ (strong) rel AC, consider the constraints: *alldiff*(x_1, x_2, x_3) and $x_1 = x_2$. If domains are $\{0, 1, 2\}$ then the problem is (strong) rel AC but is not rel PIC. For the reverse, consider *alldiff*(x_1, x_2, x_3). If domains are $\{0, 1, 2\}$, then the problem is not (strong) rel AC, but it is rel PIC. $\square$

One surprise here is that rel PIC is incomparable to (strong) rel AC whilst the binary local consistency PIC is strictly stronger than AC.

4 ALGORITHMS FOR PWIC

The generic AC-7 based algorithm for inverse local consistencies proposed in [13] can easily be adapted to enforce rel PIC. However, adapting the generic algorithm to enforce PWIC is more involved. The time and space complexities of this algorithm for rel PIC are $O(e^2kd^{2k})$ and $O(e^2k^2d)$ respectively, while for PWIC a naive adaptation would result in a very inefficient algorithm, in terms of time complexity. The time complexity of the PWC algorithm given in [9] is $O(e^2kd^{2k})$. This algorithm enforces PWC by applying AC in the dual encoding of the problem. Since PWC does not prune values from domains but instead deletes tuples from constraint relations, domains can be pruned if GAC is applied as a second step [9]. Apart from its high time complexity, this pruning method also suffers from a high space complexity, since constraints need to be extensionally represented. The time complexity of PWC can be reduced to $O(e^3d^k)$ if the algorithm of [11] for AC in the dual encoding is used. However, the space complexity can still be prohibitive.

In what follows we describe two algorithms for PWIC, which are stronger and more efficient than rel PIC, with better time complexity than a generic algorithm. These algorithms can be easily modified to achieve rel PIC. Our algorithms achieve the same pruning as a two-step procedure of PWC followed by GAC with better time and space complexity.

4.1 PWIC-1: A Simple Algorithm for PWIC

From the definition of PWIC 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 has a PW-support in all constraints intersecting with c_i. Figure 1 gives `PWIC-1`, an algorithm for PWIC based on the GAC algorithm GAC2001/3.1 [5].

function `PWIC-1`
1: $Q \leftarrow \emptyset$;
2: **for** each constraint c_i
3: **for** each variable x_j where $x_j \in var(c_i)$
4: **if** `Revise`$(x_j, c_i) > 0$
5: **if** $D(x_j)$ is empty **return** INCONSISTENCY;
6: put in Q each constraint c_m such that $x_j \in var(c_m)$;
7: **return** `Propagation`;
function `Propagation`
8: **while** Q is not empty
9: pop constraint c_i from Q;
10: **for** each unassigned variable x_j where $x_j \in var(c_i)$
11: **if** $Revise(x_j, c_i) > 0$
12: **if** $D(x_j)$ is empty **return** INCONSISTENCY;
13: put in Q each $c_m \neq c_i$ such that $x_j \in var(c_m)$;
14: **return** CONSISTENCY;
function `Revise`(x_j, c_i)
15: **for** each value $a \in D(x_j)$
16: PW $\leftarrow$ FALSE;
17: **for** each valid $\tau(\in\ rel(c_i)) \geq_l lastGAC_{x_j,a,c_i}$ such that $\tau[x_j] = a$
18: PW $\leftarrow$ TRUE;
19: **for** each c_m such that $var(c_i) \cap var(c_m) > 1$
20: **if** $\nexists$ valid $\tau'(\in\ rel(c_m))$ such that $\tau[var(c_i) \cap var(c_m)] = \tau'[var(c_i) \cap var(c_m)]$
21: PW $\leftarrow$ FALSE; **break**;
22: **if** PW=TRUE $lastGAC_{x_j,a,c_i} \leftarrow \tau$; **break**;
23: **if** PW=FALSE remove a from $D(x_j)$;
24: **return** number of deleted values;

Figure 1. A simple algorithm for PWIC.

Algorithm `PWIC-1` uses a stack (or queue) of constraints to propagate value deletions, and works as follows. In the initial phase it iterates over each constraint c_i (line 2) and updates the domain of every variable x_j involved in c_i by calling `Revise` (line 4). 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. As in GAC2001/3.1, we store a pointer $lastGAC_{x_j,a,c_i}$. This is now the most recently discovered tuple in $rel(c_i)$ that GAC-supports value a of variable x_j **and** is pairwise consistent. 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" $lastGAC_{x_j,a,c_i}$ (line 17). 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 τ) is pairwise consistent. 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 τ has PW-supports, `PWIC-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 PWIC offers here no more pruning than GAC. It checks if there is a tuple $\tau' \in rel(c_m)$ such that τ' is a PW-support of τ (lines

19-20). If such tuples are found for all intersecting constraints then $lastGAC_{x_j,a,c_i}$ is updated (line 22). If no PW-support is found on some intersecting constraint, then the iteration stops (line 21) and the algorithm looks for a new GAC-support. If no pairwise consistent GAC-support is found, a is removed from $D(x_j)$ (line 23). In this case, all constraints involving $D(x_j)$ not already in the stack are put on the stack (line 6). Constraints are then removed from the stack sequentially and their variables revised. The algorithm terminates if a domain is wiped out, in which case the problem is not PWIC, or if the stack becomes empty, in which case the problem is PWIC.

Proposition 1 *The worst-case time complexity of algorithm* `PWIC-1` *is* $O(e^2k^3d^{2k-f})$.

Proof: The complexity is determined by the revise function. `Revise`(x_j, c_i) can be called at most kd times for each constraint c_i; once for every deletion of a value from $D(x_j)$, where x_j is one of the k variables in $var(c_i)$. In each call, the algorithm performs one check to see if $lastGAC_{x_j,a,c_i}$ is valid. If $lastGAC_{x_j,a,c_i}$ is not valid, it tries to find a new GAC-support for a in $rel(c_i)$. The cost to make a GAC for the total number of calls to $Revise(x_j, c_i)$ is $O(kd^{k-1})$ (see [5] for details).

For each GAC-support τ found, `PWIC-1` iterates over the, at most e, constraints that intersect with c_i to determine if τ has PW-supports. For each such constraint c_m, it checks at most d^{k-f} tuples, i.e. those that take the same values in variables $var(c_i) \cap var(c_m)$ as in τ. The cost of each such check is $O(k - f)$ if we assume this is linear in the arity of the tuple. Therefore, for each value of x_j, $Revise$ makes $O(kd^{k-1} \times e(k - f)d^{k-f})$ checks. For kd values, the upper bound in checks performed to make one constraint PWIC is $O(ek^3d^{2k-f})$. For e constraints the complexity is $O(e^2k^3d^{2k-f})$. □

The space complexity of `PWIC-1` is $O(ekd)$.

4.2 PWIC-2: An Improved Algorithm for PWIC

Although the asymptotic time complexity of `PWIC-1` is lower than that of the generic algorithm of [13], it can still be prohibitive in practice. (`PWIC-2`) offers a significant improvement in terms of time complexity, but with an increase in the space complexity. The major bottleneck for `PWIC-1` is that each time a GAC-support τ for a value $a \in D(x_j)$ is found in $rel(c_i)$, it has to check if τ is pairwise consistent. This is done by iterating through all constraints that intersect with c_i. 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 `PWIC-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 s-th sub-tuple among the d^f sub-tuples containing the possible combinations of values for variables $var(c_i) \cap var(c_m)$. Each pointer points to the most recently discovered tuple in $rel(c_m)$ that extends sub-tuple s and is a PW-support for the current GAC-support of a value (x_j, a) on c_i.

Figure 2 gives lines 20-21 of function `Revise` of `PWIC-2` (the rest of the algorithm is the same as `PWIC-1`). 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 `PWIC-1`. If such a tuple τ is found then the algorithm checks if τ is pairwise consistent. For each constraint c_m that intersects with c_i `PWIC-2` first checks if tuple $lastPW_{c_i,c_m,s}$ is valid, where s is the sub-tuple $\tau[var(c_i) \cap var(c_m)]$. If it is valid then it PW-supports τ since they have the same values for the variables $var(c_i) \cap var(c_m)$. Otherwise, the algorithm looks for a new PW-support starting from the tuple that

function `Revise`(x_j, c_i)
  $s \leftarrow \tau[var(c_i) \cap var(c_m)]$;
  if $lastPW_{c_i,c_m,s}$ is not valid
    if $\exists$ valid $\tau'(\in rel(c_m)) >_l lastPW_{c_i,c_m,s}$
    and $\tau'[var(c_i) \cap var(c_m)] = s$
      $lastPW_{c_i,c_m,s} \leftarrow \tau'$;
    else PW $\leftarrow$ FALSE; **break**;

Figure 2. Function `Revise` of `PWIC-2`.

includes the assignment s and is immediately "after" $lastPW_{c_i,c_m,s}$ in the lexicographic order. If such a tuple τ' is found, $lastPW_{c_i,c_m,s}$ is updated. If no tuple is found in $rel(c_i)$ that is both a GAC-support for a and is pairwise conistent, then a is removed from $D(x_j)$.

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

Example 1 Consider the problem of Figure 3. There are four variables $\{x_1, \ldots, x_4\}$ and two constraints that intersect on x_1 and x_2. Tuples in italics are inconsistent and the rest are consistent. Assume we wish to determine if the values for x_3 are PWIC. PWIC-1 checks all 5 tuples of c_2 for each value of x3, as depicted in Figure 3a (for each tuple τ in c_1, all tuples of c_2 between the pair of edges starting from τ are checked against τ). PWIC-2 checks all 5 tuples only for value 0 of x3. After locating $\{0, 0, 4\}$ of c_2 as a PW-support for $\{0, 0, 0\}$ of c_1, $lastPW_{c_1,c_2,s}$, where $s = \{0, 0\}$, points to $\{0, 0, 4\}$. For the rest of x3's values, PWIC-2 only checks this.

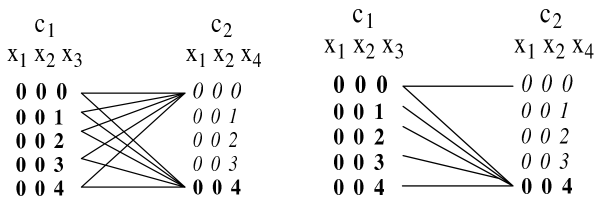

Figure 3. Applying PWIC-1 and PWIC-2.

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

Proof: The complexity is again determined by the calls to `Revise`. When looking for a GAC-support within `Revise`, `PWIC-2` is identical to `PWIC-1` and therefore the two algorithms have the same cost for this part. That is, $O(kd^{k-1})$ to make a value $a \in D(x_j)$ GAC for all calls to `Revise`(x_j, c_i). Once a GAC-support τ is found, `PWIC-2` iterates over the constraints that intersect with c_i. Assuming that $\tau[var(c_i) \cap var(c_m)] = s$, for any intersecting constraint c_m `PWIC-2` searches through the tuples that include assignment s. However, these tuples are not searched from scratch every time. Since the pointer $lastPW_{c_i,c_m,s}$ is used, the tuples that include assignment s are searched from scratch **only** during the first time a GAC-support τ with $\tau[var(c_i) \cap var(c_m)] = s$ is located. In any subsequent iteration of the for loop in line 17 of Figure 1, and in any subsequent call to `Revise`(x_j, c_i), whenever a GAC-support that includes s is located, the algorithm first checks if $lastPW_{c_i,c_m,s}$ is still valid. If it is not, then only tuples "after"

$lastPW_{c_i,c_m,s}$ are searched, As a result, in the $O(d^{k-1})$ times a GAC-support for a is located, each tuple of each intersecting constraint is checked at most once (with each check costing $O(k-f)$) and there are at most d^{k-1} checks for the validity of $lastPW_{c_i,c_m,s}$. Thus, the cost of $\texttt{Revise}(x_j, c_i)$ is $O(kd^{k-1} + e(d^{k-1} + (k-f)(d^{k-1})))$=$O(ekd^{k-1})$. For the $O(kd)$ times `Revise`is called to make a constraint PWIC, the upper bound is $O(ek^2d^k)$ checks. For e constraints the worst-case complexity is $O(e^2k^2d^k)$. □

The space complexity of `PWIC-2` is $O(e^2d^f)$. `PWIC-2` is thus not practical for large arity constraints sharing many variables. Note that, even when constraints share just two variables (and the space complexity is $O(e^2d^2)$), PWIC is stronger than GAC. It may be beneficial to apply PWIC only to variables participating in certain constraints, based on properties like the arity and the number of shared variables, and to apply GAC to the rest of the variables. Indeed, our two algorithms for PWIC apply such a hybrid propagation scheme, as they apply GAC to constraints that intersect on just one variable.

5 EXPERIMENTS

We compared algorithms that maintain GAC2001/3.1, `PWIC-1`, `PWIC-2`, `RPIC-1`, and `RPIC-2` throughout search (`RPIC-1` and `RPIC-2` are algorithms for rel PIC similar to the corresponding algorithms for PWIC). We simply refer to these search algorithms by the consistency enforced. All algorithms used the dom/deg variable ordering heuristic [4]. Figure 4 shows the average CPU time to solve 50 instances of randomly generated problems with 30 variables, uniform domain size of 10, 28 4-ary constraints, and varying constraint looseness (along the x-axis). The problems were generated using the extended model B [2]. On these instances, `PWIC-1` and `PWIC-2` are faster than GAC and `RPIC-1` and `RPIC-2` are slower. From experiments with other parameters, we conjecture that PWIC is more efficient than GAC on sparse problems, especially when the domain size is large. On the other hand, PWIC is too expensive on problems with medium or high density. Although it significantly reduces the nodes visited, it is outperformed by GAC in CPU time. Rel PIC is generally less efficient than both GAC and PWIC.

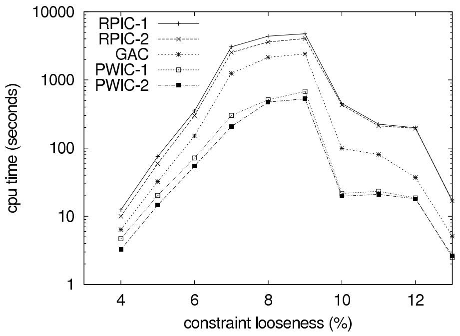

Figure 4. Comparison on random problems.

We also ran experiments on the CLib configuration benchmark library (see `www.itu.dk/doi/VeCoS/clib`). To obtain hard instances, each problem was slightly altered by adding a few variables (5-6) and constraints (8-10) randomly, as described in [11].

Table 1 gives results for GAC, and (for space reasons) only `PWIC-1` and `RPIC-1`. There is a significant difference in run times in favor of the algorithms that maintain inverse consistencies due to the additional pruning they perform. `PWIC-2` and `RPIC-2` are faster than `PWIC-1` and `RPIC-1`, by a small margin on average. The PWIC algorithms are again more efficient than the rel PIC ones.

problem	n	e	k	d	GAC nodes-time	PWIC-1 nodes-time	RPIC-1 nodes-time
machine	30	22	4	9	535874-14,38	12600-1,31	36046-4,27
fx	24	21	5	44	193372-4,06	1315-0,09	17624-1,32
fs	29	18	6	51	618654-23,29	19-0,00	1698-0,23
esvs	33	20	5	61	6179966-153,71	7859-0,24	39021-3,21

Table 1. Configuration problems. k and d are the maximum arity and domain size. Averages are over 50 hard instances for each benchmark.

6 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 study of two such consistencies, rel PIC and PWIC. Our theoretical study revealed some surprising results. For example, rel PIC and PWIC are weaker than PIC when restricted to binary constraints, while for non-binary constraints rel PIC and PWIC are incomparable to rel AC. We also described algorithms that can be used to achieve PWIC and rel PIC. One has a particularly good time complexity, competitive with GAC algorithms, though with a higher space cost. Experiments demonstrated the potential of inverse consistencies as an alternative or complementary to GAC. As future work, we will investigate ways to combine inverse consistencies with specialized GAC propagators.

ACKNOWLEDGEMENTS

We would like to thank Patrick Prosser for useful comments and suggestions on an early draft of the paper.

REFERENCES

[1] F. Bacchus, X. Chen, P. van Beek, and T. Walsh, 'Binary vs. Non-binary CSPs', *Artificial Intelligence*, **140**, 1–37, (2002).
[2] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa, 'On Forward Checking for Non-binary Constraint Satisfaction', *Artificial Intelligence*, **141**, 205–224, (2002).
[3] C. Bessière and J.C. Régin, 'Arc Consistency for General Constraint Networks: Preliminary Results', in *Proc. of IJCAI'97*, pp. 398–404, (1996).
[4] C. Bessière and J.C. Régin, 'MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems', in *Proc. of CP'96*, pp. 61–75, (1996).
[5] C. Bessière, J.C. Régin, R. Yap, and Y. Zhang, 'An Optimal Coarse-grained Arc Consistency Algorithm', *Artificial Intelligence*, **165**(2), 165–185, (2005).
[6] R. Debruyne and C. Bessière, 'Domain Filtering Consistencies', *JAIR*, **14**, 205–230, (2001).
[7] E. Freuder, 'A Sufficient Condition for Backtrack-bounded Search', *JACM*, **32**(4), 755–761, (1985).
[8] E. Freuder and C. Elfe, 'Neighborhood Inverse Consistency Preprocessing', in *Proc. of AAAI'96*, pp. 202–208, (1996).
[9] P. Janssen, P. Jègou, B. Nouguier, and M.C. Vilarem, 'A filtering process for general constraint satisfaction problems: Achieving pairwise consistency using an associated binary representation', in *Proc. of IEEE Workshop on Tools for Artificial Intelligence*, pp. 420–427, (1989).
[10] P. Jègou, 'On the Consistency of General Constraint Satisfaction Problems', in *Proc. of AAAI'93*, pp. 114–119, (1993).
[11] N. Samaras and K. Stergiou, 'Binary Encodings of Non-binary CSPs: Algorithms and Experimental Results', *JAIR*, **24**, 641–684, (2005).
[12] P. van Beek and R. Dechter, 'On the Minimality and Global Consistency of Row-convex Constraint Networks', *JACM*, **42**(3), 543–561, (1995).
[13] G. Verfaillie, D. Martinez, and C. Bessière, 'A Generic Customizable Framework for Inverse Local Consistency', in *Proc. of AAAI'99*, pp. 169–174, (1999).