A First Practical Algorithm for High Levels of Relational Consistency

Shant Karakashian[1], **Robert J. Woodward**[1], **Christopher Reeson**[1], **Berthe Y. Choueiry**[1] and **Christian Bessiere**[2]

[1]Constraint Systems Laboratory, University of Nebraska-Lincoln, USA
{shantk|rwoodwar|creeson|choueiry}@cse.unl.edu

[2]LIRMM-CNRS, University of Montpellier, France
bessiere@lirmm.fr

Abstract

Consistency properties and algorithms for achieving them are at the heart of the success of Constraint Programming. In this paper, we study the relational consistency property R($*$,m)C, which is equivalent to m-wise consistency proposed in relational databases. We also define wR($*$,m)C, a weaker variant of this property. We propose an algorithm for enforcing these properties on a Constraint Satisfaction Problem by tightening the existing relations and without introducing new ones. We empirically show that wR($*$,m)C solves in a backtrack-free manner all the instances of some CSP benchmark classes, thus hinting at the tractability of those classes.

1. Introduction

Local consistency is at the heart of the success of Constraint Programming and perhaps best distinguishes this field from other scientific disciplines that study the same combinatorial problems. In this paper, we study the relational consistency property R($*$,m)C, present an algorithm for enforcing it, and compare the impact of our approach on the performance of problem solving with that of other consistency properties.

R($*$,m)C is a relational consistency property for non-binary Constraint Satisfaction Problems (CSPs) equivalent to m-wise consistency proposed in the area of Relational Databases (Gyssens 1986). When enforced, every consistent assignment of variables appearing in the scope of a constraint can be extended to a consistent assignment of the variables in the scope of every $(m-1)$ other constraints. Enforcing R($*$,m)C filters existing relations but does not add any new constraint to the problem.

We borrow the notation 'relational (i,m)-consistency' from (Dechter and van Beek 1997; Dechter 2003), and abbreviate it to 'R($*$,m)C', where '$*$' indicates that the property is concerned with only 'the scopes of the m considered constraints whatever their sizes are.' An obvious algorithm for enforcing R($*$,m)C is joining every combination of m constraints and projecting the result on their respective scopes: $\forall R_i \in \{R_1, \cdots, R_m\}, R_i = \pi_{scope(R_i)}(\bowtie_{j=1}^{m} R_j)$. The space complexity of this obvious algorithm is too prohibitive to be useful in practice. We propose an alternative algorithm that overcomes that limitation. When enforcing R($*$,m)C on every combination of m relations in the problem, much of this work is redundant and could be avoided. We introduce a weakened variant of R($*$,m)C, which we call wR($*$,m)C and obtain by removing redundant edges from the dual graph of the CSP (Dechter 2003; Janssen et al. 1989).

The contributions of this work are as follows: (1) the introduction and discussion of the relational consistency properties R($*$,m)C and wR($*$,m)C; (2) the design of a parameterized algorithm for enforcing those properties; (3) the design of a new data structure for locating tuples in large relations; (4) the analysis of the worst-case complexity of the new algorithm; and (5) the empirical evaluation of our approach on benchmark problems, demonstrating its ability to solve, in backtrack-free manner, all the instances of some classes of those benchmarks, suggesting that it uncovers the tractability of those classes.

This paper is structured as follows. Section 2 defines non-binary CSPs. Section 3 discusses R($*$,m)C. Section 4 introduces a weakened variant of R($*$,m)C obtained by removing redundant edges from the dual graph. Section 5 presents our algorithm for enforcing R($*$,m)C along with a new data structure for facilitating its implementation. Section 6 discusses our experimental results. Section 7 reviews the state of the art in relational consistency, and Section 8 lists future work and our conclusions.

2. Background

A Constraint Satisfaction Problem (CSP) is defined by $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints or relations. We use the terms constraint and relation interchangeably. Each variable $V_i \in \mathcal{V}$ has a finite domain $D_i \in \mathcal{D}$, and is constrained by a subset of the relations in $\mathcal{C}$. In a relation R, tuple $\tau \in R$ is a combination of allowed values for the variables in the scope of R. A solution to the CSP is an assignment of a value to each variable such that all the constraints are satisfied.

The *dual graph* of a CSP is a graph whose vertices represent the relations of the CSP, and whose edges connect two vertices corresponding to relations whose scopes overlap. The 'dual CSP' is thus a binary CSP where the variables are the relations of the original CSP, their domains are the tuples of those relations, and the constraints enforce *equalities* over the shared variables. We denote by φ a *combination of m constraints* that induce a connected component in the

dual graph. We refer to the set of all such combinations of size m in a given CSP by Φ. Finally, π and $\bowtie$ denote the relational operators project and join, respectively.

To reduce the severity of the combinatorial explosion, CSPs are usually filtered by enforcing a given local consistency property. One common such property is Generalized Arc Consistency (GAC). A CSP is GAC iff for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint. Using the terminology introduced in (Debruyne and Bessière 1997), we say that a local consistency property LC is *stronger* than another LC' if in any CSP where LC holds, LC' also holds. Further, we say that LC is *strictly stronger* than LC' if LC is stronger than the LC' and there exists at least one CSP in which LC' holds but not LC. Similarly to (Bessiere, Stergiou, and Walsh 2008), we say that LC and LC' are *equivalent* when LC is stronger than LC' and vice versa. In practice, when a consistency property is stronger (respectively, weaker) than another, enforcing the former never yields less (respectively, more) pruning than enforcing the latter on the same problem does.

3. R(∗,m)C

Below, we introduce and discuss R(∗,m)C, which we define using the definition format of R(i,m)C in (Dechter 2003).

Definition 1 *A set of m relations $\mathcal{R} = \{R_1, \cdots, R_m\}$ with $m \geq 2$ is said to be R(∗,m)C iff every tuple in each relation $R_i \in \mathcal{R}$ can be extended to the variables in $\bigcup_{R_j \in \mathcal{R}} scope(R_j) \setminus scope(R_i)$ in an assignment that satisfies all the relations in $\mathcal{R}$ simultaneously. A network is R(∗,m)C iff every set of m relations, $m \geq 2$, is R(∗,m)C.*

Informally, in every given set φ of m relations, every tuple τ in every relation $R \in \varphi$ can be extended to a tuple τ' in each $R' \in \varphi \setminus \{R\}$ such that all those tuples form a consistent solution to the relations in φ. R(∗,m)C can be enforced by filtering the existing relations and without introducing any new relations to the CSP as follows. We repeatedly apply the following operation to all combinations of m relations $\{R_1, \cdots, R_m\}$ until quiescence:

$$\forall R_i \in \{R_1, \cdots, R_m\}, R_i = \pi_{scope(R_i)}(\bowtie_{j=1}^{m} R_j) \quad (1)$$

Expression (1) gives us an obvious algorithm for R(∗,m)C, but the space requirement is prohibitive in practice.

After enforcing R(∗,m)C on a constraint network, variable domains are filtered by projecting the filtered relations on the domains of the variables. Interestingly, these domain reductions do not break the R(∗,m)C property.

Theorem 1 *If a network is R(∗,m)C, domain filtering by GAC cannot enable further constraint filtering by R(∗,m)C.*

Proof: The proof is by contradiction. Assume that filtering the domains with GAC after enforcing R(∗,m)C removes value x from the domain of variable V_i. Then, there exists a relation R_a that applies to V_i where the value x for V_i does not appear in any tuple in R_a. For GAC to enable further constraint filtering by R(∗,m)C, there must exist at least one constraint R_b that applies to V_i and the value x for V_i appears in some tuple in R_b. Thus, there must be a tuple in R_b that cannot be extended to a tuple in R_a, which yields contradiction because the problem is R(∗,m)C. □

Now we compare R(∗,m)C with RmC (Dechter and van Beek 1997). For a given set $\{R_1, \cdots, R_m\}$ of m relations RmC requires the projection of the joined relations on all subsets $A \subseteq \bigcup_{i=1}^{m} scope(R_i)$. Hence, every subset introduces a new constraint, except those that have the same scope as existing constraints. In contrast, R(∗,m)C projects the joined relations on the scope of each of its original relations, without adding any new constraints.

Theorem 2 *RmC is strictly stronger than R(∗,m)C.*

Proof: Consider a CSP $\mathcal{P}$ and let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the problem obtained after enforcing RmC and R(∗,m)C on $\mathcal{P}$, respectively. Consider a partial assignment τ over some of the variables of $\mathcal{P}$, $scope(\tau)$, that is consistent with the constraints of $\mathcal{P}_{rmc}$. We prove that τ must necessarily be consistent with the constraints in $\mathcal{P}_{r*mc}$. Assume that τ is not consistent with the constraints in $\mathcal{P}_{r*mc}$. Thus, there must be at least one relation R_{x*} in $\mathcal{P}_{r*mc}$ s.t. $\tau \notin \pi_{scope(\tau)}(R_{x*})$. For every relation R in $\mathcal{P}$ there is a relation in $\mathcal{P}_{r*mc}$ and another one in $\mathcal{P}_{rmc}$ with the same scope as R. $\mathcal{P}_{r*mc}$ does not have any additional relations but $\mathcal{P}_{rmc}$ does. Thus, $\mathcal{P}_{rmc}$ must have a relation R_x s.t. $scope(R_{x*})$=$scope(R_x)$. Since τ is a consistent partial solution in $\mathcal{P}_{rmc}$, then $\tau \in \pi_{scope(\tau)}(R_x)$. $\tau \in \pi_{scope(\tau)}(R_x)$ and $\tau \notin \pi_{scope(\tau)}(R_{x*})$ is impossible because joining more relations of $\mathcal{P}_{rmc}$ and projecting them on the same scope cannot possibly introduce more tuples. Thus, we reach a contradiction and RmC is stronger than R(∗,m)C.

Below, we provide an example that is R(∗,m)C but not RmC. Let $\mathcal{P}$ be the following Boolean CSP with the four variables V_1, V_2, V_3, and V_4 and the four constraints: $C_{V_1,V_2} = C_{V_2,V_3} = C_{V_3,V_4} = C_{V_4,V_1} = \{\langle 0,0\rangle, \langle 1,1\rangle\}$. Let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the problems after RmC and R(∗,m)C are enforced on $\mathcal{P}$, respectively. The partial assignment $\langle (V_1, 0), (V_3, 1)\rangle$ is consistent in $\mathcal{P}_{r*mc}$ because $\mathcal{P}$ has no constraint between V_1 and V_3 and by definition, R(∗,m)C does not add new constraints. However, this partial assignment violates the constraint $C_{V_1,V_3} = \{\langle 0,0\rangle, \langle 1,1\rangle\}$ which is added in $\mathcal{P}_{rmc}$ by RmC. Thus, RmC is strictly stronger than R(∗,m)C. □

4. Weakening R(∗,m)C

We propose wR(∗,m)C, a weakened version of R(∗,m)C, which requires significantly less time and space than R(∗,m)C while slightly reducing the amount of pruning.

In the dual graph, edges enforce the equality of the shared variables of two adjacent vertices. Janssen et al. (1989) and Dechter (2003) observed that an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path. Such redundant edges can be removed without modifying the set of solutions. Janssen et al. (1989) introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but they are all guaranteed to have the same number of remaining edges. Figure 1 shows the dual graph of a CSP, where the edges drawn in dashed lines are redundant.

Indeed, the same value for A is enforced between R_1 and R_3 through R_4, and that for C between R_2 and R_3 through R_5.

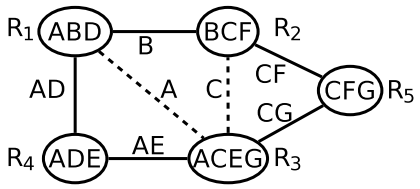

Figure 1: Dual graph.

To enforce the R(∗,m)C property on a CSP, we must consider only combinations of relations that induce a *connected* component in the dual graph because tuples can be trivially extended to relations that do not share variables. For wR(∗,m)C, instead of using the original dual graph to generate the combinations of m relations on which to enforce the R(∗,m)C property, we propose to use the minimal dual graph obtained using the algorithm of (Janssen et al. 1989). While this operation reduces the number of combinations considered (and consequently the time needed to process them and the space needed to store them), it may yield a weaker filtering of the constraints.

Definition 2 wR(∗,m)C *relative to a given minimal dual graph of a CSP $\mathcal{P}$ is defined as the property of $\mathcal{P}$ where all the combinations of m relations that induce connected components in the minimal dual graph verify the R(*,m)C consistency property. Note that $m\geq 2$.*

Given that, in general, more than one possible minimal dual network exists, the property obviously depends on the minimal dual graph chosen, and is always defined relatively to that graph. For sake of simplicity however, the particular minimal dual graph is not included in the notation.

Theorem 3 wR(∗,2)C *on any minimal dual graph of a CSP and* R(∗,2)C *are equivalent.*

Proof: The case where $m = 2$ corresponds to pairwise consistency and the proof is given in (Janssen et al. 1989). □

Theorem 4 *$\forall a, b\in\mathbb{N}$ where $a<b\leq|\mathcal{C}|$,* wR(∗,b)C *is strictly stronger than* wR(∗,a)C *on the same connected minimal dual graph of the CSP.*

Proof: Let Φ_a and Φ_b be the set of combinations of wR(∗,a)C and wR(∗,b)C, respectively. For every $\varphi_a\in\Phi_a$ there exists $\varphi_b\in\Phi_b$ such that $\varphi_a\subset\varphi_b$. *wR(∗,b)C* is stronger than *wR(∗,a)C*.

Consider the Boolean CSP $\mathcal{P}_e$ with the three variables V_1, V_2, and V_3 and the three constraints: $C_{V_1,V_2} = C_{V_2,V_3} = C_{V_1,V_3} = \{\langle 0,1\rangle, \langle 1,0\rangle\}$. Clearly, $\mathcal{P}_e$ is wR(∗,2)C but not wR(∗,3)C. □

Corollary 1 wR(∗,3)C *is strictly stronger than* R(∗,2)C *on any connected minimal dual graph of the CSP where $|\mathcal{C}|\geq 3$.*

Proof: By Theorem 3, R(∗,2)C is equivalent to wR(∗,2)C. By Theorem 4, wR(∗,3)C is stronger than wR(∗,2)C. Further, the CSP $\mathcal{P}_e$ used in the proof of Theorem 4 is R(∗,2)C but not wR(∗,3)C. □

Theorem 5 *$\forall m>2$,* R(∗,m)C *is strictly stronger than* wR(∗,m)C *on any connected minimal dual graph of the CSP.*

Proof: Every combination of relations considered by wR(∗,m)C is also considered by R(∗,m)C. Hence, R(∗,m)C is stronger than wR(∗,m)C.

Assume that wR(∗,m)C is stronger than R(∗,m)C and that the CSP of Figure 1 is inconsistent because there is no assignment for the variables A, B, D that simultaneously satisfies relations $\{R_1, R_2, R_3\}$. For example, assume that $\pi_{AB}(R_1) = \pi_{BC}(R_2) = \{\langle 1,1\rangle, \langle 0,0\rangle\}$, and $\pi_{AC}(R_3) = \{\langle 0,1\rangle, \langle 1,0\rangle\}$. For m=3, the combination $\{R_1, R_2, R_3\}$ considered by R(∗,3)C uncovers the inconsistency. However, this combination is not considered by wR(∗,m)C on the minimal dual graph obtained from removing the two dashed-line edges because the combination induces a disconnected sub-graph of that minimal dual graph. Therefore, wR(∗,m)C fails to uncover the inconsistency uncovered by R(∗,m)C. □

5. An Algorithm for Enforcing R(∗,m)C

Expression (1) gives an obvious algorithm for enforcing R(∗,m)C. However, this algorithm requires computing and materializing the join of each combination of m relations, which can be prohibitive in practice. Below we propose PROCESSQUEUE, an algorithm that avoids computing and storing the intermediate joins. First, we describe initializing the queue on which PROCESSQUEUE operates, then we discuss PROCESSQUEUE. After that, we describe the search for supports and the data structure we designed for this purpose.

Definition 3 *The support of a tuple τ of a relation R in a combination φ of relations, denoted $S_{\tau,\varphi}$, is a set of tuples that verifies the condition: $\forall R_i\in\varphi\backslash\{R\}\exists\tau_i\in S_{\tau,\varphi}$, $\tau_i\in R_i$ and the tuples in $S_{\tau,\varphi}\cup\{\tau\}$ agree on all shared variables.*

5.1 Initializing the queue

Given the dual graph (or a minimal dual graph) of a CSP, let Φ be the set of all combinations of m relations that induce connected components of the considered graph. We initialize the queue, $\mathcal{Q}$, over which our algorithm operates, to all the combination-relations pairs $\langle\varphi, R\rangle$ such that $\varphi\in\Phi$ and $R\in\varphi$. We have developed an algorithm, not reported here for lack of space, that computes Φ *while exploiting the topology of the considered graph*. The advantage of our algorithm is that it enumerates each connected component once and none of the non-connected components.

5.2 Processing the queue

PROCESSQUEUE takes as input $\mathcal{Q}$ and Φ, see Algorithm 1. It filters the relations to enforce R(∗,m)C, and returns true if it is successful and false otherwise. It proceeds by removing a combination-relation pair $\langle\varphi, R\rangle$ from the queue (Line 2), and searches a support in φ for each $\tau\in R$ (Line 5). A tuple that does not have a support is deleted from R (Line 7). When a relation loses its last tuple, the algorithm returns false (Line 8). If, after processing all the tuples in R, any tuples are deleted, the relations affected by the update of R are added to the queue. The affected relations are those that appear with R in a combination other than φ. Notice that a relation R' that appears in a combination with R needs to

Algorithm 1: PROCESSQUEUE enforces R(∗,m)C

Input: $\mathcal{Q}$, Φ
Output: $true$ if the problem is R(∗,m)C, $false$ otherwise

```
1  while (Q ≠ ∅) do
2      ⟨φ, R⟩ ← POP(Q)
3      deleted ← false
4      foreach τ ∈ R do
5          support ← SEARCHSUPPORT(τ, φ)
6          if support = false then
7              DELETE(τ, R)
8              if R = ∅ then return false
9              deleted ← true
10     if deleted then foreach φ' ∈ (Φ \ {φ}), R ∈ φ' do
11         foreach R' ∈ (φ' \ {R}) do
12             Q ← Q ∪ {⟨φ', R'⟩}
13 return true
```

be checked only in those combinations in which it appears along with R. Therefore, when added to the queue, an affected relation R' is paired with the combination φ', other than φ, that includes both R and R' (Line 12).

5.3 Searching for a support

To find a support $S_{\tau,\varphi}$ for a tuple τ of a relation R in a combination φ, SEARCHSUPPORT conducts a backtrack search on the dual encoding of the CSP induced by φ. This dual CSP is denoted $\mathcal{P}_{D\varphi}$. The variables of $\mathcal{P}_{D\varphi}$ are the relations in φ. Their domains are the tuples of the relations except for the variable corresponding to R which is assigned the tuple τ. The constraints in $\mathcal{P}_{D\varphi}$ are binary, and enforce the equality of the shared scope of the relations in φ. A solution to $\mathcal{P}_{D\varphi}$ is $S_{\tau,\varphi}$, the support set of τ in φ. The search stops at the first solution and returns $S_{\tau,\varphi}$. It returns false if no solution is found. The search process uses forward checking and dynamic variable ordering with the domain/degree heuristic.

5.4 The index-tree data structure

In order to effectively implement the above mentioned forward-checking, we need to locate all the tuples in a relation R_j that are consistent with a tuple τ_i of a relation R_i. For that purpose, we designed the new index-tree data structure, which we introduce below. We assume that the relations are implemented as tables of consistent tuples and that the variables are in a canonical order. Each table includes a column to indicate that the tuple is deleted (1) or not (0).

An index tree is built for each relation and each subset of its scope that is shared with another relation in the problem. Given two relations R_i and R_j and $\mathcal{V}_s$=$scope(R_i) \cap scope(R_j)$, the index tree $IT_{R_j,\mathcal{V}_s}$ returns for $\tau_i \in R_i$ all tuples $\tau_j \in R_j$ to which τ_i can be extended, that is $\pi_{\mathcal{V}_s}(\tau_i) = \pi_{\mathcal{V}_s}(\tau_j)$. An index tree $IT_{R_j,\mathcal{V}_s}$ is a rooted tree, with a dummy root, where all leaves are at height $|\mathcal{V}_s|$. The level of a node in the tree corresponds to a variable in $\mathcal{V}_s$. The nodes are labeled with values of the variables in $\mathcal{V}_s$. Each leaf node holds a list of pointers to tuples in R_j. Figure 2 shows an example of an index tree for the relation R_j and

	A	B	C	D	*deleted*
t_1	0	0	1	0	0
t_2	0	1	1	0	0
t_3	0	1	1	1	0
t_4	1	1	1	1	0

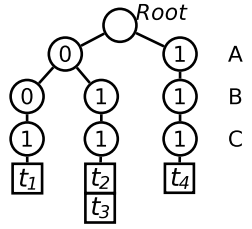

Figure 2: $IT_{R_j,\{A,B,C\}}$.

$\mathcal{V}_s$=$\{A, B, C\}$.

The tree is built as follows. The tuples of R_j are sequentially inserted in the tree. For a given tuple $\tau_j \in R_j$, we consider $\pi_{\mathcal{V}_s}(\tau_j)$. Traversing $IT_{R_j,\mathcal{V}_s}$ from the root, we match the value of a variable in $\pi_{\mathcal{V}_s}(\tau_j)$ with the label of a child of the current node in the tree. If the two values match, we move to that child node in the tree and to the value of the next variable in $\pi_{\mathcal{V}_s}(\tau_j)$. Otherwise, we add a new child node with the value of the variable in $\pi_{\mathcal{V}_s}(\tau_j)$. When the variables in $\mathcal{V}_s$ are exhausted, we insert τ_j at the end of the list at the leaf node.

When searching for the tuples in R_j that are consistent with τ_i, we traverse the tree as explained above for $\pi_{\mathcal{V}_s}(\tau_i)$. If, at a given level, no child to a tree node can be found, we conclude that no such tuple exists and return null. Otherwise, we return, from the list of pointers at the leaf, the non-deleted matching tuples.

The complexity of building the index tree is $\mathcal{O}(|\mathcal{V}_s|td)$ for time and $\mathcal{O}(|\mathcal{V}_s|t)$ for space, where t is the number of tuples in the relation and d the largest domain size of the variables in $\mathcal{V}_s$. This bound is reached when each leaf node points to a single tuple. The time complexity of a query is $\mathcal{O}(|\mathcal{V}_s|d+t)$.

5.5 Improving the search for support

We propose two improvements to the search for support.

When $S_{\tau,\varphi}$ is found, it is stored for the tuple-combination pair $\langle \tau, \varphi \rangle$, and reused as long as every tuple in $S_{\tau,\varphi}$ remains valid, similar to ACS-residue algorithm (Likitvivatanavong et al. 2007). The importance of this improvement is further discussed in the complexity analysis.

Further, once $S_{\tau,\varphi}$ is found, the support of every tuple $\tau' \in S_{\tau,\varphi}$ can be directly set to be $(S_{\tau,\varphi} \cup \{\tau\}) \setminus \{\tau'\}$, thus saving SEARCHSUPPORT the effort of searching for supports for all τ'. This mechanism is reminiscent of the multi-directional support of (Lecoutre and Hemery 2007).

5.6 Complexity analysis

The time complexity of the algorithm is dominated by PROCESSQUEUE, hence the initialization phase is omitted from the analysis. Let t be the maximum number of tuples in a relation. It is bounded by $\mathcal{O}(d^k)$, where d is maximum domain size and k is the maximum arity of the relations. The number of constraints is e and the maximum number of combinations is $\binom{e}{m}$ and bounded by $\mathcal{O}(minimum(e^m, e^{\frac{e}{2}}))$. Below, we assume that $m < \frac{e}{2}$.

PROCESSQUEUE has two nested loops. The outer loop iterates over the combination-relation pairs in $\mathcal{Q}$. The number

of times that the outer loop iterates is the initial size of $\mathcal{Q}$, which is $\mathcal{O}(e^m)$, plus the number of times a combination-relation pair is added to $\mathcal{Q}$ in Line 12. A relation can participate at most in e^{m-1} combinations. Therefore, whenever a tuple is deleted $\mathcal{O}(e^{m-1})$ pairs are queued in Line 12. There are $\mathcal{O}(te)$ tuples and each tuple is deleted at most once. Thus, Line 10 is executed at most $\mathcal{O}(te)$ times, each time enqueuing $\mathcal{O}(e^{m-1})$ pairs. Therefore, the outer loop iterates at most $\mathcal{O}(te^m)$ times. The inner iterates over the tuples in a relation, $\mathcal{O}(t)$ times. When a support for a tuple has been identified, SEARCHSUPPORT costs $\mathcal{O}(m)$ to verify that every tuple in the support is still valid. When any tuple in the support has been deleted, SEARCHSUPPORT executes a backtrack search on $\mathcal{P}_{D\varphi}$. $\mathcal{P}_{D\varphi}$ has m variables of maximum domain size t, and the first variable is instantiated. Thus, the complexity of the backtrack search is $\mathcal{O}(t^{m-1})$, and that of the inner loop is $\mathcal{O}(t^m)$. Thus, PROCESSQUEUE is $\mathcal{O}(t^{m+1}e^m)$. The time complexity of PROCESSQUEUE is not worse than that of the obvious algorithm based on Expression (1), which is $\mathcal{O}(t^{m+1}e^{m+1})$.

When intermediate joins are not stored, the space complexity of the obvious algorithm is $\mathcal{O}(t^m)$, and constitutes a major bottleneck for its practical implementation. The space complexity of PROCESSQUEUE is dominated by the space for storing the $\mathcal{O}(e^2)$ index trees, which is $\mathcal{O}(kte^2)$.

Thus, our algorithm dramatically reduces the space complexity while slightly improving the time complexity.

6. Experimental Results

To evaluate the performance of our algorithm for enforcing wR(∗,m) (i.e., R(∗,m)C on the minimal dual graph), we compare it against GAC2001 (Bessiere et al. 2005) and maxRPWC (Bessiere, Stergiou, and Walsh 2008). All those algorithms are integrated as full lookahead strategies in a backtrack search procedure. After enforcing wR(∗,m) in the lookahead schema, we filter the domains of the uninstantiated variables by projecting the constraints on the variables. The search procedure finds the first solution of the original CSP using the domain/degree heuristic for dynamic variable ordering. During search, we timestamp the deleted tuples by the variable's instantiation. Upon backtracking, we restore all tuples that have the timestamp of the variable's instantiation.

The experiments are conducted on the benchmarks of the CSP Solver Competition[1] with a time limit of one hour per instance. They compare wR(∗,2)C, wR(∗,3)C, wR(∗,4)C, GAC, and maxRPWC. We split the benchmark problems into three groups, discussed in Tables 1, 2 and 3 respectively. The tables give the number of nodes visited (#Nodes), the CPU time in seconds (Time), and the maximum time (Max time) for the instances *completed* within a one-hour time limit. They also give the number of instances completed (#C), the number of instances with the fastest running time (#F), and the number of instances solved backtrack free (#BF). Time out is denoted as '-' and memory out as 'mem.' CPU time includes preprocessing. Importantly, the averages of #Nodes, Time, and Max time are computed over *only* the instances completed by *all* the compared algorithms, but algorithms that do not complete any instance are not taken into consideration. Thus, those values should be considered in light of the number of completed instances.

Table 1: Results on the benchmark problems of the first group.

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
modifiedRenault (instances: 50, vars: 111, dom: 42, rels: 147, arity: 10)						
wR(∗,2)C	192.5	2.99	18.45	46	**27**	41
wR(∗,3)C	82.5	7.55	12.43	**50**	4	48
wR(∗,4)C	82.5	33.88	70.61	**50**	2	**50**
GAC	1,324,309.8	402.44	2,436.40	26	14	4
maxRPWC	2,110.8	305.37	2,886.63	31	3	19
rand-8-20-5 (instances: 20, vars: 20, dom: 5, rels: 18, arity: 8)						
wR(∗,2)C	941.3	1,162.22	2,979.77	**16**	**14**	0
wR(∗,3)C	-	-	-	0	0	0
wR(∗,4)C	mem	-	-	0	0	0
GAC	30,501.7	1,795.26	3,480.93	9	2	0
maxRPWC	-	-	-	0	0	0
rand-10-20-10 (instances: 20, vars: 20, dom: 10, rels: 5, arity: 10)						
wR(∗,2)C	0.0	0.20	0.24	20	4	20
wR(∗,3)C	0.0	0.21	0.25	20	1	20
wR(∗,4)C	0.0	**0.18**	**0.22**	20	19	20
GAC	210.0	9.34	13.17	20	0	0
maxRPWC	0.0	2.45	6.81	20	0	20
dag-rand (instances: 25, vars: 23, dom: 3, rels: 16, arity: 15)						
wR(∗,2)C	0.0	27.21	31.51	**25**	**25**	**25**
wR(∗,3)C	0.0	37.75	40.15	**25**	0	**25**
wR(∗,4)C	mem	-	-	0	0	0
GAC	-	-	-	0	0	0
maxRPWC	-	-	-	0	0	0
aim-100 (instances: 24, vars: 100, dom: 2, rels: 154, arity: 3)						
wR(∗,2)C	66,905.8	12.65	72.43	18	5	5
wR(∗,3)C	5,578.4	24.40	252.07	18	4	7
wR(∗,4)C	127.5	11.47	65.89	**19**	3	**12**
GAC	30,993,972.2	602.91	2,758.48	17	0	1
maxRPWC	11,408,274.4	298.69	1,348.66	17	**7**	1
aim-200 (instances: 24, vars: 200, dom: 2, rels: 315, arity: 3)						
wR(∗,2)C	2,670.2	35.51	71.98	12	**7**	4
wR(∗,3)C	580.2	35.91	79.96	**14**	**7**	8
wR(∗,4)C	443.8	240.13	837.54	**14**	2	**9**
GAC	1,876,247.6	542.48	2,114.74	8	0	0
maxRPWC	842,488.8	414.05	1,583.34	8	1	0

The usefulness of stronger consistency is best illustrated on the problems of Table 1. wR(∗,m)C is the fastest on most instances, and able to solve more instances than GAC or maxRPWC[2]. In many instances, GAC takes more than 100 times the CPU time of wR(∗,m)C. In particular, many `modifiedRenault` instances are solved in a few seconds with wR(∗,m)C, but not completed in one hour by GAC. Moreover, wR(∗,m)C solves many more instances backtrack free than GAC and maxRPWC do. We emphasize that all `dag-rand` and `modifiedRenault` are solved backtrack free by wR(∗,2)C and wR(∗,4)C, respectively. Thus, wR(∗,m)C hints at the tractability of the corresponding CSP

[1] http://www.cril.univ-artois.fr/CPAI08/

[2] Bessiere et al. (2008) showed that pairwise consistency (i.e., R(*,2)C) followed by GAC is strictly stronger than maxRPWC, which is strictly stronger than GAC.

Table 2: Results on the benchmark problems of the second group.

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
lexVg (instances:63, vars:100, dom:26, rels: 20, arity:10)						
wR(∗,2)C	51.3	3.97	57.77	51	0	27
wR(∗,3)C	51.3	61.61	611.4	43	0	27
wR(∗,4)C	4.5	260.77	2,871.45	39	0	**35**
GAC	50.5	0.43	8.77	**63**	**43**	26
maxRPWC	50.5	0.42	8.08	**63**	31	26
ogdVg (instances:65, vars:100, dom:26, rels: 20, arity:10)						
wR(∗,2)C	10.1	1.23	4.81	25	0	9
wR(∗,3)C	10.1	213.35	825.43	14	0	9
wR(∗,4)C	9.4	889.49	2,811.22	9	0	9
GAC	10.6	0.16	0.77	**35**	**19**	**11**
maxRPWC	10.6	0.16	0.78	**35**	**19**	**11**
renault (instances:2, vars:101, dom:42, rels: 134, arity:10)						
wR(∗,2)C	101.0	2.24	2.34	2	0	2
wR(∗,3)C	101.0	9.29	9.57	2	0	2
wR(∗,4)C	101.0	71.54	72.15	2	0	2
GAC	101.0	1.04	1.13	2	**2**	2
maxRPWC	101.0	370.68	377.54	2	0	2
ssa (instances:8, vars:435, dom:2, rels: 738, arity:5)						
wR(∗,2)C	574.0	0.16	0.3	2	1	2
wR(∗,3)C	574.0	0.22	0.41	2	0	2
wR(∗,4)C	574.0	0.33	0.61	2	0	2
GAC	574.0	0.06	0.11	2	**2**	2
maxRPWC	574.0	0.07	0.12	2	1	2

Table 3: Results on the benchmark problems of the third group.

Algorithm	#Nodes	Time	Max time	#C	#F	#BF
rand-3-20-20 (instances:50, vars:20, dom:20, rels: 60, arity:3)						
wR(∗,2)C	10,494.1	1,565.43	3,005.42	28	2	0
wR(∗,3)C	-	0	0	1	0	0
wR(∗,4)C	-	0	0	0	0	0
GAC	52,711.5	255.18	436.58	**48**	**46**	0
maxRPWC	52,331.5	1,945.08	3,560.47	23	0	0
rand-3-20-20-fcd (instances:50, vars:20, dom:20, rels: 60, arity:3)						
wR(∗,2)C	6,498.5	1,107.93	2,769.95	35	1	0
wR(∗,3)C	-	0	0	6	0	0
wR(∗,4)C	-	0	0	0	0	0
GAC	31,627.5	171.73	418.1	**49**	**48**	0
maxRPWC	31,203.2	1,220.33	3,453.52	30	0	0
dubois (instances: 13, vars: 300, dom: 2, rels: 200, arity: 3)						
wR(∗,2)C	34,172,289.7	259.05	454.45	5	0	0
wR(∗,3)C	34,172,289.7	850.92	1,474.69	4	0	0
wR(∗,4)C	7,381,321.7	955.42	1,640.00	3	0	0
GAC	26,352,376.3	92.04	160.57	**7**	**6**	0
maxRPWC	26,352,376.3	91.96	159.30	**7**	1	0
pret (instances: 8, vars: 150, dom: 2, rels: 100, arity: 3)						
wR(∗,2)C	22,677,416.3	151.63	153.04	4	0	0
wR(∗,3)C	2,129,528.0	88.79	92.32	4	0	0
wR(∗,4)C	2,129,528.0	275.04	286.58	4	0	0
GAC	13,024,912.3	40.48	41.78	4	**4**	0
maxRPWC	13,024,912.3	40.93	43.00	4	0	0

class, and constitutes another step towards empowering constraint solvers to solve problems without search, a target identified during the ModRef'09 Workshop. Stronger consistency almost always consistently reduces the number of nodes visited, but not the CPU time. When search with a given consistency property visits relatively few nodes, enforcing a stronger property on the same instance may be overkill and wasteful. This remark holds for wR(∗,2)C and wR(∗,3)C on `dag-rand`, but not for `rand-10-20-10` where wR(∗,4)C beats all tested algorithms.

Table 2 shows the results of the second group of benchmarks. On these problems, all tested algorithms visit few nodes. The time for enforcing wR(∗,m)C is wasted and increases with the value of m. As for the third group in Table 3, wR(∗,m)C visits fewer nodes than both GAC and maxRPWC for most of the instances, but is not able to outperform them in terms of CPU time.

We do not report the results of R(∗,m)C for the following reasons. For $m = 2$, R(∗,2)C and wR(∗,2)C are equivalent and the latter is significantly cheaper than the former. In general, wR(∗,m)C considers significantly fewer combinations of constraints than R(∗,m)C: it scales better than and outperforms R(∗,m)C.

The goal of our experiments is to evaluate different consistency properties under similar conditions. Our solver does not implement the advanced heuristics used in the Solver Competition. Hence, we cannot compare the CPU time in our experiments to that of the competition. Nevertheless, we achieve better CPU time with wR(∗,2)C on `dag-rand` than reported in the 2008 and 2009 competitions.

7. Related Work

Janssen et al. (1989) introduced *pairwise consistency* as a special case of m-wise consistency, which was proposed in the area of relational databases (Gyssens 1986). Pairwise consistency requires that every tuple in a relation can be extended to a tuple in every other relation. Pairwise consistency and R(∗,2)C are equivalent. The authors proposed to enforce this consistency property by enforcing arc-consistency on the dual CSP. Importantly, the authors also described an algorithm for removing the redundant edges from the dual CSP to avoid revising unnecessary relation pairs. We use their redundancy removal algorithm for wR(∗,m)C. While m-wise consistency is equivalent to R(∗,m)C, to the best of our knowledge, our work is the first to propose and evaluate an algorithm for enforcing it.

Jégou (1993) proposed hyper-k-consistency, which requires the tuples in every (k-1) relations to be extendible to every k^{th} relation. Generalizing the early work on local consistency for CSPs in (Montanari 1974; Mackworth 1977; Jégou 1993), Dechter and van Beek (1997) formalized *relational consistency* for non-binary CSPs in terms of *relational m-consistency* and *relational (i, m)-consistency*. Enforcing any of the above listed properties may require the addition of new constrains to the problem modifying its topology, which we avoid doing in our approach.

None of the above-listed approaches evaluates practical algorithms for enforcing the proposed properties.

Next, we describe more recent approaches to relational consistency that specify and evaluate the corresponding propagation algorithms.

Stergiou and Walsh (1999) studied arc consistency on three different encodings of non-binary CSPs (i.e., the hidden variable, dual, and double encodings). Samaras and Stergiou (2005) designed specialized arc-consistency algorithms for those encodings. Their arc-consistency algorithm for the dual encoding improves performance by grouping tuples that have the same supports, but yields filtering equivalent to pairwise consistency and R($*$,2)C. While it is specialized for pairs of relations, our proposed algorithm is parameterized and applies to any number of relations. Our algorithm can benefit from the tuple grouping of (Samaras and Stergiou 2005). Further, we avoid redundant checks as proposed in (Janssen et al. 1989), which is an improvement over the approach of (Samaras and Stergiou 2005).

Bessiere et al. (2008) provided detailed theoretical, algorithmic, and empirical studies of domain filtering consistencies for non-binary CSPs. The consistency properties that they studied do not modify the topology of the constraint network and are restricted to combinations of two relations. Further, they are stronger than GAC (which is relational (1,1)-consistency), but are weaker than pairwise consistency followed by GAC. Our work complements and extends their approach by considering combinations of an arbitrary number of constraints and updating the constraint definitions, thus providing stronger consistency properties. In our experiments, we compare our work against maxRPWC, which exhibits the best performance in their study.

Finally, we mention the consistency properties Conservative Path Consistency introduced in (Debruyne 1997) and the stronger property Conservative Dual Consistency introduced in (Lecoutre, Cardon, and Vion 2007), which do not alter the topology of the constraint graph. However, they are both restricted to binary CSPs and consider only three constraints at the same time.

8. Conclusions and Future Work

In this paper, we studied the relational consistency property R($*$,m)C, proposed a weaker variant of it, wR($*$,m)C, and presented a parameterized algorithm for enforcing it. Our algorithm operates by tightening the existing constraints, without adding new ones. To demonstrate its usefulness, we evaluated it against algorithms for GAC and maxRPWC. Our experiments showed that, by maintaining a stronger consistency, the performance of search can be improved by two orders of magnitude on many benchmark problems. Several instances were solved in a backtrack-free manner, hinting at the tractability of the corresponding problem class.

Our algorithm can be further improved by reducing redundant consistency checks, for example by grouping tuples (Samaras and Stergiou 2005) or exploiting complex residual supports (Likitvivatanavong et al. 2007; Lecoutre and Hemery 2007; Lecoutre et al. 2008). Other interesting avenues for future work are to exploit the tightness of the constraints to avoid considering ineffective combinations of relations and to design techniques that automatically identify the level of consistency necessary for a given problem.

Acknowledgments

The authors acknowledge the help of Kostas Stergiou and the feedback of anonymous reviewers. Experiments were conducted on the equipment of the Holland Computing Center at UNL. Shant Karakashian was partially supported by NSF CAREER Award #0133568, and Robert Woodward by an undergraduate research grant (UCARE) of the University of Nebraska-Lincoln and by a B.M. Goldwater Scholarship.

References

Bessiere, C.; Régin, J.-C.; Yap, R. H.; and Zhang, Y. 2005. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence* 165(2):165–185.

Bessiere, C.; Stergiou, K.; and Walsh, T. 2008. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence* 172:800–822.

Debruyne, R., and Bessière, C. 1997. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. of the 15 th IJCAI*, 412–417.

Debruyne, R. 1997. A Strong Local Consistency for Constraint Satisfaction. In *ICTAI 99*, 202–209.

Dechter, R., and van Beek, P. 1997. Local and Global Relational Consistency. *Theor. Comput. Sci.* 173(1):283–308.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Gyssens, M. 1986. On the Complexity of Join Dependencies. *ACM Trans. Database Systems* 11(1):81–108.

Janssen, P.; Jégou, P.; Nougier, B.; and Vilarem, M. 1989. A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In *IEEE WS on Tools for AI*, 420–427.

Jégou, P. 1993. On the Consistency of General Constraint-Satisfaction Problems. In *AAAI 1993*, 114–119.

Lecoutre, C., and Hemery, F. 2007. A Study of Residual Support in Arc Consistency. In *IJCAI 07*, 125–130.

Lecoutre, C.; Likitvivatanavong, C.; Shannon, S.; Yap, R.; and Zhang, Y. 2008. Maintaining Arc Consistency with Multiple Residues. *Constraint Programming Letters* 2:3–19.

Lecoutre, C.; Cardon, S.; and Vion, J. 2007. Conservative Dual Consistency. In *AAAI 07*, 237–242.

Likitvivatanavong, C.; Zhang, Y.; Shannon, S.; Bowen, J.; and Freuder, E. C. 2007. Arc Consistency During Search. In *Proc. of the 20 th IJCAI*, 137–142.

Mackworth, A. K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8:99–118.

Montanari, U. 1974. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Inf. Sciences* 7:95–132.

Samaras, N., and Stergiou, K. 2005. Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. *JAIR* 24:641–684.

Stergiou, K., and Walsh, T. 1999. Encodings of Non-Binary Constraint Satisfaction Problems. In *AAAI 1999*, 163–168.