A First Practical Algorithm for High Levels of Relational Consistency
Shant Karakashian, Robert J. Woodward, Christopher Reeson, Berthe Y. Choueiry, Christian Bessiere

To cite this version:

HAL Id: lirmm-00558132
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00558132
Submitted on 21 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A First Practical Algorithm for High Levels of Relational Consistency

Shant Karakashian¹, Robert J. Woodward¹, Christopher Reeson¹, Berthe Y. Choueiry¹ and Christian Bessiere²

¹Constraint Systems Laboratory, University of Nebraska-Lincoln, USA
²LIRMM-CNRS, University of Montpellier, France

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 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: \( \forall R_i \in \{ R_1, \cdots, R_m \}, R_i = \pi_{\text{scope}(R_i)}(\bigwedge_{j=1}^m R_j) \). 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 (V, D, C) where V is a set of variables, D is a set of domains, and C is a set of constraints or relations. We use the terms constraint and relation interchangeably. Each variable \( V_i \in V \) has a finite domain \( D_i \in D \), and is constrained by a subset of the relations in 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 \( \varphi \) a combination of m constraints that induce a connected component in the
null
Indeed, the same value for $A$ is enforced between $R_1$ and $R_2$ through $R_4$, and that for $C$ between $R_2$ and $R_3$ through $R_5$.

![Figure 1: Dual graph.](image)

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 $P$ is defined as the property of $P$ 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 $a \neq b \leq |C|$, $wR(*,a)C$ is strictly stronger than $wR(*,b)C$ on the same connected minimal dual graph of the CSP.

**Proof:** Let $\Phi_a$ and $\Phi_b$ be the set of combinations of $R(*,a)C$ and $R(*,b)C$, 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 $P_c$ with the three variables $V_1$, $V_2$, and $V_3$ and the three constraints: $C_{V_1,V_2} = C_{V_2,V_3} = C_{V_1,V_3} = \{\{0,1\},\{1,0\}\}$. Clearly, $P_c$ is $wR(*,2)C$ but not $wR(*,3)C$.

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

**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 $P_c$ 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 $\pi_{AB}(R_1) = \pi_{BC}(R_2) = \{\{1,1\},\{0,0\}\}$, and $\pi_{AC}(R_3) = \{\{0,1\},\{1,0\}\}$. 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 $\tau$ of a relation $R$ in a combination $\varphi$ of relations, denoted $S_{\tau,\varphi}$, is a set of tuples that verifies the condition: $\forall R_i \in \varphi \setminus \{R\} \exists \tau_i \in S_{\tau,\varphi}, \tau_i \in R_i$ 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 $\Phi$ be the set of all combinations of $m$ relations that induce connected components of the considered graph. We initialize the queue, $Q$, over which our algorithm operates, to all the combination-relations pairs $\{\varphi, R\}$ such that $\varphi \in \Phi$ and $R \in \varphi$. We have developed an algorithm, not reported here for lack of space, that computes $\Phi$ 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 $Q$ and $\Phi$, 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 $\Phi$ 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 $\varphi$. Notice that a relation $R'$ that appears in a combination with $R$ needs to
Algorithm 1: ProcessQueue enforces R(σ,m)C

```
Input: Q, Φ
Output: true if the problem is R(σ,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          Q ← Q \ {φ', R'}
12      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 Sτ,φ 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 P Dφ. The variables of P Dφ 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 P Dφ are binary, and enforce the equality of the shared scope of the relations in φ. A solution to P Dφ is Sτ,φ, the support set of τ in φ. The search stops at the first solution and returns Sτ,φ. 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 Rj that are consistent with a tuple τj of a relation Rj. 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 Rj and Rj and V s=scope(Rj) ∩ scope(Rj), the index tree IT Rj,vs returns for τj ∈ Rj all tuples τl ∈ Rj to which τj can be extended, that is π V s(τj) = π V s(τj). An index tree IT Rj,vs is a rooted tree, with a dummy root, where all leaves are at height |Vs|. The level of a node in the tree corresponds to a variable in Vs. The nodes are labeled with values of the variables in Vs. Each leaf node holds a list of pointers to tuples in Rj. Figure 2 shows an example of an index tree for the relation Rj and V s=\{A, B, C\}.

The tree is built as follows. The tuples of Rj are sequentially inserted in the tree. For a given tuple τj ∈ Rj, we consider π V s(τj). Traversing IT Rj,vs from the root, we match the value of a variable in π V s(τj) 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 π V s(τj). Otherwise, we add a new child node with the value of the variable in π V s(τj). When the variables in Vs are exhausted, we insert τj at the end of the list at the leaf node.

When searching for the tuples in Rj that are consistent with τj, we traverse the tree as explained above for π V s(τj). 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 \(O(|V_s|d)\) for time and \(O(|V_s|t)\) for space, where t is the number of tuples in the relation and d is the largest domain size of the variables in Vs. This bound is reached when each leaf node points to a single tuple. The time complexity of a query is \(O(|V_s|(d+t))\).

5.5 Improving the search for support

We propose two improvements to the search for support. When Sτ,φ is found, it is stored for the tuple-combination pair (τ, φ), and reused as long as every tuple in Sτ,φ 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 Sτ,φ is found, the support of every tuple τ' ∈ Sτ,φ can be directly set to be \((S_{τ,φ} \cup \{τ\}) \setminus \{τ'\}\), 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 \(O(d^k)\), 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 \(^eC_m\) and bounded by \(O(\text{minimum}(e^m, e^m))\). Below, we assume that \(m < \frac{e}{2}\).

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

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

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 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#Nodes</th>
<th>Time</th>
<th>Max time</th>
<th>#C</th>
<th>#F</th>
<th>#BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>modifiedRenault</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(instances: 50, vars: 15, dom: 42, rels: 174, arity: 10)</td>
<td>192.5</td>
<td>2.99</td>
<td>18.45</td>
<td>46</td>
<td>27</td>
<td>41</td>
</tr>
<tr>
<td>$wR(*,3)C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>82.5</td>
<td>7.55</td>
<td>12.43</td>
<td>50</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>$wR(*,4)C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>82.5</td>
<td>33.88</td>
<td>70.61</td>
<td>50</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>GAC</td>
<td>1,324,309.8</td>
<td>402.44</td>
<td>2,436.40</td>
<td>26</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>maxRPWC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2,110.8</td>
<td>305.37</td>
<td>2,886.63</td>
<td>31</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>rand-8-20-5 (instances: 20, vars: 20, dom: 5, rels: 18, arity: 8)</td>
<td>941.3</td>
<td>1,162.22</td>
<td>2,979.77</td>
<td>16</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>$wR(*,2)C$</td>
<td>0.0</td>
<td>0.20</td>
<td>0.24</td>
<td>20</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>$wR(*,3)C$</td>
<td>0.0</td>
<td>0.21</td>
<td>0.25</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>$wR(*,4)C$</td>
<td>0.0</td>
<td>0.18</td>
<td>0.22</td>
<td>20</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>GAC</td>
<td>210.0</td>
<td>9.34</td>
<td>13.17</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>maxRPWC</td>
<td>0.0</td>
<td>2.45</td>
<td>6.81</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>rand-10-20-10 (instances: 20, vars: 20, dom: 10, rels: 5, arity: 10)</td>
<td>0.0</td>
<td>27.21</td>
<td>31.51</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>$wR(*,2)C$</td>
<td>0.0</td>
<td>37.75</td>
<td>40.15</td>
<td>25</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>$wR(*,3)C$</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$wR(*,4)C$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>maxRPWC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>dag-rand (instances: 25, vars: 23, dom: 3, rels: 16, arity: 15)</td>
<td>66,905.8</td>
<td>12.65</td>
<td>72.43</td>
<td>18</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$wR(*,2)C$</td>
<td>5,578.4</td>
<td>24.40</td>
<td>252.07</td>
<td>18</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>$wR(*,3)C$</td>
<td>127.5</td>
<td>11.47</td>
<td>65.89</td>
<td>19</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>GAC</td>
<td>30,993,972.2</td>
<td>602.91</td>
<td>2,758.48</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>maxRPWC</td>
<td>11,408,274.4</td>
<td>298.69</td>
<td>1,348.66</td>
<td>17</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>aim-100 (instances: 24, vars: 100, dom: 2, rels: 154, arity: 3)</td>
<td>2,670.2</td>
<td>35.51</td>
<td>71.98</td>
<td>12</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>$wR(*,2)C$</td>
<td>580.2</td>
<td>35.91</td>
<td>79.96</td>
<td>14</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$wR(*,3)C$</td>
<td>443.8</td>
<td>240.13</td>
<td>837.54</td>
<td>14</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>GAC</td>
<td>1,876,247.6</td>
<td>542.48</td>
<td>2,144.74</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>maxRPWC</td>
<td>842,488.8</td>
<td>414.05</td>
<td>1,583.34</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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. 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

---

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#Nodes</th>
<th>Time</th>
<th>Max time</th>
<th>#C</th>
<th>#F</th>
<th>#BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand-3-20-20</td>
<td>10,494.1</td>
<td>1,565.43</td>
<td>3,005.42</td>
<td>28</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>rand-3-20-20</td>
<td>6,498.5</td>
<td>1,107.93</td>
<td>2,769.95</td>
<td>35</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rand-5-20-20</td>
<td>63,627.5</td>
<td>171.73</td>
<td>418.31</td>
<td>49</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>dag-rand</td>
<td>31,202.3</td>
<td>1,220.33</td>
<td>3,453.52</td>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pre</td>
<td>23,677.416.3</td>
<td>151.63</td>
<td>153.04</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pre</td>
<td>2,129,528.0</td>
<td>88.79</td>
<td>92.32</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pre</td>
<td>2,129,528.0</td>
<td>275.04</td>
<td>286.58</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>maxRPWC</td>
<td>13,024,912.3</td>
<td>40.48</td>
<td>41.78</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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(2,2)C and wR(3,3)C on dag-rand, but not for rand-10-20-10 where wR(4,4)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 R(m,m)C is wasted and increases with the value of m. As for the third group in Table 3, wR(m,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,m)C for the following reasons. For m = 2, R(2,2)C and wR(2,2)C are equivalent and the latter is significantly cheaper than the former. In general, wR(m,m)C considers significantly fewer combinations of constraints than R(m,m)C: it scales better than and outperforms R(m,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(2,2)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(2,2)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,m)C. While m-wise consistency is equivalent to R(m,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 kth 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.

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(2,2)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,m)C. While m-wise consistency is equivalent to R(m,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 kth 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.
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 $R(\ast,2)C$. 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 $R(\ast,m)C$, proposed a weaker variant of it, $wR(\ast,m)C$, 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


