Exploring Parameterized Relational Consistency

Shant K. Karakashian, Robert J. Woodward and Berthe Y. Choueiry

Constraint Systems Laboratory
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Email: {shantk|rwoodwar|choueiry}@cse.unl.edu

TR-UNL-CSE-2009-0009

August 27, 2009

Abstract

Consistency properties and algorithms for achieving them are at the heart of the success of Constraint Programming. For non-binary Constraint Satisfaction Problems (CSPs), the relational-consistency property R(i,j)C of [Dechter and van Beek 1997] may add new non-binary constraints to the constraint network, thus modifying its topology. The domain-filtering properties of [Bessière *et al.* 2008] filter the domains of the variables and leave the constraints unchanged but are restricted to combinations of two constraints. We restate the property of m-wise consistency [Gyssens 1986; Jégou 1993] as relational ($*$,m)-consistency, R($*$,m)C. R($*$,m)C ensures that any tuple in a relation is consistent in every combination of m constraints. The main contributions of this document are the design of an algorithm for enforcing R($*$,m)C and the evaluation of its effectiveness in a search procedure solving CSPs. This document thus establishes the usefulness in practice of higher consistency levels in non-binary CSPs.

Contents

1 Introduction

Local consistency techniques are the heart of the success of Constraint Programming and perhaps best distinguish this field from other scientific disciplines that study the same combinatorial problems. In continuation of the properties and algorithms introduced in [Waltz 1975; Montanari 1974; Mackworth 1977; Freuder 1985], Dechter and van Beek [1997] defined the concept of *relational consistency* to address the consistency properties of non-binary Constraint Satisfaction Problems (CSPs). They defined the consistency property of relational m-consistency (RmC), involving every combination of m constraints in the CSP, and the more relaxed property of relational (i, m)-consistency (R(i, m)C), involving every combination of i variables. In practice, enforcing RmC or R(i, m)C may require the generation of $\mathcal{O}(n^i)$ new non-binary constraints, where n is the number of variables in the CSP.

Another research direction focused on the effect of consistency properties on the domains of the variables [Mohr and Masini 1988; Bessière *et al.* 2005; Lhomme and Régin 2005; Bessière *et al.* 2008; Cheng and Yap 2004]. *Domain filtering* has the advantage of reducing the search space explored for solving the CSP. While most work considered constraints *individually* (GAC), Bessière et al. [2008] studied the effects of combinations of pairs of constraints.

In this document we introduce a special form of relational consistency, which we call R($*$,m)C and which operates on every combination of m constraints. Unlike RmC and R(i, m)C, R($*$,m)C does *not* add new constraints to the CSP and, thus, keeps the topology and width of the network unchanged. Instead, it operates on the relations defining the constraints, filtering them to remove inconsistent tuples. In comparison to R(i,m)C, the '$*$' in R($*$,m)C is used to indicate that the property affects only 'those variables that are in the scope of an existing constraint, whatever the size of the scope is.' More formally, we define R($*$,m)C to ensure that every tuple in a relation can be extended to a partial solution over the variables in every set of m constraints that is consistent with those constraints. R($*$,m)C is semantically equivalent to relational m-wise consistency studied in [Gyssens 1986; Jégou 1993][1]. However, neither paper evaluated or even proposed practical algorithms for implementing relational m-wise consistency. We choose to use the notation R($*$,m)C instead of the notation m-wise consistency simply and purely for the sake of situating this consistency property in the context of the

[1] [Janssen *et al.* 1989] present relational pairwise consistency as requiring that the 'overlap' of every pair of constraints can be 'extended' to the constraints in the pair. However, it is easy to prove that relational pairwise consistency and R($*$,2)C are equivalent.

terminology R(i, m)C familiar to the CP community. The contributions of our document are as follows:

1. The (re-)definition of a new (parametric) relational consistency property, R($*$,m)C, that does not modify the topology of the constraint network.

2. The design of an algorithm, along with its data structures, for enforcing R($*$,m)C.

3. Similarly to [Bessière *et al.* 2008], the integration of our algorithm in a backtrack search procedure with full lookahead for solving non-binary CSPs.

4. The evaluation of the cost (in terms of CPU) and effectiveness (in terms nodes visited as a measure of pruning power) of the resulting search procedure on randomly generated and benchmark problems.

We also identify ways to improve our algorithm in the future.

In summary, we establish in this document that higher consistency levels are feasible and advantageous in practice.

This document is structured as follows. Section 2 reviews the definition of non-binary CSPs. Section 3 defines R($*$,m)C. Section 4 describes our algorithm for enforcing R($*$,m)C. Section 5 discusses our experimental results. Section 6 discusses future work and concludes this report.

2 Basic Definitions

A Constraint Satisfaction Problem (CSP) is defined by the tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ set of domains, and $\mathcal{C}$ set of constraints. Each variable $V_i \in \mathcal{V}$ has a finite domain $D_i \in \mathcal{D}$, and is constrained by a subset of the constraints in $\mathcal{C}$. For a given constraint $C_i \in \mathcal{C}$, $vars(C_i)$ denotes the scope of the constraint. Every constraint C_i is associated with a relation R_i, which gives the allowed combinations of values for the variables in $vars(C_i)$. Such a combination of values, said to be consistent with C_i, is a tuple $\tau \in R_i$ of size $|vars(C_i)|$. In this report, we use constraints (C_i) and relations (R_i) interchangeably. A solution for the CSP is a tuple made of one value per variable such that all the constraints are satisfied, i.e. the projection of the solution tuple on the scope of each constraint C_i is consistent with R_i.

In this report, we denote by a *combination φ of m constraints* a set of m constraints such that primal graph induced by the constraints in the set is connected. Note that the primal graph of a non-binary CSP is the graph whose nodes are the variables in the scope of the constraints and whose edges connect every two variables that appear in the scope of one or more constraints [Dechter 2003]. Further, we denote by ζ is the set of all possible combinations φ of size m in a given CSP. Finally, π and $\bowtie$ denote the relational operators project and join.

3 Definition of R($*$,m)C

Although the definition of R($*$,m)C is intuitive and obvious, we state it below using the definition format of R(i,m)C of [Dechter 2003]:

Definition 3.1 *A set of m relations $\mathcal{R} = \{R_1, \cdots, R_m\}$ 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 $\bigcup_{R_j \in \mathcal{R} \setminus R_i} vars(R_j)$ in an assignment that satisfies all the relations in $\mathcal{R}$ simultaneously. A network is R($*$,m)C iff every set of m relations is R($*$,m)C.*

R($*$,m)C can thus be enforced by filtering the existing relations using the following operation on each combination of m relations $\{R_1, \cdots, R_m\}$ and without introducing to the CSP any relation whose scope was not already constrained in the original CSP:

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

Expression (1) gives us an obvious algorithm for enforcing R($*$,m)C. Even if each join is computed only once and then its tuples filtered iteratively, the space requirement of such an operation is too prohibitive to be of any usefulness in practice use.

Once R($*$,m)C is enforced on a constraint network, variable domains can subsequently be filtered (i.e., domain filtering) by simple projection of the filtered relations on the domains of the variables. Unlike GAC, we do not need to loop between the filtering of the domains and that of the the constraints because any value for a variable that appears in any relation in the network, necessarily appears in all of them. R($*$,m)C is related to other consistency properties as follows:

1. As stated in the introduction, R($*$,m)C is equivalent to relational m-wise consistency proposed in the area of Relational Databases [Gyssens 1986; Jégou 1993].

2. R($*$,m)C is obviously equivalent to (1,$m-1$)-consistency on the dual CSP.

3. If all pairs of relations in the CSP overlap on exactly one variable, then, R($*$,2)C and GAC have the same 'domain-filtering power' (proof is similar to that of Theorem 2 of [Bessière *et al.* 2008]). Furthermore, on a normalized binary CSP, where the constraints on the same pair of variables are combined, R($*$,2)C and AC have the domain-filtering power (similarly to what is stated in [Bessière *et al.* 2008]). Clearly, R($*$,2)C cannot be beneficial and should not be used in those two situations, as it would only incur computational overhead.

Below we discuss the relationship between relational m-consistency (RmC) of [Dechter and van Beek 1997] and R($*$,m)C. For a given set $\{R_1, \cdots, R_m\}$ of m relations RmC requires the projection of the joined relations on all subsets of size $|\bigcup_{i=1}^{m} vars(R_i)|-1$ of $A \subseteq \bigcup_{i=1}^{m} vars(R_i)$. Hence, every subset introduces a new constraint, except those that have the same scope of existing constraints. Because R($*$,m)C projects the join on the scope of each of its original relations, no new constraints are added. Although R($*$,m)C has the favorable property that no new constraints are introduced, it is weaker, in terms pruning power and consistency, than relational m-consistency[2].

Theorem 3.1 R($*$,m)C is a weaker consistency, in terms of pruning power and consistency, than relational m-consistency

Proof: Consider a CSP $\mathcal{P}$, and let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the same problem after enforcing RmC and R($*$,m)C on $\mathcal{P}$, respectively. We consider a partial assignment τ over some of the variables of $\mathcal{P}$, $vars(\tau)$, that is consistent with the constraints of $\mathcal{P}_{rmc}$ and prove that it must necessarily be consistent with the constraints in $\mathcal{P}_{r*mc}$. Let's 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 $\mathcal{P}_{r*mc}$ such that $\tau \notin \pi_{vars(\tau)}(R_{x*})$. Given the definitions of RmC and R($*$,m)C, there must exist one relation in $\mathcal{P}_{rmc}$ (which adds many new constraints to the problem) that has the same scope of a relation in $\mathcal{P}_{r*mc}$ (which does not add new constraints to the problem). Thus, $\mathcal{P}_{rmc}$ must have a relation R_x such that $var(R_{x*})$=$var(R_x)$. Given that τ is a consistent partial solution in $\mathcal{P}_{rmc}$, then $\tau \in \pi_{vars(\tau)}(R_x)$. $\tau \in \pi_{vars(\tau)}(R_x)$ and $\tau \notin \pi_{vars(\tau)}(R_{x*})$ is impossible because joining more relations of $\mathcal{P}_{rmc}$ and projecting them on the

[2]Note that relational m-consistency of of [Dechter and van Beek 1997] and hyper-m-consistency of [Jégou 1993] are most likely equivalent, the proof being outside the scope of this document.

same scope $vars(\tau)$ cannot possibly introduce more tuples. Thus, we reach a contradiction and R($*$,m)C is not a stronger consistency than RmC.

Below we provide an example that shows that RmC can be stronger than R($*$,m)C. 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:

$$C_{V_1,V_2} = C_{V_2,V_3} = C_{V_3,V_4} = C_{V_4,V_1} = \{\langle 0,0\rangle, \langle 1,1\rangle\} \quad (2)$$

Let $\mathcal{P}_{rmc}$ and $\mathcal{P}_{r*mc}$ be the problems after enforcing RmC and R($*$,m)C on $\mathcal{P}$, respectively. The partial assignment $\langle (V_1,0),(V_3,1)\rangle$ is consistent in $\mathcal{P}_{r*mc}$, because it is consistent with the the constraints in $\mathcal{P}_{r*mc}$, which are identical to the ones $\mathcal{P}$ for all values of m. However, this partial assignment violates the constraint $C_{V_1,V_3} = \{\langle 0,0\rangle, \langle 1,1\rangle\}$ in $\mathcal{P}_{rmc}$. In this case, RmC is a stronger consistency than R($*$,m)C.

In conclusion, R($*$,m)C is a weaker consistency than RmC.

□

Corollary 3.1 R($*$,m)C is sound and does not eliminate any solution.

Because R($*$,m)C is a weaker consistency than relational m-consistency, its soundness follows from the soundness of relational m-consistency.

4 An Algorithm for R($*$,m)C

In this section we describe our algorithm for enforcing the R($*$,m)C property on a CSP. The algorithm has three main components: initializing the constraint queue (Algorithm 1), processing the constraint queue (Algorithm 2), and finding and maintaining the support structure (FINDSUPPORT). This last function is used in both Algorithm 1 and Algorithm 2. Enforcing R($*$,m)C is achieved by calling Algorithm 2 on queue returned by Algorithm 1.

4.1 The set of combinations of m constraints

Given a CSP problem, we first generate the set ζ of all combinations φ_i of m constraints, such that the graph induced by φ_i is a connected graph. There is potentially a factorial number of such combinations in a constraint network. We have developed an algorithm, not reported here for lack of space, that computes all the connected combinations of m constraints in CSP while exploiting the topology of the dual graph of the CSP. That algorithm generates every connected component once while not generating any non-connected component.

4.2 The *Last* data structure

We achieve the R($*$,m)C property when every tuple τ of every relation R_i in every combination φ of m relations can be 'extended' successfully to all the $(m-1)$ remaining relations in φ, that is all tuples have the same values for the common variables. We say that the set of $(m-1)$ tuples that 'extends' τ to the constraints in φ is the 'support' of R_i's τ in φ. When, in at least one combination, no support can be found for a tuple, then the tuple is deleted. In order to avoid repeatedly rediscovering this support when enforcing the R($*$,m)C property on the CSP, and similarly to [Bessière *et al.* 2005; 2008], we use a data structure $Last((\tau, R_i), \varphi)$, which is a list of pointers to the tuples supporting $\tau \in R_i$ in the $(m-1)$ remaining constraints in φ. This list is initialized to `nil`. When a support is first found for τ, this list points to the $(m-1)$ supporting tuples. The support is *valid* as long as none of $(m-1)$ supporting tuples is deleted. The algorithms below focus on identifying, using, and updating such supports.

Note that the data structure $Last((\tau, R_i), \varphi)$ is used to remember the last current solution that supports $\tau \in R_i$ in the combination φ. When any of the supporting tuples is deleted, the search for a new support is restarted from the longest consistent partial solution. Thus, $Last()$ plays a different role than the data structure with the same name in GAC/AC-2001 algorithms. In fact the role of $Last$ in GAC/AC-2001 algorithms is fulfilled by our data structure *IndTree* introduced in Section 4.6.

4.3 Initializing the constraints queue

Algorithm 1 considers each tuple τ in each relation R in each combination of constraints $\varphi \in \zeta$, and tries to extend the tuple to the remaining relations in the combination φ using FINDSUPPORT. If no support is found for τ, then it is deleted from R. (As we explain in Section 4.6, deleting a tuple is achieved by flagging it as such in the table that stores the tuples of the relation.) Further, the relations that appear in any combination φ' of m relations containing R and such that $\varphi' \neq \varphi$ are added to the constraint queue as their tuples may be supported by the deleted tuple τ.

4.4 Processing the constraint queue

The initialization phase deletes some tuples from the constraints, but does not fully enforce the R($*$,m)C property. Some tuples deleted by Algorithm 1 could

Algorithm 1: INITIALIZE-$\mathcal{Q}$, initializes the queue.

```
   Input: ζ
   Output: Q: queue of constraints
 1 foreach φ ∈ ζ do
 2 |   foreach R ∈ φ do
 3 |   |   deleted ← false
 4 |   |   foreach τ ∈ R do
 5 |   |   |   support ←FINDSUPPORT((τ, R), φ)
 6 |   |   |   if support = false then
 7 |   |   |   |   DELETE(τ)
 8 |   |   |   |   if R = ∅ then return false
 9 |   |   |   |   deleted ← true
10 |   |   |   end
11 |   |   end
12 |   |   if deleted then  foreach φ' ∈ (ζ \ {φ}) do
13 |   |   |   if R ∈ φ' then  Q ← Q ∪ (φ' \ {R})
14 |   |   end
15 |   end
16 end
17 return Q
```

have been in the support of some other tuples. Hence some deletions may leave some tuples without any support. Therefore, we should seek new supports for these tuples, and, if none is found, we should delete them. The procedure PROCESSQUEUE given in Algorithm 2 revises every relation in the queue to ensure that all their tuples are properly supported in each combination of m constraints where the relation appears.

4.5 Finding a support

The predicate function VALIDSUPPORT((τ, R), φ) examines the data structure *Last*((τ, R), φ) to determine whether or not there is a 'valid' support for $\tau \in R$ in φ. A valid support exists when the list of pointers is not empty and when none of the $m-1$ tuples supporting $\tau \in R$ has been flagged 'deleted.' If a valid support is found, then the predicate returns $true$, otherwise it returns $false$.

In order to find a support, of $(m-1)$ tuples, for a tuple τ of a relation R

Algorithm 2: PROCESSQUEUE, delete tuples that have lost their support.

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

```
1  while Q ≠ ∅ do
2      R ← POP(Q)
3      deleted ← false
4      foreach φ s.t. R ∈ φ do
5          foreach τ ∈ R do
6              support ←FINDSUPPORT((τ, R), φ)
7              if support = false then
8                  DELETE(τ)
9                  if R = ∅ then return false
10                 deleted ← true
11             end
12         end
13     end
14     if deleted then foreach φ' ∈ (ζ \ φ) do
15         if R ∈ φ' then Q ← Q ∪ (φ' \ {R})
16     end
17 end
18 return true
```

in a combination φ, we conduct a depth first search with partial look-ahead (à la forward checking) on the dual CSP induced by the m relations in φ and in which the assignment $R \leftarrow \tau$ is made. A solution to that dual CSP provides a support for $\tau \in R$, which is used to initialize or update *Last*$((\tau, R_i), \varphi)$. One important functionality to implement the look ahead is the ability to determine that a tuple $\tau_i \in R_i$ can be matched with some tuple in R_j, where R_i and R_j are two 'variables' in the dual CSP. In Section 4.6, we propose an index tree data-structure, *IndTree*, to facilitate matching the tuple τ_i in R_j.

One could further improve the runtime performance by updating the support of each tuple τ_i in *Last*$((\tau, R_i), \varphi)$ with the set of tuples returned by the search procedure, from which τ_i is removed and to which τ is added.

Algorithm 3: FINDSUPPORT, finds a support for a tuple.

Input: $(\tau, R_i), \varphi$
if VALIDSUPPORT((τ, R),φ) **then return** $true$ **else**
  Last($(\tau, R_i), \varphi$) $\leftarrow$ SEARCH(φ, $R_i \leftarrow \tau$)
end
return $Last((\tau, R_i), \varphi)$

4.6 Matching a tuple in a relation

We say that $\tau_i \in R_i$ is matched in R_j if we can find a non-deleted tuple τ_j in R_j such that the variables in $vars(R_i) \cap vars(R_j)$ have the same assignments in τ_i and τ_j. The performance of matching (or finding a support) for a tuple $\tau_i \in R_i$ in another relation R_j (where $vars(R_i) \cap vars(R_j) \neq \emptyset$) is important in practice. Below, we introduce a new 'index data-structure' to facilitate this operation.

We assume that the relations are implemented as tables of consistent tuples (i.e., supports) and that the order of the tuples is fixed. We also assume that each table includes a column *del* to indicate that the tuple is deleted (1) or not (0). For each relation R_x and for each subset of the scope of R_x, $scope_o$, for which R_x overlaps with another relation in the problem, we build a tree structure *IndTree*($scope_o$,R_x), where $scope_o$ is lexicographically sorted, as follows. The root of the tree is a dummy node. Each level in the tree corresponds to a variable in $scope_o$ following the lexicographic order. Each node in a given level corresponds to a value that the variable at that level has in the relation. All the nodes at level 1 are connected to the root node. At any given level, a node is connected to a node at the preceding level *iff* the two corresponding variable-value pairs appear in a tuple in the relation R_x. Thus, we have a one-to-one correspondence between a path in the tree and the projection of a tuple in R_x on $scope_o$. Finally, each leaf is annotated with a list of pointers to the originating tuples in R_x. At the construction stage, those pointers reflect the order of the tuples in the relation. Figure 1 illustrates such a structure. For a CSP with e non-binary constraints and maximal constraint arity k, we have a maximum of $\mathcal{O}(e^2)$ such structures. Each structure has $\mathcal{O}(d^{(k-1)})$ nodes and takes $\mathcal{O}(d^{(k-1)})$ effort to build (i.e., linear in the number of tuples in R_x).

In order to locate a support for a given tuple τ in a relation R_x, we traverse the tree *IndTree*($scope_o$,R_x), with $scope_o$= $vars(\tau) \cap vars(R_x)$, from the root down to a leaf following the nodes corresponding to the values in $\pi_{scope_o}(\tau)$. If, at any level, no tree node can be found with the corresponding value in τ, we conclude

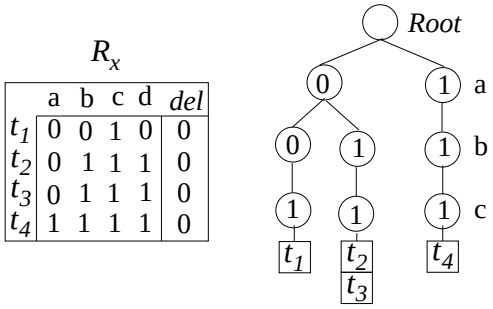

Figure 1: Example of *IndTree*($scope_o$,R_x) where $vars(R_x)$={a,b,c,d} and $scope_o$={a,b,c}.

that τ does not have a support in R_x. If we reach a leaf, the annotation at the leaf gives a list of pointers to the tuples in R_x that match with τ.

We implemented an additional (optional) feature for the annotations. Every time a tuple is deleted from a relation, all the annotations where it appears are accessed, and the corresponding pointers are moved to the tail of the annotation list. In this context, whenever the first pointer in an annotation points to a deleted tuple, it becomes obvious that no other tuple in the annotation can be 'alive' and finding a support for τ in R_x returns failure. This optional feature has showed improvements in some special instances in our experiments in Section 5. We refer to this feature as *index updating*. Note that when index updating is enabled, our data structure directly access the consistent tuple alive more efficiently than the data structure *Last* of in GAC/AC-2001 algorithms [Bessière *et al.* 2005].

4.7 Complexity Analysis

The time complexity of our algorithm is dominated by the PROCESSQUEUE, hence we omit the initialization phase from the analysis. We assume uniform domain size d for all variables, uniform arity k for all constraints, and uniform number of tuples t in each constraint. We denote by e the number of constraints ($e = |\mathcal{C}|$), and by δ the number of combinations of constraints (δ=$|\zeta|$). The number of combinations is bounded by above by $\binom{e}{m}$, but this bound is reached only for very dense problems (complete graphs). In practice, the number of combinations is much less than this upper bound, therefore we use δ. As a reminder, our algorithm for generating all connected combinations of m constraints exploits the structure of the dual graph and does *not* generate combinations that are not connected, thus, the upper limit.

Algorithm 2 has three nested loops: the first loop iterates over all the relations in the queue; the second loop iterates over each combination where a relation appears; and the third loop iterates over each tuple in the relation. The innermost loop iterates $\mathcal{O}(t)$ times, and the middle loop at most δ times. As for the outermost loop, every time a tuple is deleted, at most $\mathcal{O}(e)$ relations are queued in Line 15 of Algorithm 2. Since the condition in Line 14 is satisfied at most once for any tuple in a relation, and since there are $t \times c$ tuples, then the loop in Line 1 of Algorithm 2 iterates at most $\mathcal{O}(t \cdot e)$ times.

When FINDSUPPORT is called for a tuple in Line 6 of Algorithm 2 and if the tuple has not lost its support, then it costs only $\mathcal{O}(m)$ to check the existence of a valid support in Line 1 of Algorithm 3. Let β be the cost of FINDSUPPORT and α be the number of times FINDSUPPORT is called in the case when the tuple has lost its support. Multiplying the costs of the nested loops we get:

$$\mathcal{O}(t^2 e^2 \delta m + \alpha\beta) \tag{3}$$

When a tuple does not have a support, Line 2 of Algorithm 3 is executed. Finding a support for a tuple is finding a matching tuple from each of the m-1 constraints in the combination. The worst-case time complexity of this operation is $\mathcal{O}(t^{m-1})$. Using the index-tree structure has the same complexity, because deleted tuples must be discarded. However, in practice, the index-tree structure exploits the selectivity of the relations and demonstrates much improved performance. As the arity of the relations increases, the selectivity of the relations also increases, and we observe better performance of our algorithm for problems with high constraint arity in the experiments (see Section 5). Therefore, $\beta = \mathcal{O}(t^{m-1})$.

The number of times a tuple τ can lose a support is bounded by the number of tuples that can participate in any support for τ. There are $m - 1$ constraints in a combination that make the support for a given τ, and each has t constraints. Hence the number of times τ loses support is $\alpha = \mathcal{O}(tm)$. Substituting α and β in Expression (3) we get: $\mathcal{O}(t^2 e^2 \delta m + m t^m)$.

The space complexity of the algorithm is dominated by the space requirement for the indexes constructed on the constraints. The space complexity of each index is $\mathcal{O}(t)$, that is the number of tuples in the relation, since there is a pointer to each tuple in the constraint. The number of nodes in the tree can be at most $d \times scope_o$. There are $\mathcal{O}(e)$ index trees, therefore the space complexity is: $\mathcal{O}(e \cdot t)$.

4.8 Integration with backtrack search

Our backtrack search mechanism for non-binary CSPs implements a full lookahead schema that maintains R($*$,m)C. The algorithm proceeds by assigning a value x to variable V_i taken from its domain, it then removes from all the relations R_i such that $V_i \in vars(R_i)$ the tuples that do not have x for V_i. Then, each relation R_i that has lost any tuples is processed as follows. For every combination φ such that $R_i \in \varphi$, every relation $R' \in \varphi$, $R' \neq R_i$ is added to the constraint queue. Then the queue is passed to Algorithm 2 to propagate the effect of those deletions. Finally, all updated relations are projected on the variables' domains for domain filtering.

5 Experimental Results

Our approach was motivated by an online tool for playing Minesweeper[3] where the puzzle is modeled as a CSP and various propagation algorithms are developed to support the user in solving the puzzle [Bayer *et al.* 2006]. We have used this puzzle as a tool to 'demystify' Constraint Programming to the general public and to illustrate to Computer Science students the usefulness of consistency properties and the operation of propagation algorithms.

5.1 Experimental Setup

We evaluated our algorithm on several benchmark problems[4] and randomly generated instances using the Model B generator of [Stergiou 2009]. Regarding the choice of benchmarks, Table 5.1, we make the following comments:

- The Renault benchmarks are the hardest used in the literature. We solve 46 out of 50 instances. Prior publications reported only 27 solved instances.
- Positive table constraints benchmarks have very large tables.
- Boolean benchmarks were chosen because the initial inspiration for our research was Minesweeper.

[3] http://minesweeper.unl.edu

[4] Renault configuration, Positive Table Constraints, and Boolean CSPs all taken from http://www.cril.univ-artois.fr/ lecoutre/research/benchmarks

- Random instances were chosen to compare the performance of the algorithms with increasing arity.

All benchmarks are hard, with large constraint arity and very large relations.

Table 1: Benchmark problems.

Name	$\|\mathcal{V}\|$	Domain size	$\|\mathcal{C}\|$	Arity	Number of tuples
Renault	[108, 111]	[2, 42]	[147, 159]	[2, 10]	[3, 48721]
`rand-8-20-5`	20	5	18	8	[77512, 78726]
`rand-10-20-10`	20	10	5	10	10,000
`aim-50`	50	2	[75, 279]	[2, 3]	[2,7]
`aim-100`	100	2	[155, 562]	[2, 3]	[2,7]
`aim-200`	200	2	[312, 1157]	[2, 3]	[2,7]
Random, Model B	20	10	5	[5, 12]	10000

All experiments were executed on a 2.4GHz Quad-Core AMD Opteron machine with 32GB of memory. Below, we discuss the performance of solving the CSPs with backtrack search while maintaining the properties GAC, maxRPWC [Bessière *et al.* 2008], and R($*$,m)C in a full-lookahead schema during search. As a reminder:

Definition 5.1 (Max Restricted Pairwise Consistency [Bessière *et al.* 2008]) *A non-binary CSP is Max Restricted Pairwise Consistent (maxRPWC) iff* $\forall V_i \in \mathcal{V}$ *and* $\forall x \in D_{V_i}$, $\forall C_j \in \mathcal{C}$ *where* $V_i \in vars(C_j)$, $\exists \tau \in C_j$ *such that* $\pi_{V_i}(\tau) = a$, τ *is valid and* $\forall C_l \in \mathcal{C}$ *(*$C_l \neq C_j$*), s.t.* $vars(C_i) \cap vars(C_l) \neq \emptyset$, $\exists \tau' \in C_l$, *s.t.* $\pi_{vars(C_j) \cap vars(C_l)}(\tau) = \pi_{vars(C_j) \cap vars(C_l)}(\tau')$ *and* τ' *is valid. In this case,* τ' *is said to be pairwise-support of* τ.

We implemented GAC2001/3.1 [Bessière *et al.* 2005], maxRPWC-1 [Bessière *et al.* 2008], and our algorithms R($*$,m)C and R($*$,m)Ci, respectively without and with the index updating scheme described in Section 4.6. We use dynamic variable ordering with the dom/deg ordering heuristic (with static degree). To measure the performance of the search, we report the CPU time in seconds and the number of nodes visited (#NV) for finding the first solution.

5.2 Renault Benchmarks

Our first experiment compared GAC, maxRPWC, R($*$,m)C, and R($*$,m)Ci on the Renault configuration problems. The set has 50 CSP instances that have between 108 and 111 variables, 147 and 159 constraints, largest domain size 42, and maximum arity of 10. We set the time limit to 20 minutes. The results are shown in Table 2. 'Completed' gives the number of instances solved within 20 minutes. Nodes visited (#NV) and CPU time (Time) in seconds are average over those 18 instances that were completed by all of the algorithms. The maximum time is the largest time taken by an algorithm for the 18 instances completed by all algorithms. 'Fastest' gives the number of times a given algorithm finished first among the four tested. As it can be seen from the results, R(*,2)C significantly improves

Table 2: Results on the Renault benchmark.

Algorithm	**#NV**	**Time**	**Maximum time**	**Completed**	**Fastest**
GAC	300,195.33	61.63	560.16	21	19
maxRPWC	1,140.61	118.01	253.24	29	0
R(*,2)C	**100.28**	**11.60**	**15.85**	**46**	**28**
R(*,2)Ci	**100.28**	16.96	29.43	**46**	0

the performance for solving the 18 instances solved by all algorithms and are able to solve 46 out of the 50 instances of this difficult benchmark.

R($*$,2)C's improved performance with respect to maxRPWC's is best explained by considering the number of nodes visited. R($*$,2)C did more filtering than maxRPWC, which allowed it to solve most of the problems almost backtrack free. Both R($*$,2)C and maxRPWC consider combinations of two constraints, and they only differ in that R($*$,2)C actually tightens the constraints. We conclude that a slightly larger investment in the pruning effort is rewarded by a significant reduction of the exponential search effort, thus making it possible to solve more problems within the same time limit.

5.3 Positive Table Constraints

Our next experiment was the 'Positive Table Constraints' benchmark, which has two sets of problems. Here, we set the time limit to three hours. Table 3 shows the results on the first set, which has 20 unsatisfiable instances of 20 variables,

domain size 10, with 5 constraints of arity 10. Table 4 shows the results on the

Table 3: Results on the Positive Table Constraints `rand-8-20-5` (all unsatisfiable).

Algorithm	#NV	Time	Maximum time	Completed	Fastest
GAC	210.10	8.19	11.55	20	0
maxRPWC	**0.00**	1.70	4.51	20	0
R(*,2)C	**0.00**	**0.07**	**0.10**	20	20
R(*,2)Ci	**0.00**	0.09	0.13	20	0

second set, which has 20 satisfiable instances, of 20 variables, domain size 5, with 18 constraints, and arity 8. The number of nodes visited (#NV) and CPU time (Time) in seconds are averaged over the 20 instances in Table 3, and over the instances completed by both GAC and R($*$,2)C in Table 4. 'Maximum time' is the largest time spent on any instance by an algorithm. 'Fastest' gives the number of times a given algorithm finished first among the four tested. Empty cells in the tables below indicate that the experiment did not complete in the allocated time limit.

Table 4: Results of the Positive Table Constraints `rand-10-20-10` (all satisfiable).

Algorithm	#NV	Time	Maximum time	Completed	Fastest
GAC	60,273.27	3,956.59	10,072.60	15	2
maxRPWC	-	-	-	0	0
R(*,2)C	**1,552.11**	2,901.71	7,210.45	18	2
R(*,2)Ci	**1,552.11**	**2,161.12**	7,756.12	18	14

Both R($*$,2)C and maxRPWC solved the instances in the first set in a backtrack-free manner. R($*$,2)C was faster than maxRPWC because of the huge size of the relations in the problem instances. R($*$,2)C took advantage of the selectivity of the tuples to tighten the constrains and simplify the problem. The same phenomenon appears to a larger extent in the second set. This set consists of looser instances, hence backtracking is inevitable. Moreover, it has large relations (about 70,000 tuples) with high constraint arity (8). The high arity induces a high selectivity among the tuples, which is exploited by R($*$,2)C. R($*$,2)C deletes tuples, hence

simplifying the problem. As a consequence, although R($*$,2)C visited about 1,500 nodes, it was visiting them with smaller relations.

5.4 Dimacs Benchmarks

Table 5.4 shows the results of the experiment on Boolean CSPs from the Dimacs benchmarks: `aim-50` with 50 variables, `aim-100` with 100 variables, and `aim-200` with 200 variables. Each problem class is divided into subclasses according to the average number of constraints. We report, for each subclass, the number of instances, number of variables, number of constraints, percentage of the instances solvable, the average time in seconds, average number of nodes visited and the number of instances completed. Note that the averages for time and nodes visited exclude instances that were not completed by any of the compared algorithms. If an algorithm did not complete any of the instances, then it is not a compared algorithm in that subclass. If no instance in a subclass was completed by all of the compared algorithms, then we do not report the average time and nodes visited. The last row shows the total number of instances completed. Each problem was ran with a time limit of one hour, and not all instances were completed within the time limit.

Table 5: Results on Dimacs benchmarks `aim-50`, `aim-100`, and `aim-200`.

				GAC			**maxRPWC**			**R($*$,2)C**			**R($*$,3)C**		
#inst.	$\|\mathcal{V}\|$	e	%solv	Time	#NV	comp	Time	#NV	comp	Time	#NV	comp	Time	#NV	comp
8	50	75	0.50	1.31	112K	8	0.71	35K	8	**0.48**	5K	8	9.05	**816.75**	8
8	50	95	0.50	1.42	45K	8	0.73	28K	8	**0.48**	4K	8	9.08	**159.25**	8
4	50	159	1.00	0.69	225	4	0.74	205.75	4	**0.48**	90.25	4	9.12	**53.50**	4
4	50	279	1.00	0.76	80	4	0.78	61.25	4	**0.60**	50.75	4	9.16	**50.00**	4
8	100	155	0.50	1K	77M	5	504.81	26M	5	10.23	86K	5	**2.05**	**128.20**	**6**
8	100	194	0.50	979.95	35M	4	642.30	15M	4	104.12	175K	4	**1.01**	**100.00**	**5**
4	100	316	1.00	0.86	5K	4	**0.76**	3K.75	4	2.71	378	4	62.46	**106.50**	4
4	100	562	1.00	3.81	214	4	**3.80**	143.25	4	9.78	108	4	628.65	**100.00**	4
8	200	312	0.5	-	-	0	-	-	0	**0.10**	**200**	2	1.59	**200.00**	**5**
8	200	387	0.5	-	-	0	-	-	0	**1.07**	535	2	4.05	**200.00**	**3**
8	200	642	1.00	-	-	2	-	-	3	-	-	**4**	-	-	1
8	200	1,157	1.00	290.00	96K	4	222.10	54K	4	558.87	4K	4	2K	**200.00**	4
80						47			48			53			**56**

This problem set has neither high arity nor huge relations. However it has a huge search space. GAC and maxRPWC visited in some instances millions of nodes, while R($*$,3)C completed the search in an almost backtrack-free manner.

The powerful filtering of R($*$,3)C explains the high performance of R($*$,3)C on these problems, especially in terms of the number of instances completed.

5.5 Randomly Generated Problems

Finally, in our last experiment, we studied the effect of varying the arity of the constraints while fixing the number of variables to 20, domain size to 10, number of constraints to 5 and the number of support tuples in the constraints to 10,000. As the arity increases, the problem becomes tighter, and exhibits the 'phase transition' phenomenon. The results averages over on 50 instances are shown in Figure 2.

This final experiment clearly illustrates the relative advantages of the three different consistency algorithms. When the arity is low, R($*$,2)C and R($*$,3)C have poor performance. R($*$,3)C suffers more than R($*$,2)C because of its higher complexity and because it does not draw any remarkable advantage from its filtering power. GAC and maxRPWC take advantage of their lower polynomial complexity and explore the search space quicker to find a solution. Notice that the number of nodes visited is almost the same for all algorithms up to arity 7.

As the constraint arity increases and the problems become tighter and more difficult, the advantages of R($*$,3)C and R($*$,2)C start showing up. R($*$,m)C takes advantage of the high constraint arity. Search visits fewer nodes at arity eight (8), and proceeds backtrack free for arity nine and above. The performance of maxRPWC improves when for constraint arity nine and above, and is rewarded by a sharp decline of the CPU time curve of maxRPWC after arity eight. GAC is clearly a 'loser' as the arity grows to eight: it is not able to filter as much as the other algorithms and consequently is not able to reduce the CPU time.

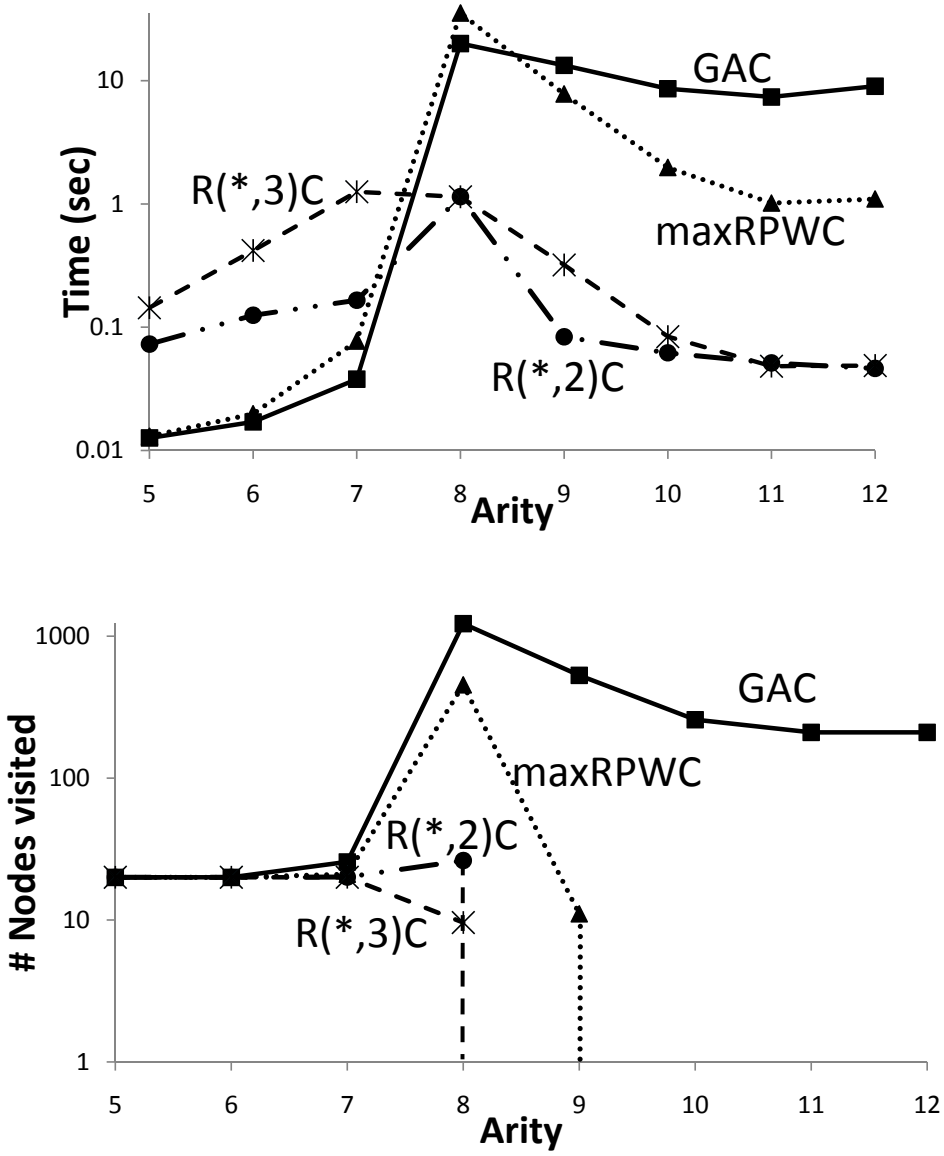

Figure 2: Random non-binary CSPs with 20 variables, domain size 10, 5 constraints and 10,000 (support) tuples per constraint. Constraint arity varies from 5 to 12.

6 Future Work & Conclusions

An anonymous reviewer of a previous version of this report suggested the two following avenues, which remain to be investigated:

- Compare the performance of our algorithm for R($*$,2)C with that of the filtering algorithms on the dual and the double encodings of non-binary CSPs reported in [Stergiou and Walsh 1999; Stergiou and Samaras 1998].
- Demonstrate that, despite the recent advances with the implementation of GAC (e.g., the specialized algorithm for table constraint by [Cheng and Yap 2004]), many problems still benefit from the use of consistency algorithms with stronger pruning power than GAC.

[Janssen *et al.* 1989] introduced an algorithm for removing redundant constraints in the dual graph of a non-binary CSP. This algorithm can be of great value to us as it can reduce the number of combinations of constraints to be considered for R($*$,m)C, and for a given combination of constraints, the number of redundant checks. While our algorithms can still be improved, especially by removing redundant edges in the dual graph of a CSP as advised by [Janssen *et al.* 1989], our work establishes that the exploitation of higher levels of consistency in non-binary CSPs can be advantageous in practice and deserves further exploration.

While it seems that R($*$,2)C is likely the most useful form of relational consistency in the context of backtrack search, our tests on Boolean CSPs (see Table 5.4) establish the usefulness of R($*$,3)C. The fact that R($*$,m)C for m>3 does not seem to be useful in the context of *search* does not rule out its usefulness in contexts where the number of constraint combinations considered is restricted.

The goal of the document is to introduce the first algorithm for computing R($*$,m)C. We believe that this algorithm must be quickly reported to serve as a foundation for further investigations. Note that m-wise consistency was introduced years ago in the database community without any algorithms or experiments. To the best of our knowledge, our algorithm is the first general algorithm for this purpose.

To summarize, we presented in this report an algorithm to enforce a *parametrized* relational consistency property. This property, unlike most other well-studied consistency properties, is enforced by tightening the existing constraints and without introducing any additional ones. Importantly, we empirically evaluated our algorithm on difficult benchmark problems and demonstrated its significance for solving

1. Hard problems specially when the relations are large and have high arity, and also

2. Boolean CSPs which have small relations of arity.

We hope that these results encourage the community to investigate more efficient algorithms for enforcing higher levels of consistency in non-binary CSPs.

Acknowledgments

We are grateful to Kostas Stergiou and Christian Bessière for countless explanations about their work and for generously sharing with us their benchmarks and their generator of random instances. Their responsiveness and cooperation have been exemplary and inspiring. We also acknowledge the feedback of anonymous reviewers of a previous version of this report. Experiments were conducted on the Research Computing Facility at UNL. Shant Karakashian was partially supported by NSF CAREER Award #0133568, and Robert Woodward by an undergraduate research grant (UCARE) of University of Nebraska-Lincoln.

References

Ken Bayer, Josh Snyder, and Berthe Y. Choueiry. An Interactive Constraint-Based Approach to Minesweeper. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 2006)*, pages 1933–1934, Boston, Massachussets, 2006.

Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

Christian Bessière, Kostas Stergiou, and Toby Walsh. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence*, 172:800–822, 2008.

Kenil C.K̃. Cheng and Roland H.C̃. Yap. Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. In *14th International Conference on Principles and Practice of Constraint Programming (CP 08)*, volume LNCS 5202, pages 509–523. Springer, 2004.

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

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

Eugene C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 (4):755–761, 1985.

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

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

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

Olivier Lhomme and Jean-Charles Régin. A Fast Arc Consistency Algorithm For N-Ary Constraints. In *AAAI 2005*, pages 405–410, 2005.

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

Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, Munich, W. Germany, 1988.

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

Kostas Stergiou and Nikos Samaras. Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. *Journal of Artificial Intelligence Research*, 24:641–684, 1998.

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

Kostas Stergiou. Personal communication, 2009.

David Waltz. Understanding Line Drawings of Scenes with Shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, Inc., 1975.

Addendum To Technical Report on "Exploring Parameterized Relational Consistency" (TR-UNL-CSE-2009-0009)

Shant K. Karakashian*, Robert J. Woodward*,
Berthe Y. Choueiry*, and Christian Bessiere**

*Constraint Systems Laboratory
Department of Computer Science & Engineering
University of Nebraska-Lincoln
Email: {shantk|rwoodwar|choueiry}@cse.unl.edu

**LIRMM-CNRS
University Montpellier, France
Email: bessiere@lirmm.fr

November 6, 2009

In this document, we revise the pseudo-code of all three algorithms in the technical report, improving on their performance.

Initializing the constraints queue

The initialization phase Algorithm 1 builds a queue of all combination-relation pairs.

Algorithm 1: INITIALIZE-$\mathcal{Q}$ initializes the queue.

Input: ζ: set of all possible combinations
Output: $\mathcal{Q}$: a queue of all combination-constraint pairs

```
1 foreach φ ∈ ζ do
2 |   foreach R ∈ φ do
3 |   |   Q ← Q ∪ {⟨φ, R⟩}
4 |   end
5 end
6 revision-time ← 0
```

Processing the constraint queue

The procedure PROCESSQUEUE, described in Algorithm 2, revises every relation-combination pair in the queue to ensure that all their tuples are supported in each combination of m constraints where the relation appears.

We modified the queue of relations (as described in the technical report), into a queue of combination-relation pairs for the following reason. Originally, when a relation R_i is popped from the queue for revision,

- It was revised in *every combination* where it appears, and
- When the revision modified R_i, every other relation in every other combination where the relation R_i appears was inserted in the queue.

According to the new queue management strategy, when a pair of combination-relation $\langle \varphi, R_i \rangle$ is popped from the queue for revision,

- It is revised in *only* the paired combinationϕ, and
- When the revision modified R_i, every other relation in every other combination where the relation R_i appears is inserted in the queue paired with the corresponding combination.

This mechanics saves in computational effort, while maintaining soundness and completeness.

Algorithm 2: ProcessQueue deletes tuples that have lost their support.

Input: $\mathcal{Q}$, ζ,*revisiouTime*
Output: *true* is the problem is R($*$,m)C, $false$ otherwise

```
1  consistent ← true
2  while (Q ≠ ∅) ∧ (consistent = true) do
3  |  ⟨φ, R⟩ ← Top(Q)
4  |  revision-time ← revision-time +1
5  |  foreach ⟨φ, R'⟩ ∈ Q do
6  |  |  Remove(⟨φ, R'⟩, Q)
7  |  |  deleted ← false
8  |  |  foreach τ ∈ R' do
9  |  |  |  if RevisionTime(τ) = revision-time then
10 |  |  |  |  GoTo 8
11 |  |  |  end
12 |  |  |  support ←FindSupport((τ, R'), φ)
13 |  |  |  if support = false then
14 |  |  |  |  Delete(τ)
15 |  |  |  |  if R' = ∅ then
16 |  |  |  |  |  consistent ← false
17 |  |  |  |  |  GoTo 29
18 |  |  |  |  |  deleted ← true
19 |  |  |  |  end
20 |  |  |  end
21 |  |  end
22 |  |  if deleted then foreach φ' ∈ ζ do
23 |  |  |  if R' ∈ φ' then foreach R'' ∈ (φ' \ {R'}) do
24 |  |  |  |  Q ← Q ∪ {⟨φ', R''⟩}
25 |  |  |  end
26 |  |  end
27 |  end
28 end
29 return consistent
```

To access all the combination-relation pairs in the queue pertaining to the same combination, we implement a hash-table on the queue whose indices are combinations and the values are the relations in the combinations.

Further, when we find the tuples $\{\tau'\}$ that support the tuple τ in a given combination ϕ, all those tuples are guaranteed 'support' and need not be rechecked for support in the combination ϕ. We use a 'time stamp' mechanism to record this situation and save redundant checks, see Line 10.

revision-time is a global variable throughout the execution so that the time stamp uniquely marks a revision of a combination. The time stamp remains the same during the revision of all the relations in a given combination. For that purpose, we need to revise, for a given same combination, all combination-relation pairs in the queue sequentially.

Finding a support

The marking of the tuples with the time stamp is performed in the FindSupport algorithm. Every time a support is found (either by search or simply retrieved from the data structure *Last*), all the tuples in the support are marked with the time stamp in Line 10 of Algorithm 3.

Algorithm 3: FindSupport finds a support for a tuple in a combination.

```
   Input: (τ, R_i), φ, revision-time
 1 support ← true
 2 if Last((τ, R_i), φ) = ∅ then
 3 |   Last((τ, R_i), φ) ← Search(φ, R_i ← τ)
 4 |   if Last((τ, R_i), φ) = ∅ then
 5 |   |   support ← false
 6 |   |   Goto 12
 7 |   end
 8 end
 9 foreach τ' ∈ Last((τ, R_i), φ) do
10 |   RevisionTime(τ') ← revision-time
11 end
12 return support
```