Light Algorithms for Maintaining Max-RPC During Search

Julien Vion and **Romuald Debruyne**
École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
{julien.vion, romuald.debruyne }@emn.fr

Abstract

This article presents two new algorithms whose purpose is to maintain the Max-RPC domain filtering consistency during search with a minimal memory footprint and implementation effort. Both are sub-optimal algorithms that make use of *support residues*, a backtrack-stable and highly efficient data structure which was successfully used to develop the state-of-the-art AC-3^{rm} algorithm. The two proposed algorithms, Max-RPC^{rm} and L-Max-RPC^{rm} are competitive with best, optimal Max-RPC algorithms, while being considerably simpler to implement. L-Max-RPC^{rm} computes an *approximation* of the Max-RPC consistency, which is guaranteed to be strictly stronger than AC with the same space complexity and better worst-case time complexity than Max-RPC^{rm}. In practice, the difference in filtering power between L-Max-RPC^{rm} and standard Max-RPC is nearly indistinguishable on random problems. Max-RPC^{rm} and L-Max-RPC^{rm} are implemented into the Choco Constraint Solver through a *strong consistency global constraint*. This work opens new perspectives upon the development of strong consistency algorithms into constraint solvers.

Introduction

This paper presents a new algorithm for enforcing the Max-RPC consistency (Debruyne and Bessière 2001), called Max-RPC^{rm}. It is a coarse-grained algorithm that makes use of *support residues* (Likitvivatanavong et al. 2004) in a way similar to the state-of-the-art AC-3^{rm} algorithm (Lecoutre and Hemery 2007). We also propose L-Max-RPC^{rm}, a simpler, lightweight version of the algorithm that computes an approximation of Max-RPC with good practical behavior and low space complexity.

The most successful techniques for solving problems with CP are based on local consistencies. Local consistencies remove values or instantiations that cannot belong to a solution. The most used, studied and versatile local consistency is Arc Consistency (AC), which removes values that do not appear in the instantiations a given constraint allows. AC is the highest level of consistency that can be obtained by considering the constraints separately. Higher levels of consistency take into account several constraints at once. They require more computing power to be enforced, but by cutting branches of the search tree earlier, the object is to reduce the (exponential) number of explored nodes in order to solve the problems faster. Max-Restricted Path Consistency (Max-RPC) is a promising consistency that lies between Arc and Path consistencies.

This work was supported by the project CANAR (ANR-06-BLAN-0383-02)

Background

A *binary constraint network* (CN) $\mathcal{N}$ consists of a pair $(\mathscr{X}, \mathscr{C})$, where $\mathscr{X}$ is a set of n variables and $\mathscr{C}$ a set of e binary constraints. The domain $\text{dom}(X)$ of variable $X \in \mathscr{X}$ is the finite set of at most d values that variable X can take. The constraints $\mathscr{C}$ specify the allowed combinations of values for given pairs of variables. A binary instantiation I is a set of two variable/value pairs, $\{(X, a), (Y, b)\}$, denoted $\{X_a, Y_b\}$. An instanciation $\{X_a, Y_b\}$ is *valid* iff $a \in \text{dom}(X)$ and $b \in \text{dom}(Y)$. A *binary relation* R is any set of instantiations. A *binary constraint* C is a pair $(\text{vars}(C), \text{rel}(C))$, where $\text{vars}(C)$ is a set of two variables and $\text{rel}(C)$ is a binary relation. $I[X]$ denotes the value of X in the instantiation I. We also denote C_{XY} the constraint such that $\text{vars}(C) = \{X, Y\}$. Given a constraint C, an instantiation I of $\text{vars}(C)$ (or of a superset of $\text{vars}(C)$, considering only the projection of I on the variables in $\text{vars}(C)$), *satisfies* C iff $I \in \text{rel}(C)$. We say that I is *allowed* by C. A *solution* of a CN $\mathcal{N}(\mathscr{X}, \mathscr{C})$ is an instantiation I_S of all variables in $\mathscr{X}$ s.t. (1.) $\forall X \in \mathscr{X}, I_S[X] \in \text{dom}(X)$ (I_S is *valid*), and (2.) I_S satisfies (is *allowed* by) all the constraints in $\mathscr{C}$.

Local consistencies

Definition 1 (Support,Arc-consistency)**.** Let $\mathcal{N} = (\mathscr{X}, \mathscr{C})$ be a CN, $C \in \mathscr{C}$ and $X \in \text{vars}(C)$. A *support* for a value $a \in \text{dom}(X)$ w.r.t. C is an instantiation $I \in \text{rel}(C)$ s.t. $I[X] = a$. A value $a \in \text{dom}(X)$ is *arc-consistent* (AC) w.r.t. C iff it has a support w.r.t. C. $\mathcal{N}$ is AC iff $\forall C \in \mathscr{C}$, $\forall X \in \text{vars}(C)$, $\forall a \, \text{dom}(X)$, a is AC w.r.t. C.

Definition 2 (Closure)**.** Let $\mathcal{N}(\mathscr{X}, \mathscr{C})$ be a CN, Φ a local consistency (e.g., *AC*). $\Phi(\mathcal{N})$ is the *closure* of $\mathcal{N}$ for Φ, i.e., the CN obtained from $\mathcal{N}$ where all allowed instantiations (resp. values in the case of domain filtering consistencies) that are not Φ-consistent have been removed.

For AC and for most consistencies applied on discrete domains, the closure is unique.

Restricted Path Consistencies

Path Consistency (Montanari 1974) is one of the most studied local consistencies. Contrary to AC, applying Path Consistency may require to add constraints, modifying the structure of the CN, or to alter the relation of existing constraints by removing allowed instanciations. PC is thus only applicable on CNs defined in extension (relations are defined by exhautively listing allowed or forbidden instanciations).

Definition 3 (Path Consistency)**.** A binary instantiation $\{X_a, Y_b\}$ is Path Consistent iff $\forall Z \in \mathscr{X}\backslash\{X, Y\}$, $\exists c \in \mathrm{dom}(Z)$ s.t. $\{X_a, Z_c\}$ and $\{Y_b, Z_c\}$ are allowed.

Restricted Path Consistencies (RPC, k-RPC and Max-RPC) are designed to catch some of the properties of Path Consistency in order to achieve strong *domain filtering consistencies* (Debruyne and Bessière 2001), which only prune values from domains and leave the structure of the CN unchanged. Max-RPC is the strongest and most promising of these consistencies (Debruyne and Bessière 1997).

Definition 4 (Max-RPC)**.** A binary CN $\mathscr{N}$ is *Max-Restricted Path Consistent* (Max-RPC) iff it is arc-consistent and for each value X_a, and each variable $Y \in \mathscr{X}\backslash X$, at least one support $\{X_a, Y_b\}$ of X_a is PC.

(Debruyne and Bessière 1997) propose Max-RPC-1, a *fine-grained* algorithm that enforces Max-RPC on a given binary CN. Max-RPC-1 is close to AC-6, a fine-grained AC algorithm that does not exploit the bidirectionnality of constraints. With g being the maximal degree of the variables in the CN (the degree of a variable is the number of constraints involving it) and c the number of 3-cliques in the constraint graph, Max-RPC-1 has a worst-case time complexity in $O(eg + ed^2 + cd^3)$ and is proved to be optimal. It has a space complexity in $O(ed + cd)$.

An enhanced version of Max-RPC-1, called Max-RPC-En1 is proposed in (Debruyne 1999). By exploiting the bidirectionnality of constraint in a manner similar to AC-7, Max-RPC-En1 manages to enforce a consistency stronger than Max-RPC with the same worst-case complexities (but slightly higher average-case time complexity).

None of these articles show how these algorithms perform when maintaining Max-RPC during search. The Quick search algorithm, that maintains Max-RPC-EnR (a slightly weaker variant of Max-RPC-En1) during search is described in (Debruyne 1998) (PhD Thesis in French).

A new coarse grained algorithm for Max-RPC

This section presents Max-RPC^{rm}, a new *coarse-grained* algorithm for Max-RPC. This algorithm uses *support residues* (Likitvivatanavong et al. 2004), which were successfully used to develop the state-of-the-art AC-3^{rm} algorithm (Lecoutre and Hemery 2007). *rm* stands for *multi-directional residues*; a residue is a support which has been stored during the execution of the procedure that proves that a given value is AC. During forthcoming calls, this procedure simply checks whether that support is still valid before searching for another support from scratch. The data structures are stable on backtrack (they do not need to be reinitialized nor restored), hence a minimal overhead on the management of data. Despite being theoretically suboptimal in the worst case, Lecoutre & Hemery showed in (Lecoutre and Hemery 2007) that AC-3^{rm} behaves better than the optimal algorithm in most cases.

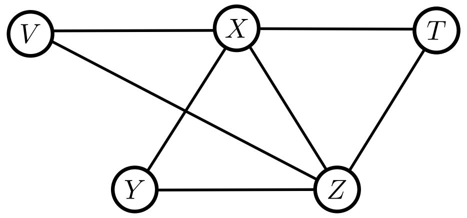

Figure 1: Example of CN (macro-structure).

Max-RPC^{rm}

Coarse-grained means that the propagation in the algorithm is managed on a variable or a constraint level, whereas fine-grained algorithms such as AC-7 or Max-RPC-1 manage the propagation on a value level. Propagation queues for coarse-grained algorithms are lighter, can be implemented very efficiently and do not require to manage extra data structures for recording which values a given instantiation supports. Moreover, variable-oriented propagation schemes permit to implement revision ordering heuristics very efficiently (Boussemart, Hemery, and Lecoutre 2004). In the following, when a variable is picked from the constraint queue, the variable with the smallest domain is selected first.

As proposed in (Bessière and Régin 1997) for the GAC-Schema algorithm, we refer to the `firstSup` and `nextSup` methods, that permit to iterate over supports of a given value in an user-defined way. In this way, fast adhoc algorithms (e.g., for arithmetical or positive table constraints) can be specified. `firstSup` has two parameters: C and X_a, and returns the first support found for X_a in $\mathrm{rel}(C)$. `nextSup` has an additional parameter: we give to the method the last support found, so as it can find the first support strictly after the last one given a static ordering of $\mathrm{rel}(C)$.

Algorithms 1 to 4 describe Max-RPC^{rm}, O-Max-RPC^{rm} and L-Max-RPC^{rm}. Lines 8-14 of Algorithm 1 and Lines 6-11 and 14-15 of Algorithm 3 are added to a standard AC-3^{rm} algorithm. The greyed parts correspond to elements to be removed in order to apply L-Max-RPC^{rm}, which is described in a further section.

Algorithm 1 contains the main loop of the algorithm. It is based on a queue containing variables that have been modified (i.e., have lost some values), which may cause some values in the neighbor variables to lose their supports. In the example depicted on Figure 1 (considering only constraints in $\mathscr{C}^\Phi$), if the variable X is modified, then the algorithm must check whether all values in T still have a support w.r.t. the constraint C_{XT}, all values in V have a support w.r.t. C_{XV}, and so on for Y and Z. This is performed by Lines 4-7 of

Algorithm 1: `MaxRPC` ($P = (\mathscr{X}, \mathscr{C}), \mathscr{Y}$)

```
$\mathscr{Y}$: the set of variables modified since the last call to MaxRPC
1  $\mathscr{Q} \leftarrow \mathscr{Y}$ ;
2  while $\mathscr{Q} \neq \emptyset$ do
3      pick $X$ from $\mathscr{Q}$ ;
4      foreach $Y \in \mathscr{X} \mid \exists C_{XY} \in \mathscr{C}$ do
5          foreach $v \in \text{dom}(Y)$ do
6              if revise($C_{XY}, Y_v$, true) then
7                  $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Y\}$;
8      foreach $(Y, Z) \in \mathscr{X}^2 \mid \exists(C_{XY}, C_{YZ}, C_{XZ}) \in \mathscr{C}^3$ do
9          foreach $v \in \text{dom}(Y)$ do
10             if revisePC($C_{YZ}, Y_v, X$) then
11                 $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Y\}$;
12         foreach $v \in \text{dom}(Z)$ do
13             if revisePC($C_{YZ}, Z_v, X$) then
14                 $\mathscr{Q} \leftarrow \mathscr{Q} \cup \{Z\}$;
```

Algorithm 2: `revisePC` (C_{YZ}, Y_a, X): boolean

```
$Y$: the variable to revise because PC supports in $X$ may have
     been lost
1  if $pcRes[C_{YZ}, Y_a][X] \in \text{dom}(X)$ then
2      return false ;
3  $b \leftarrow$ findPCSupport($Y_a, Z_{res[C_{YZ}, Y_a]}, X$) ;
4  if $b = \perp$ then
5      return revise($C_{YZ}, Y_a$, false) ;
6  $pcRes[C_{YZ}, Y_a][X] \leftarrow b$; return false;
```

Algorithm 1. The `revise` function depicted in Algorithm 3 controls the existence of such supports. It removes the value and returns **true** iff it does not have any (so **false** if the value has not been removed).

The domain of the (modified) variable that has been picked is also likely to have contained values that used to make supports in constraints situated on the opposite side of a 3-clique Path Consistent. In Figure 1, if X is modified, then the supports of V and Z w.r.t. C_{VZ}, the supports of Y and Z w.r.t. C_{YZ} and the supports of T and Z w.r.t. C_{TZ} need to be checked. This is the purpose of Lines 8-14 of Algorithm 1 and of the function `revisePC` (Algorithm 2).

Algorithm 3 iterates over the supports (X_b) of the value to revise (Y_a), on Lines 3 and 17, in search of a PC instantiation $\{X_b, Y_a\}$. The Path Consistency of the instantiation is checked on Lines 6-11 by calling `findPCSupport` (Algorithm 4) on each variable Z that forms a 3-clique with X and Y. `findPCSupport` returns either a support of the instantiation $\{X_b, Y_a\}$ in Z, or the special value $\perp$ if none can be found. Iff no PC support for Y_a can be found, the value is removed and the function returns **true**.

Residues. The `revise` function firstly checks the validity of the residue (Lines 1-2). Residues are stored in the global data structure $res[C, X_a]$, which has an $O(ed)$ space complexity. The algorithm also makes use of residues for the PC supports, stored in the structure $pcRes$ with an $O(cd)$

Algorithm 3: `revise` ($C_{XY}, Y_a, supportIsPC$): boolean

```
$Y_a$: the value of $Y$ to revise against $C_{XY}$ – supports in $X$
       may have been lost
$supportIsPC$: false if one of $pcRes[C_{XY}, Y_a]$ is no longer
               valid
1  if $supportIsPC \wedge res[C_{XY}, Y_a] \in \text{dom}(X)$ then
2      return false ;
3  $b \leftarrow$ firstSup($C_{XY}, Y_a$)[$X$] ;
4  while $b \neq \perp$ do
5      $PConsistent \leftarrow$ true ;
6      foreach $Z \in \mathscr{X} \mid (X, Y, Z)$ form a 3-clique do
7          $c \leftarrow$ findPCSupport($Y_a, X_b, Z$) ;
8          if $c = \perp$ then
9              $PConsistent \leftarrow$ false ;
10             break;
11         $currentPcRes[Z] \leftarrow c$ ;
12     if $PConsistent$ then
13         $res[C_{XY}, Y_a] \leftarrow b$ ; $res[C_{XY}, X_b] \leftarrow a$ ;
14         $pcRes[C_{XY}, Y_a] \leftarrow currentPcRes$ ;
15         $pcRes[C_{XY}, X_b] \leftarrow currentPcRes$ ;
16         return false ;
17     $b \leftarrow$ nextSup($C_{XY}, Y_a, \{X_b, Y_a\}$)[$X$] ;
18 remove $a$ from dom($Y$) ;
19 return true ;
```

Algorithm 4: `findPCSupport` (X_a, Y_b, Z): value

```
1  $c_1 \leftarrow$ firstSup($C_{XZ}, X_a$)[$Z$] ;
   $c_2 \leftarrow$ firstSup($C_{YZ}, Y_b$)[$Z$] ;
2  while $c_1 \neq \perp \wedge c_2 \neq \perp \wedge c_1 \neq c_2$ do
3      if $c_1 < c_2$ then
4          $c_1 \leftarrow$ nextSup($C_{XZ}, X_a, \{X_a, Z_{c_2 - 1}\}$)[$Z$] ;
5      else
6          $c_2 \leftarrow$ nextSup($C_{YZ}, Y_b, \{Y_b, Z_{c_1 - 1}\}$)[$Z$] ;
7  if $c_1 = c_2$ then return $c_1$ ;
8  return $\perp$ ;
```

space complexity (c is the number of 3-cliques in the CN). The idea is to associate the residue found by the `revise` function with the found PC value for each third variable of the 3-clique. In this way, at the end of the processing, $(X_a, res[C_{XY}, X_a], pcRes[C_{XY}, X_a][Z])$ forms a 3-clique in the micro-structure of the constraint graph for all 3-cliques (X, Y, Z) of the CN and for all $a \in \text{dom}(X)$.

In the example depicted on Figure 2, at the end of the processing we have $res[C_{XY}, X_a] = b$, $pcRes[C_{XY}, X_a][Z] = a$, $pcRes[C_{XY}, Y_a][Z'] = a$, and so on. The algorithm exploits the bi-directionnality of the constraints: if Y_b is a support for X_a with $\{Z_a, Z'_a\}$ as PC supports, then X_a is also a support for Y_b with the same PC supports. This is done on Lines 13-15 of Algorithm 3.

If a lost PC support is detected on Line 1 of `revisePC`, then an alternative support is searched. If none can be found, then the current support of the current value is no longer PC, and another one must be found. This is done by a call to `revise` on Line 5 of Algorithm 2.

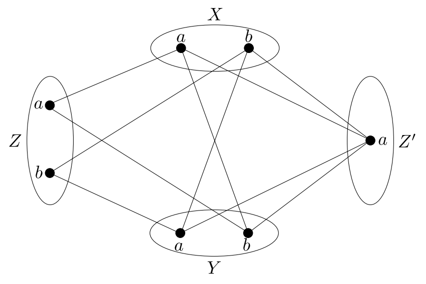

Figure 2: Example with two 3-cliques (microstructure)

$$\begin{tikzcd} & & & \text{AC} \arrow[dd, dash, "/" marking] \\ \text{Max-RPC-En} \arrow[r] & \text{Max-RPC} \arrow[r] & \text{L-Max-RPC} \arrow[ur] \arrow[dr] & \\ & & & \text{O-Max-RPC} \end{tikzcd}$$

Figure 3: Comparing the consistencies. An arrow means "is strictly stronger than" and the crossed line means "is incomparable with".

One-pass Max-RPC

One common way to define approximation of a strong consistency Φ is to remove the propagation process in the algorithm. The strong consistency property is thus checked only once for each value of the CN. This process ensures that all the values that were not Φ-consistent before the first call to the algorithm will be filtered. One example of this idea is proposed in (Freuder and Elfe 1996) for the Neighborhood Inverse Consistency property under the name one-pass NIC.

With our Max-RPC^{rm} algorithm, this propagation process lies in Line 7 and the **foreach do** loop on Lines 8-14 of Algorithm 1. In order to apply one-pass Max-RPC, they must be removed. The `revisePC` function and *pcRes* data structure are no longer useful and can be removed, together with Lines 11 and 14-15 of Algorithm 3 (all greyed lines in the algorithms). We call the obtained algorithm **O-Max-RPC**^{rm}. The same approximation can be used on Max-RPC-1 to define the O-Max-RPC-1 algorithm. The closure obtained by applying this consistency is not unique and will depend on the order in which the modified variables are picked from $\mathcal{Q}$. As the loss of AC supports is not propagated, O-Max-RPC is incomparable with AC: there exists CNs that are not AC after applying O-Max-RPC on them, and AC CNs on which O-Max-RPC can filter some values. O-Max-RPC is not incremental.

Light Max-RPC

We propose another approximation of Max-RPC, that lies between one-pass and full Max-RPC. The idea is to keep the propagation process of the original Max-RPC algorithm, but only to propagate the loss of AC supports. This ensures that the obtained algorithms enforce a consistency that is at least as strong as AC.

For Max-RPC, this means that we remove the propagation of lost PC supports (the **foreach do** loop on Lines 8-14 of Algorithm 1, the `revisePC` function, the *pcRes* data structure and Lines 11 and 14-15 of Algorithm 3). The algorithm can be obtained by removing all the greyed parts in Algorithms 1-3. Line 7 of Algorithm 1 is kept, so as to propagate the loss of AC supports. We call this algorithm **L-Max-RPC**^{rm}. The same approximation can be used to define the L-Max-RPC-1 algorithm. L-Max-RPC is strictly stronger than AC: any CN on which L-Max-RPC has been applied is either empty or AC, and there exists at least one AC CN on which L-Max-RPC can filter some values. Applying L-Max-RPC on a given CN does not lead to an unique closure, L-Max-RPC is also strictly stronger than O-Max-RPC given the same initial ordering of variables in $\mathcal{Q}$.

Experiments we conducted (see below) show empirically that the filtering power of L-Max-RPC is only slightly weaker than that of Max-RPC on random problems, despite the significant gains in space and time complexities. A summary of the different consistencies defined in this paper is given on Figure 3.

Complexity issues

We use these additional notations: c is the number of 3-cliques in the constraint graph ($c \leq \binom{n}{3} \in O(n^3)$), g is the maximum degree of a variable and s is the maximum number of 3-cliques that share the same single constraint in the constraint graph. If the constraint graph is not empty (at least two variables and one constraint), we have the following relation: $s < g < n$. Let us remind here that in a binary CN, $e \leq ng/2$. The complexities are devised in terms of constraint checks (assumed in constant time).

Proposition 1. *After an initialization phase in $O(eg)$, Max-RPC^{rm} has a worst-case time complexity in $O(ed^3 + csd^4)$ and a space complexity in $O(ed + cd)$.*

Proof sketch. The initialization phase consists in detecting and linking all 3-cliques to their associated constraints and variables, which can be done in $O(eg)$.

The main loop of the algorithm depends on the variable queue $\mathcal{Q}$. Since variables are added to $\mathcal{Q}$ when they are modified, they can be added at most d times in the queue, which implies that the main loop can be performed $O(nd)$ times. This property remains true when the algorithm is called multiple times, removing one value from the domain of one variable every time. The algorithm is incremental, especially when maintaining Max-RPC in a systematic search algorithm. We consider separately the two parts of the main loop.

1. the **foreach do** loop at Lines 4-7 of Algorithm 1.
   This loop can be performed $O(g)$ times. Since in the worst case, the whole CN is explored thoroughly in an homogeneous way, it is amortized with the $O(n)$ factor of the main loop in a global $O(e)$ complexity. The **foreach do** loop at Lines 5-7 involves $O(d)$ calls to `revise` (total $O(ed^2)$ revises).
   `revise` (Algorithm 3) first calls `firstSup`, which has a complexity of $O(d)$ (without any assumption on the nature of the constraint). The **while do** loop can be performed $O(d)$ times. Calls to `nextSup` (Line 17) are part of the loop. The **foreach do** loop on Lines

Algorithm	Time complexity	Space cplx
AC-3^{rm}	$O(ed^3)$	$O(ed)$
O-Max-RPC^{rm}	$O(eg + ed^2 + cd^3)$	$O(c + ed)$
L-Max-RPC^{rm}	$O(eg + ed^3 + cd^4)$	$O(c + ed)$
Max-RPC^{rm}	$O(eg + ed^3 + csd^4)$	$O(cd + ed)$
Max-RPC-En	$O(eg + ed^2 + cd^3)$	$O(cd + ed)$

Table 1: Summary of complexities

6-11 can be performed $O(s)$ times, and involves a call to `findPCSupport` in $O(d)$. Thus, `revise` is in $O(d + sd^2)$. The global complexity of this first part is thus $O(ed^3 + esd^4)$. The $O(es)$ factor is amortized to $O(c)$, thus a final result in $O(ed^3 + cd^4)$.

2. the **foreach do** loop at Lines 8-14 of Algorithm 1. The number of turns this loop can perform is amortized with the main loop to an $O(cd)$ factor. Each turn executes $O(d)$ calls to `revisePC`, whose worst-case time complexity is capped by a call to `revise` on Line 5 of Algorithm 2. This part of the algorithm is thus in $O(cd^2.(d + sd^2)) = O(csd^4)$.

The algorithm uses three data structures: storing the 3-cliques in $\Theta(c)$, storing the AC residues in $O(ed)$ (*res* data structure) and storing the PC residues in $O(cd)$ (*pcRes* data structure), hence a space complexity in $O(ed + cd)$. □

If `revise` is called due to the removal of a value that does not appear in any support, its complexity falls down to $O(sd)$. In practice, this happens very regularly, which explains the good practical behavior of the algorithm.

Proposition 2. *After an initialization phase in $O(eg)$, L-Max-RPC^{rm} has a worst-case time complexity in $O(ed^3 + cd^4)$ and a space complexity in $O(c + ed)$.*

Proof sketch. As L-Max-RPC^{rm} skips the 2nd part of the algorithm, the $O(csd^4)$ term is removed. As the *pcRes* data structure is removed, the remaining data structures are in $O(c + ed)$. □

Proposition 3. *After an initialization phase in $O(eg)$, O-Max-RPC^{rm} has a worst-case time complexity in $O(ed^2 + cd^3)$ and a space complexity in $O(c + ed)$.*

Proof sketch. As O-Max-RPC^{rm} prevents the modifications in `revise` to be propagated, every variable is revised only once. The main loop of the algorithm is thus performed n times instead of $O(nd)$. Thus the worst-case complexity is reduced by an $O(d)$ factor w.r.t. L-Max-RPC^{rm}. O-Max-RPC^{rm} has the same data structures as L-Max-RPC^{rm}. □

Note that with all variants of the algorithm, the initialization phase in $O(eg)$ is performed previously to the first call to Algorithm 1 and only once when maintaining the Max-RPC property throughout the search. Table 1 gives a summary of the complexities of the different algorithms studied in this paper, as well as those of AC-3^{rm} and Max-RPC-En for reference.

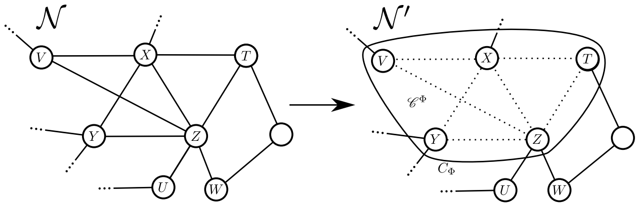

Figure 4: A strong consistency global constraint C_Φ, used to enforce the strong local consistency on a subset of constraints $\mathscr{C}^\Phi$. $\mathscr{N}'$ is the new CN obtained when replacing $\mathscr{C}^\Phi$ by the global constraint.

Implementation

In order to integrate strong consistency algorithms such as Max-RPC^{rm} into an event-based solver such as *Choco* (Laburthe, Jussien, and others 2008), we designed a new *global constraint* and an object-oriented generic scheme, as detailed in (Vion, Petit, and Jussien 2009).

Choco, like many mature solvers, use an AC-5 based propagation scheme (van Hentenryck, Deville, and Teng 1992). *Propagators* are associated with constraints to enforce a given level of local consistency. We call them event-based solvers. One of the reasons for which CP is currently applied with success to real-world problems is that some propagators are encoded through *filtering algorithms*, which exploit the semantics of the constraints. Global constraints (Beldiceanu and Contejean 1994; Régin 1994; Bessière and van Hentenryck 2003) are constraints whose semantics may correspond to well-known Operations Research problems. Powerful resolution algorithms that exist for these problems are thus used to implement propagators. Filtering algorithms are often derived from well-known Operations Research techniques. This provides powerful implementations of propagators. Each propagator is called according to the events that occur in domains of the variables involved in its constraint. Most often, an event is a value deleted by another constraint. At each node of the search tree, the pruning is performed within the constraints. The fix-point is obtained by propagating events among all the constraints. As constraints are considered independently, this scheme does not appear to enable the implementation of strong consistency algorithms.

Given a local consistency Φ, the principle of our global constraint is to deal with the subset $\mathscr{C}^\Phi$ of constraints on which Φ should be applied, within a new global constraint C_Φ added to the CN. Constraints in $\mathscr{C}^\Phi$ are connected to C_Φ instead of being included into the initial CN $\mathscr{N}$ (see Figure 4). In this way, events related to constraints in $\mathscr{C}^\Phi$ are handled in a closed world, independently from the propagation queue of the solver. This permits to implement any (strong) consistency algorithm in an event-based constraint solver with minimal implementation effort. As a side effect, this scheme permits to easily apply different levels of consistency in the same CN, and to coexist with semantics-based global constraints used to solve the given problem.

Experiments

We implemented the algorithms using our own binary constraint solver, and in the general-purpose Choco Solver (Laburthe, Jussien, and others 2008), using the method described above. On the Figures 5-6, each point is the average result over 100 generated binary random problem of various characteristics solved using our binary constraint solver. A binary random problem is characterized by a quadruple (n, d, γ, t) whose elements respectively represent the number of variables, the number of values, the density[1] of the constraint graph and the tightness[2] of the constraints.

Pre-processing: Figure 5 compares the time and memory used for the initial propagation on rather large problems (200 variables, 30 values). Left hand figures are results with 5% density, right hand figures are results with 15%. Topmost figures compare Max-RPC-1, Max-RPC^{rm}, their Light variants, and Max-RPC-En1. The two small figures show the percentage of removed values w.r.t. the tightness of the constraints. There is a transition phase from a zone where no values can be removed, to a zone where the inconsistency of the problems are detected during the pre-processing phase. The earliest the transition phase occurs, the strongest the algorithm is. Of course, Max-RPC-1 and Max-RPC^{rm} detect the same inconsistent values. With low densities, Max-RPC-En1 also has the same filtering power. The weaker Light variants nearly coincide between each other. All algorithms show a peak in cpu time near the threshold. Although Max-RPC^{rm} tends to be the slower algorithm, L-Max-RPC^{rm} is the fastest algorithm before the peak and only very slightly slower than L-Max-RPC-1 after the peak. Bottommost pictures show the comparison between one-pass variants, with Max-RPC-1 and L-Max-RPC^{rm} given for reference. The two one-pass algorithms have about the same performances, and are only slightly faster than L-Max-RPC^{rm} despite their lower filtering power.

These graphs show that L-Max-RPC^{rm} is very competitive w.r.t. Max-RPC-1 and Max-RPC-En1 in both speed and filtering power, despite its simplicity and low space complexity.

Maintaining Max-RPC during search: Figure 6 depicts experiments with a systematic search algorithm, where the various levels of consistency are maintained throughout search. The variable ordering heuristic is $dom/ddeg$.[3] The impact of the number of variables, number of values and the density of the problem on the search time and the number of nodes is evaluated. All runs were done at the threshold point. For each point, we found the tightness t_{tp} where the transition phase occurs, and used that value to generate the instances. The cpu time (in seconds) and number of nodes is shown for each algorithm on the left-hand figure. The relative difference in cpu time and number of nodes between running respectively L-Max-RPC^{rm} and Max-RPC-EnR w.r.t. AC-3^{rm} at each node of the search is depicted of the right-hand figures. All graphs show a very similar behavior between maintaining Max-RPC-EnR and maintaining L-Max-RPC^{rm}, despite the latter's simplicity and low memory usage.

The top left graph shows how the behavior of the different algorithms evolves as the number of variables grows. The density is evaluated using the $\mathrm{neighb}(\overline{g}, n)$ function so that the average degree $\overline{g}$ of the variables remains constant.[4] The graph shows that the additional filtering of Max-RPC algorithms tends to reduce the number of nodes by about 40% w.r.t. maintaining AC. More time is gained when the number of variables grows, up to 30% less time taken with Max-RPC algorithms at 110-120 variables. Top right graph shows that the number of nodes and time tends to be more reduced as the size of the domains grows, up to 60% nodes less and 30% time less around 60 variables.

The bottom left graph shows that even though the number of nodes is reduced by a stable 45-50% factor when the density grows, it does not longer compensate the additional time spent in the filtering algorithm. With the given characteristics (50 variables, 20 values), maintaining AC is more efficient when the density is above 14%.

These figures tends to show that maintaining Max-RPC is especially efficient on large problems (in terms of number of variables/values) that are not too much dense, and confirms the competitiveness of L-Max-RPC^{rm} over Max-RPC-En.

Mixing local consistencies:[5] Table 2 shows the effectiveness of the possibility of mixing two levels of consistency within the same model, using the Choco Solver. The first row corresponds to the median results over 50 instances of problems $(35, 17, 44\%, 31\%)$, and the second row to $(105, 20, 5\%, 65\%)$ instances. Given its higher density, the first problem is better resolved by using AC-3^{rm} while the second one shows better results with Max-RPC. Third row corresponds to instances where two problems are concatenated and linked with a single additional loose constraint. On the last two columns, we maintain AC on the denser part of the model, and (L-)Max-RPC^{rm} on the rest. Mixing the two consistencies entails a faster solving, which emphasizes the interest of our approach. The last two rows present the results with larger problems.

Conclusion & Perspectives

This paper presented Max-RPC^{rm}, a new, simple algorithm for enforcing the Max-RPC domain filtering consistency. Two variants of the algorithm, O-Max-RPC^{rm} and L-Max-RPC^{rm} were proposed, studied, experimented and compared to the legacy Max-RPC algorithms. Experiments showed that L-Max-RPC^{rm} is competitive with state-of-the-art, optimal algorithms, although being considerably simpler to implement and requiring less data structures. Max-RPC^{rm} and its variants were implemented into the Choco Solver, exploiting (Vion, Petit, and Jussien 2009)'s generic

[1]The density is the proportion of constraints in the graph w.r.t. the maximal number of possible constraints, i.e., $\gamma = e/\binom{n}{2}$.

[2]The tightness is the proportion of instantiations forbidden by each constraint.

[3]The process of weighting constraints for $dom/wdeg$ is not defined when more than one constraint lead to a domain wipeout.

[4]$\mathrm{neighb}(\overline{g}, n) = \overline{g}/(n-1)$

[5]These results are extracted from (Vion, Petit, and Jussien 2009)

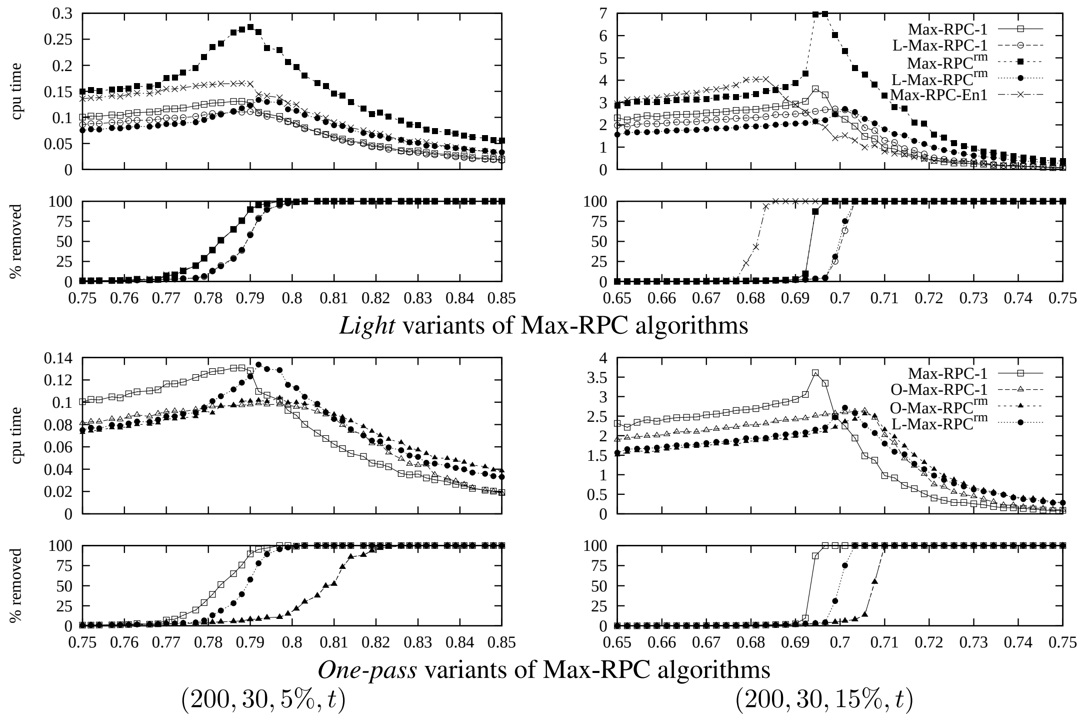

Figure 5: Initial propagation: cpu time and filtering power on homogeneous random problems (200 variables, 30 values).

		AC-3^{rm}	Max-RPC^{rm}	L-Max-RPC^{rm}	AC-3^{rm}+Max-RPC^{rm}	AC-3^{rm}+L-Max-RPC^{rm}
$(35, 17, 44\%, 31\%)$	*cpu (s)*	**6.1**	25.6	11.6	non	non
	nodes	21.4k	5.8k	8.6k	applicable	applicable
$(105, 20, 5\%, 65\%)$	*cpu (s)*	20.0	19.4	**16.9**	non	non
	nodes	38.4 k	20.4 k	19.8 k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	96.8	167.2	103.2	90.1	**85.1**
$+(105, 20, 5\%, 65\%)$	*nodes*	200.9k	98.7k	107.2k	167.8k	173.4k
$(110, 20, 5\%, 64\%)$	*cpu (s)*	73.0	60.7	**54.7**	non	non
	nodes	126.3k	54.6k	56.6k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	408.0	349.0	272.6	284.1	**259.1**
$+(110, 20, 5\%, 64\%)$	*nodes*	773.0k	252.6k	272.6k	308.7k	316.5k

Table 2: Mixing two levels of consistency in the same model

scheme for adding strong local consistencies to the set of features of constraint solvers. This technique allows a solver to use different levels of consistency for different subsets of constraints in the same model. The interest of this feature is validated by our experiments.

This work opens many perspectives upon the development of strong consistency algorithms. The use of backtrack-stable data structures such as support residues, and the development of approximations of strong consistencies seem very promising. Future works include the development of strong consistency algorithms that can be applied on non-binary CNs.

References

Beldiceanu, N., and Contejean, E. 1994. Introducing global constraints in CHIP. *Mathl. Comput. Modelling* 20(12):97–123.

Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 398–404.

Bessière, C., and van Hentenryck, P. 2003. To be or not to be... a global constraint. In *Proceedings of CP'03*, 789–794.

Boussemart, F.; Hemery, F.; and Lecoutre, C. 2004. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, 29–43.

Debruyne, R., and Bessière, C. 1997. From restricted path consistency to max-restricted path consistency. In *Proceedings of CP'97*, 312–326.

Debruyne, R., and Bessière, C. 2001. Domain filtering

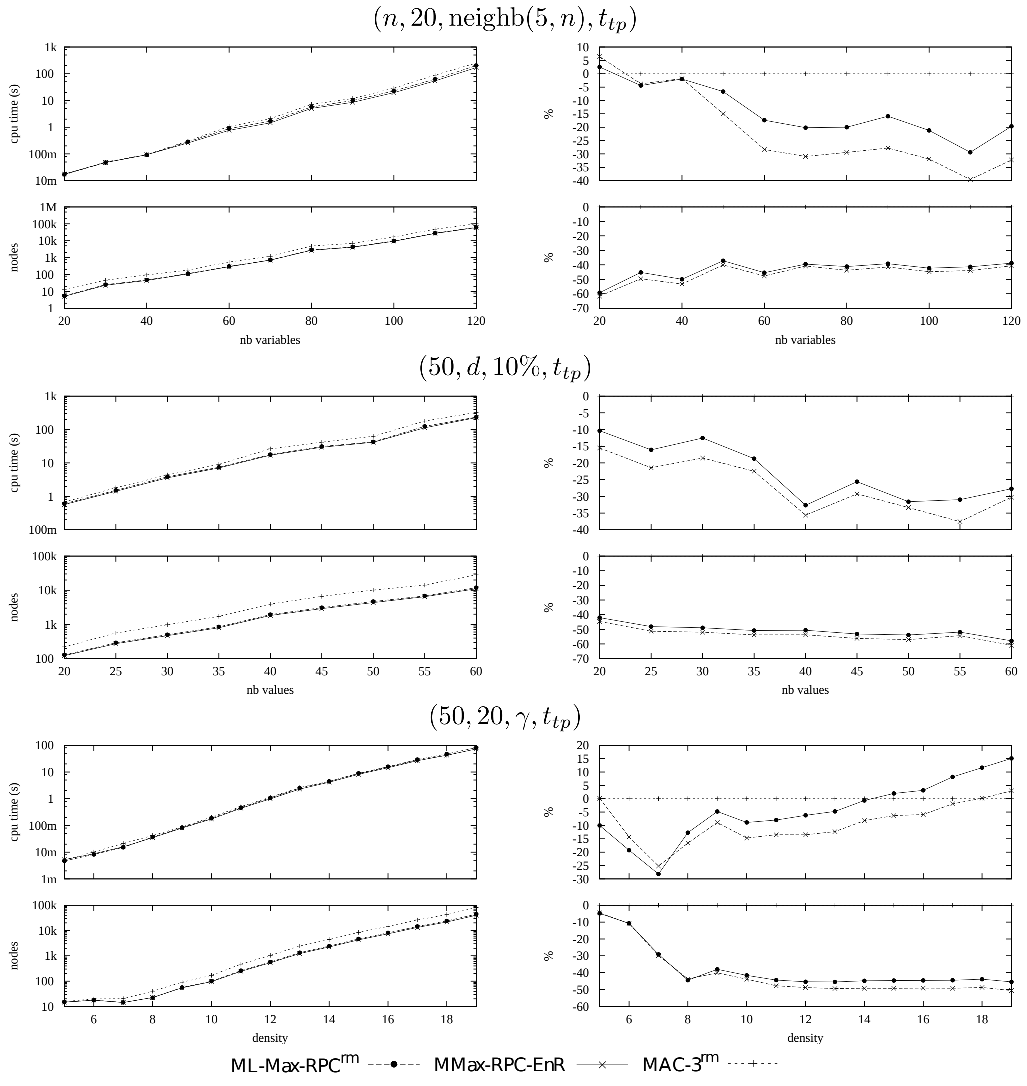

Figure 6: Full search: cpu time and nodes on homogeneous random problems at the transition point. Right-hand figures show relative differences w.r.t. MAC-3^{rm}.

consistencies. *Journal of Artificial Intelligence Research* 14:205–230.

Debruyne, R. 1998. *Consistances locales pour les problèmes de satisfaction de contraintes de grande taille*. Ph.D. Dissertation, Université Montpellier II.

Debruyne, R. 1999. A strong local consistency for constraint satisfaction. *Proceedings of ICTAI'1999* 202–209.

Freuder, E., and Elfe, C. 1996. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'1996*, volume 1, 202–208.

Laburthe, F.; Jussien, N.; et al. 2008. Choco: An open source Java constraint programming library. `http://choco.emn.fr/`.

Lecoutre, C., and Hemery, F. 2007. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, 125–130.

Likitvivatanavong, C.; Zhang, Y.; Bowen, J.; and Freuder, E. 2004. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, 93–107.

Montanari, U. 1974. Network of constraints : Fundamental properties and applications to picture processing. *Information Science* 7:95–132.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, 362–367.

van Hentenryck, P.; Deville, Y.; and Teng, C. 1992. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence* 57:291–321.

Vion, J.; Petit, T.; and Jussien, N. 2009. A generic scheme for integrating strong consistencies into constraint solvers. In *Proceedings of ERCIM workshop CSCLP'2009*. to appear.