

# From Restricted Path Consistency to Max-Restricted Path Consistency

Romuald Debruyne and Christian Bessière

LIRMM (UMR 5506 CNRS), 161 rue Ada, 34392 Montpellier Cedex 5 - France  
Email: {debruyne, bessiere}@lirmm.fr

**Abstract.** There is no need to show the importance of the filtering techniques to solve constraint satisfaction problems i.e. to find values for problem variables subject to constraints that specify which combinations of values are consistent. They can be used during a preprocessing step to remove once and for all some local inconsistencies, or during the search to efficiently prune the search tree. Recently, in [5], a comparison of the most practicable filtering techniques concludes that restricted path consistency (RPC) is a promising local consistency that requires little additional cpu time compared to arc consistency while removing most of the path inverse inconsistent values. However, the RPC algorithm used for this comparison (presented in [1] and called RPC1 in the following) has a non optimal worst case time complexity and bad average time and space complexities. Therefore, we propose RPC2, a new RPC algorithm with  $O(end^2)$  worst case time complexity and requiring less space than RPC1 in practice. The second aim of this paper is to extend RPC to new local consistencies,  $k$ -RPC and Max-RPC, and to compare their pruning efficiency with the other practicable local consistencies. Furthermore, we propose and study a Max-RPC algorithm based on AC-6 that we used for this comparison.

## 1 Introduction

Finding a solution in a constraint network (CN) involves looking for a set of value assignments, one for each variable, so that all the constraints are simultaneously satisfied. Many exponential search algorithms have been proposed to solve this NP-hard task. To avoid combinatorial explosion, the search tree has to be pruned as much as possible. Thus, filtering techniques are used to remove some local inconsistencies before or during the search.

Arc and path consistencies are the most studied local consistencies. While the latter is seldom used, most of real applications maintain arc consistency (MAC [10]) or a limited version of arc consistency (forward checking [8]) during the search. Indeed, arc consistency (AC) removes some values that cannot belong to any solution, which can strongly reduce the search space. More, AC can be enforced cheaply. On the other hand path consistency (PC) removes some inconsistent pairs of values with a huge complexity. The constraints must be represented in extension, and the structure of the network can be changed. PC is almost never used because of these important drawbacks.

On large and hard CNs, MAC outperforms forward checking. Although maintaining whole arc consistency on easy and very small CNs is useless, it widely speeds up the search on hard CNs. The harder a CN is, the more useful filtering techniques are. It is then necessary to take a careful look at local consistencies stronger than AC that do not fall in the same traps as PC. Recently, some powerful local consistencies have been proposed. A comparison in [5] concludes that restricted path consistency (RPC [1]) is a promising local consistency. It requires little additional cpu time compared to AC while removing most of the path inverse inconsistent values (PIC [7]). Furthermore, it does not delete any pair of values and thus, no constraint is added or modified in the network. In this paper we extend the idea of RPC to new local consistencies,  $k$ -RPC and Max-RPC, in order to remove more inconsistent values. The greater  $k$  is, the more powerful  $k$ -RPC is. Although Max-RPC removes more inconsistent values than PIC and  $k$ -RPC for any  $k$ , enforcing a high level of  $k$ -RPC is often more expensive than achieving Max-RPC. But in order to compare RPC and Max-RPC with the other practicable local consistencies, we need efficient algorithms to achieve them.

The experimental evaluation presented in [5] shows that the non optimal algorithm RPC1 has the same bad behaviour as AC-4 [9] on which it is based. AC-4 has a heavy  $O(ed^2)$  data structure and its average time complexity is close to the worst case. In addition to these drawbacks, RPC1 has an  $O(end^3)$  worst case time complexity. Therefore, in this paper we propose a new RPC algorithm, called RPC2, which has an  $O(end^2)$  worst case time complexity and requires less space than RPC1 in practice. Moreover, we propose and study a Max-RPC algorithm based on AC-6.

After some recalls in section 2, we present the  $k$ -RPC and the Max-RPC local consistencies and study their pruning efficiency. Some recalls on RPC1 are given in section 4. We propose a new RPC algorithm called RPC2 in section 5 and a Max-RPC algorithm in section 6. An experimental evaluation is given in section 7 and some remarks conclude this paper.

## 2 Definitions and notations

A *network of binary constraints*  $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$  is a set  $\mathcal{X} = \{i, j, \dots\}$  of  $n$  variables, each taking value in its respective finite *domain*  $D_i, D_j, \dots$  elements of  $\mathcal{D}$  and a set  $\mathcal{C}$  of  $e$  binary constraints.  $d$  is the size of the largest domain. A *binary constraint*  $C_{ij}$  is a subset of the cartesian product  $D_i \times D_j$  that denotes the compatible pairs of values for  $i$  and  $j$ . We note  $C_{ij}(a, b) = true$  to specify that  $((i, a), (j, b)) \in C_{ij}$ . We then say that  $(j, b)$  is a *support* of  $(i, a)$  on  $C_{ij}$ . With each CN we associate a *constraint graph* in which nodes represent variables and arcs connect pairs of variables which are constrained explicitly. The *neighborhood* of  $i$  is the set of variables linked to  $i$  in the constraint graph. A domain  $\mathcal{D}' = \{D'_i, D'_j, \dots\}$  is a *sub-domain* of  $\mathcal{D} = \{D_i, D_j, \dots\}$  if  $\forall i, D'_i \subseteq D_i$ . An *instantiation* of a set of variables  $S$  is an indexed set of values  $\{I_j\}_{j \in S}$  s.t.  $\forall j \in S \quad I_j \in D_j$ . An instantiation  $I$  of  $S$  satisfies a constraint  $C_{ij}$  if  $\{i, j\} \not\subseteq S$  or  $C_{ij}(I_i, I_j)$  is true. An instantiation is *consistent* if it satisfies all the constraints.

A pair of values  $((i, a), (j, b))$  is *path consistent* if for all  $k \in \mathcal{X}$  s.t.  $j \neq k \neq i \neq j$ , this pair of values can be extended to a consistent instantiation of  $\{i, j, k\}$ .  $(j, b)$  is a *path consistent support* of  $(i, a)$  if  $((i, a), (j, b))$  is path consistent. A *solution* of  $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$  is a consistent instantiation of  $\mathcal{X}$ . A value  $(i, a)$  is *consistent* if there is a solution  $I$  such that  $I_i = a$ . A CN is *consistent* if it has at least one solution. In the following we denote by  $P|_{D_i=\{a\}}$  the CN obtained by restricting  $D_i$  to  $\{a\}$  in  $P$ .

### 3 Restricted path consistency and its extensions

Restricted path consistency allows to detect more inconsistent values than AC while avoiding the drawbacks of PC. RPC is based on the following remark: If a value  $(i, a)$  has an unique support  $(j, b)$  on a constraint  $C_{ij}$ , the path inconsistency of  $((i, a), (j, b))$  leads to the inconsistency of  $(i, a)$ . So, in addition to the arc inconsistent value deletions, an RPC algorithm checks the path consistency of the pairs of values  $((i, a), (j, b))$  if  $(j, b)$  is the unique support of  $(i, a)$  in  $D_i$ . These tests can directly lead to the deletion of  $(i, a)$  if  $((i, a), (j, b))$  is found to be path inconsistent. RPC does not require to explicitly maintain the list of allowed pairs of values (since it does not remove pairs of values from constraints), and it avoids the prohibitive cost of path consistency, which checks all the pairs of values.

RPC checks the path consistency of a support  $(j, b)$  only when it is the unique support of a value  $(i, a)$ . It can be extended to a more pruningful filtering technique: Instead of checking path consistency of supports only when there is an unique one, we could do that each time  $(i, a)$  has at most  $k$  supports. This is the principle of  $k$ -restricted path consistency ( $k$ -RPC), which ensures that the values that have at most  $k$  supports on a constraint have at least one path consistent support on this constraint. RPC is 1-RPC and AC corresponds to 0-RPC. The greater  $k$  is, the smaller the probability for a value having  $k$  supports to have no path consistent support is. But we cannot say that the smaller  $k$  is, the smaller the cpu time to number of value deletions ratio of  $k$ -RPC is. Indeed, this rely on the constraint network.  $k$ -RPC can detect the inconsistency of a CN that has no  $(k-1)$ -restricted path inconsistent value. A real advantage of  $k$ -RPC is that it can be used in an adaptative way, reusing the filtering effort. If  $k$ -RPC holds in a CN and we want to enforce  $(k+1)$ -RPC, we have only to consider the values that have exactly  $(k+1)$  supports on a constraint and to propagate their possible deletion. Obviously, we can stop enforcing higher levels of  $k$ -RPC as soon as all the values have a path consistent support on each constraint. In this case  $d$ -RPC holds ( $d$  being the size of the largest domain). If after the deletion of the  $k'$ -restricted path inconsistent values no value has more than  $k'$  supports on any constraint,  $k''$ -RPC holds for all  $k'' > k'$ . A CN is said Max-restricted path consistent if all the values have a path consistent support on each constraint, whatever is the number of supports they have. Although a Max-restricted path consistent CN is  $k$ -restricted path consistent for all  $k$ , enforcing Max-RPC can be less expensive than achieving  $k$ -RPC. Indeed, as opposed to  $k$ -RPC, enforcing Max-RPC does not require to determine the set of values that have no more than  $k$  supports on

- A binary CN is **(i, j)-consistent** iff  $\forall i \in \mathcal{X}, D_i \neq \emptyset$  and any consistent instantiation including any j additional variables.
- A domain  $D_i$  is arc consistent iff,  $\forall a \in D_i, \forall j \in \mathcal{X}$  s.t.  $C_{ij} \in \mathcal{C}$ , there exists  $b \in D_j$  s.t.  $C_{ij}(a, b)$ . A CN is **arc consistent** ((1, 1)-consistent) iff  $\forall D_i \in \mathcal{D}, D_i \neq \emptyset$  and  $D_i$  is arc consistent.
- A pair of variables  $(i, j)$  is path consistent iff  $\forall (a, b) \in C_{ij}, \forall k \in \mathcal{X}$ , there exists  $c \in D_k$  s.t.  $C_{ik}(a, c)$  and  $C_{jk}(b, c)$ . A CN is **path consistent** ((2, 1)-consistent) iff  $\forall i, j \in \mathcal{X}, (i, j)$  is path consistent.
- A binary CN is **strongly path consistent** iff it is node-consistent, arc consistent and path consistent.
- A binary CN is **k-restricted path consistent** iff  
 $\forall i \in \mathcal{X}, D_i$  is a non empty arc consistent domain and,  
 $\forall (i, a) \in D_i$  for all  $j \in \mathcal{X}$  s.t.  $(i, a)$  has at most  $k$  supports in  $D_j$ ,  
 $\exists b \in D_j$  s.t.  $C_{ij}(a, b)$  and for all  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ ,  
 $\exists c \in D_k$  s.t.  $C_{ik}(a, c) \wedge C_{jk}(b, c)$ .
- A binary CN is **restricted path consistent** iff it is 1-restricted path consistent.
- A binary CN is **Max-restricted path consistent** iff  
 $\forall i \in \mathcal{X}, D_i$  is a non empty arc consistent domain and,  
 $\forall (i, a) \in D_i$  for all  $j \in \mathcal{X}$  linked to  $i$ ,  
 $\exists b \in D_j$  s.t.  $C_{ij}(a, b)$  and for all  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ ,  
 $\exists c \in D_k$  s.t.  $C_{ik}(a, c) \wedge C_{jk}(b, c)$ .
- A binary CN is **path inverse consistent** iff it is (1, 2)-consistent i.e.  $\forall (i, a) \in D_i \forall j, k \in \mathcal{X}$  s.t.  $j \neq i \neq k \neq j, \exists (j, b) \in D_j$  and  $(k, c) \in D_k$  s.t.  $C_{ij}(a, b) \wedge C_{ik}(a, c) \wedge C_{jk}(b, c)$
- A binary CN is **neighborhood inverse consistent** iff  $\forall (i, a) \in D_i, (i, a)$  can be extended to a consistent instantiation including the neighborhood of  $i$ .
- A binary CN  $P$  is **singleton arc consistent** iff  $\forall i \in \mathcal{X}, D_i \neq \emptyset$  and  $\forall (i, a) \in D_i, P|_{D_i=\{a\}}$  has an arc consistent sub-domain.
- A binary CN  $P$  is **singleton restricted path consistent** iff  $\forall i \in \mathcal{X}, D_i \neq \emptyset$  and  $\forall (i, a) \in D_i, P|_{D_i=\{a\}}$  has a restricted path consistent sub-domain.

**Fig. 1.** The most practicable local consistencies

a constraint (which can be expensive if  $k$  is great). The properties corresponding to RPC,  $k$ -RPC, Max-RPC and the most usual local consistencies are presented in fig.1.

In order to compare the pruning efficiency of local consistencies, we use the transitive relation “stronger” introduced in [5]. A local consistency  $LC$  is *stronger* than another local consistency  $LC'$  if in any CN in which  $LC$  holds,  $LC'$  holds too. For example RPC is stronger than AC since an RPC algorithm removes at least all the arc inconsistent values. A local consistency  $LC$  is *strictly stronger* than another local consistency  $LC'$  if  $LC$  is stronger than  $LC'$  and there is at least one CN in which  $LC'$  holds and  $LC$  does not hold.

**Theorem 1.** *If  $k > k' \geq 0$ ,  $k$ -RPC is strictly stronger than  $k'$ -RPC.*

*Proof.* Trivial. ■

**Theorem 2.** *Max-RPC is strictly stronger than  $k$ -RPC,  $\forall k \geq 0$ .*

*Proof.* Trivial. ■

**Theorem 3.** *Singleton arc consistency is stronger than Max-RPC.*

*Proof.* Suppose that there exists a CN  $P$  with a singleton arc consistent value  $(i, a)$  that is not max-restricted path consistent. Let  $j \in \mathcal{X}$  be a variable such that  $(i, a)$  has no path consistent support in  $D_j$ . For each support  $b$  of  $(i, a)$  in  $D_j$ , there exists a variable  $k$  such that  $\nexists c \in D_k / C_{ik}(a, c) \wedge C_{jk}(b, c)$ . Therefore, all the values of  $D_j$  are arc inconsistent w.r.t.  $P|_{D_i=\{a\}}$  and  $(i, a)$  is not singleton arc consistent. ■

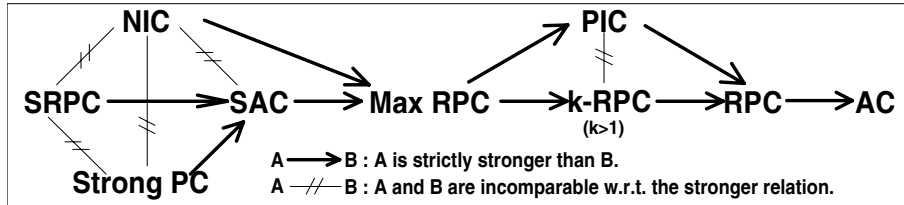
**Theorem 4.** *Neighborhood inverse consistency is stronger than max-restricted path consistency.*

*Proof.* Let us show that a neighborhood inverse consistent value  $(i, a)$  is max-restricted path consistent. If NIC holds for  $(i, a)$ , there exists a consistent instantiation  $I$  including the neighborhood of  $i$ .  $\forall j \in \mathcal{X}$  linked to  $i$ ,  $I_j$  is a path consistent support of  $(i, a)$  since  $\forall k \in \mathcal{X}$  linked to  $i$  and  $j$ ,  $I_k$  is a support of both  $(i, a)$  and  $(j, I_j)$ . Therefore, Max-RPC holds for  $(i, a)$ . ■

**Theorem 5.** *If  $|\mathcal{X}| \geq 3$ , max-restricted path consistency is stronger than path inverse consistency.*

*Proof.* Suppose that there exists a CN  $P$  that is max-restricted path consistent but not path inverse consistent. Let  $(i, a)$  be a Max-RPC value of  $P$  that is not path inverse consistent and  $j, k$  two variables such that  $(i, a)$  cannot be extended to a consistent instantiation of  $\{i, j, k\}$ .

- If  $\exists C_{ij} \in \mathcal{C}$  and  $\exists C_{ik} \in \mathcal{C}$ : Since PIC does not hold,  $\exists(b, c) \in C_{jk}$  and all the values of  $D_j$  are max-restricted path inconsistent. So, Max-RPC does not hold.
- If  $\exists C_{jk} \in \mathcal{C}$ :  $(i, a)$  is not arc consistent and so  $(i, a)$  is max-restricted path inconsistent.
- If  $C_{jk} \in \mathcal{C} \wedge \exists C_{ij} \in \mathcal{C}$  (resp.  $C_{jk} \in \mathcal{C} \wedge \exists C_{ik} \in \mathcal{C}$ ): If  $(i, a)$  has no support in  $D_k$  (resp.  $D_j$ ) it is max-restricted path inconsistent. Otherwise, all the supports of  $(i, a)$  in  $D_k$  (resp.  $D_j$ ) have no support in  $D_j$  (resp.  $D_k$ ). So a Max-RPC algorithm will delete all the supports of  $(i, a)$  in  $D_k$  (resp.  $D_j$ ) and then  $(i, a)$ .
- If  $C_{ij} \in \mathcal{C}$ ,  $C_{jk} \in \mathcal{C}$  and  $C_{ik} \in \mathcal{C}$ : Since  $(i, a)$  cannot be extended to a consistent instantiation of  $\{i, j, k\}$ ,  $(i, a)$  has no path consistent support in  $D_j$  and Max-RPC does not hold. ■



**Fig. 2.** Relations between the local consistencies

Fig.2 sums up the relations between the most practicable filtering techniques. A continuous arrow from  $A$  to  $B$  means that the local consistency  $A$  is strictly stronger than  $B$ . There is a crossed line between  $A$  and  $B$  if  $A$  and  $B$  are incomparable w.r.t. the stronger relation. A proof of the relations presented in fig.2 can be found in [6]. Especially, if  $A$  is not stronger than  $B$  ( $B$  is strictly stronger than  $A$  or  $A$  and  $B$  are incomparable), a CN in which  $A$  holds and  $B$  does not hold can be found in [6] and [4].

Some local consistencies are incomparable with respect to the “strong” relation. Moreover, fig.2 gives some qualitative properties, but no quantitative information. It can be interesting to determine if a local consistency can detect much more inconsistent values than another local consistency.

In order to determine the pruning efficiency of  $k$ -RPC, an experimental evaluation has been done. The aim of this evaluation is not to compare the cpu time to number of value deletions ratio. A part of such a comparison can be found in [5]. We only want to show how much a particular local consistency is able to detect inconsistency on some random CNs, with a fixed number of variables and values, when the number of constraints and the constraints tightness are varying. The CN generator involves four parameters:  $N$  the number of variables,  $D$  the common size of the initial domains,  $p_1$  the proportion of constraints in the network ( $p_1=1$  corresponds to the complete graph) and  $p_2$  the proportion of forbidden pairs of values in a constraint (the tightness). For each possible pair  $(p_1, p_2)$ , 50 random CNs having 100 variables and 20 values in each domain were generated. For each local consistency and each density, fig.3 presents the value of  $p_2$  such that for any tightness greater than this value, the filtering technique has detected the inconsistency of the 50 generated CNs. As an example, for SAC the limit is 0.6 at density 0.15. Therefore, for a smaller tightness, at least one of the 50 random CNs is singleton arc consistent. NIC [7] has an exponential worst case time complexity and becomes really prohibitive when the variables have large neighborhoods. Therefore, this experimental evaluation gives no results on NIC. Path consistency being widely studied, strong path consistency (enforcing both arc and path consistency) is also presented although its huge space and time complexities make it prohibitive on large CNs.

Obviously, Max-RPC removes much more values than AC. But the main result is that 2-RPC is not stronger than PIC because of some very unusual CNs. PIC has only detected the inconsistency of a few 2-restricted path consistent CNs for density between 0.09 and 0.16.

#### 4 Some recalls on RPC1

To enforce RPC we have to achieve arc consistency and to check the path consistency of  $((i, a), (j, b))$  pairs of values such that  $(j, b)$  is the unique support of  $(i, a)$ . RPC1 [1] determines the pairs of values that have to be considered using AC-4, which maintains the number of supports that the values have on each constraint. This determination does not require this counting. We have to look for only the arc consistent values that have no more than one support on a constraint. Whatever is the constraint network, RPC1 performs all possible constraint checks to build its lists of supported values and to initialize its counters. RPC1 has a bad average time complexity because most of this costly initialisation phase is useless, especially on networks with loose constraints.

The  $O(ed^2)$  worst case space complexity of the lists of supported values is another disadvantage. These lists are often more costly in space than the propagation list  $List_{PC}$  used by RPC1 because the average space complexity of  $List_{PC}$  is far from its  $O(end)$  worst case space complexity.

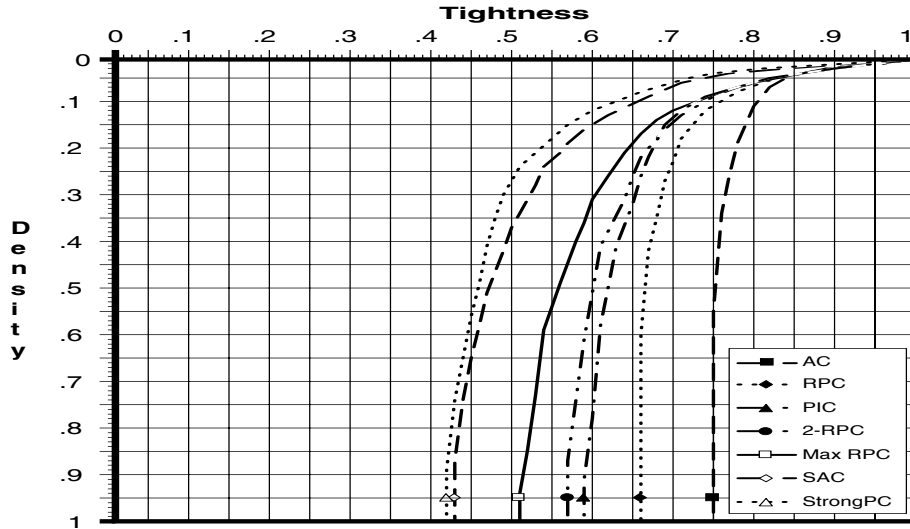


Fig. 3. Evaluation of inconsistency detection on random CNs with  $N=100$  and  $D=20$

But the most important drawback of RPC1 is its worst case time complexity. When a common support  $(k, c)$  of  $(i, a)$  and  $(j, b)$  is found, this information is not stored. Therefore, if a value  $(k, c')$  is deleted, RPC1 cannot determine the set of pairs of values that may be no longer path consistent because of  $(k, c')$  deletion. It overestimates this set. So, the deletion of a value can lead to some useless path consistency checks. In addition, RPC1 does not store enough information to know the values that have already been considered during some previous path consistency checks. Therefore, to check if a pair of values is still path consistent, RPC1 considers again some already checked values. This leads to an  $O(end^3)$  worst case time complexity.

## 5 RPC2

### 5.1 Bases of the algorithm

RPC2 enforces AC and determines the pairs of values for which path consistency has to be proved by checking for each arc-value pair  $[(i, j), a]$  if  $(i, a)$  has zero, one or at least two supports in  $D_j$ . If  $(i, a)$  has no compatible value in  $D_j$  it is an arc inconsistent value, otherwise, if it has an unique support  $b$ , the path consistency of  $((i, a), (j, b))$  has to be checked. In addition, like AC-7 [2], RPC2 takes advantage of the bidirectionality of constraints to reduce the number of constraint checks performed.

The second idea is that when the path consistency of a pair  $((i, a), (j, b))$  has to be checked, for each variable  $k$  linked to  $i$  and  $j$ , RPC2 looks for the smallest common support of  $(i, a)$  and  $(j, b)$  in  $D_k$ . If the pair is path consistent,

RPC2 stores for each smallest common support  $(k, c)$  found, that it is currently supporting  $((i, a), (j, b))$ . So, as long as  $c$  is in  $D_k$ ,  $((i, a), (j, b))$  is path consistent with respect to  $k$  and if  $(k, c)$  has to be deleted, we know that if a common support of  $(i, a)$  and  $(j, b)$  exists in  $D_k$  it is greater than  $c$ . This “AC-6 like behavior” [3] leads to an  $O(end^2)$  worst case time complexity.

## 5.2 The algorithm

The data structures of RPC2 are:

- each initial domain is considered as the integer range  $1..|D_i|$ . The current domain is represented by a table of booleans. We use the following constant time functions and procedures to handle the current domain:
  - $last(D_i)$  returns the greatest value of  $D_i$  if  $D_i \neq \emptyset$  and  $nil$  otherwise.
  - if  $a \in D_i \setminus last(D_i)$ ,  $next(D_i, a)$  returns the smallest value in  $D_i$  greater than  $a$ .  $next(D_i, nil)$  returns the lowest value of  $D_i$  if  $D_i \neq \emptyset$  and  $nil$  otherwise.
  - $remove(D_i, a)$  removes the value  $a$  from  $D_i$  and stops the algorithm if  $a$  was the unique value in  $D_i$  (the CN is inconsistent).
- a pair of values  $(b, a')$  is in  $S_{ij_a}^{AC}$  if  $(j, b)$  is currently supported by  $(i, a)$ . If  $a' = nil$   $(i, a)$  is the unique support of  $(j, b)$  in  $D_i$ , otherwise  $(i, a')$  is the second current support of  $(j, b)$  and there is a direct access between the pair  $(b, a')$  in  $S_{ij_a}^{AC}$  and a pair  $(b, a)$  in  $S_{ij_{a'}}^{AC}$ . If  $S_{ij_a}^{AC} \neq \emptyset$ ,  $first(S_{ij_a}^{AC})$  returns the first pair of values in  $S_{ij_a}^{AC}$  and  $(nil, nil)$  otherwise. If  $(b, a')$  is in  $S_{ij_a}^{AC}$  and is not the last pair of values of  $S_{ij_a}^{AC}$ ,  $next(S_{ij_a}^{AC}, (b, a'))$  returns the successor of  $(b, a')$  in  $S_{ij_a}^{AC}$  and  $(nil, nil)$  otherwise.
- if we do not consider the constraint checks required to check the path consistency of the pairs of values, RPC2 never performs a constraint check twice. To ensure this property, it uses the array  $L$ .  $L_{ija} = b$  if  $\forall b' \in D_j$  s.t.  $b' \leq b$ , RPC2 has already checked if  $(j, b')$  is a support of  $(i, a)$ .
- if  $((i, a), (j, b)) \in S_{k_c}^{PC}$  and  $(a \in D_i \wedge b \in D_j)$ ,  $(j, b)$  is the unique support of  $(i, a)$  in  $D_j$  and  $(k, c)$  is currently supporting  $((i, a), (j, b))$  i.e.  $(k, c)$  is the smallest value in  $D_k$  such that  $C_{ik}(a, c)$  and  $C_{jk}(b, c)$ .
- an arc-value pair  $[(i, j), a]$  is in  $InitList$  if RPC2 has not yet determined if  $(i, a)$  has 0, 1 or at least 2 supports in  $D_j$ . A value  $(j, b)$  is in  $DeletionList$  if  $b$  has been removed from  $D_j$  but this deletion has not been propagated.  $(i, a, j, b, nil, nil)$  is in  $CheckPCList$  if  $(j, b)$  is the unique support of  $(i, a)$  in  $D_j$  and the path consistency of  $((i, a), (j, b))$  has to be checked to determine if  $(i, a)$  is restricted path consistent.  $(i, a, j, b, k, c)$  with  $k \neq nil$  and  $c \neq nil$  is in  $CheckPCList$  if  $c$  has been removed from  $D_k$  and a support of  $((i, a), (j, b))$  greater than  $c$  has to be found in  $D_k$  to prove the path consistency of  $((i, a), (j, b))$  w.r.t.  $k$ .

For each arc-value pair  $[(i, j), a]$ , RPC2 uses the function *TryToFindTwo-Supports* to determine if  $(i, a)$  has zero, one or at least two supports in  $D_j$ . This function tries first to infer two supports looking for undeleted values in

---

```

procedure RPC2();
1 DeletionList  $\leftarrow$   $\emptyset$ ; CheckPCList  $\leftarrow$   $\emptyset$ ; InitList  $\leftarrow$   $\emptyset$ ;
2 forall  $(i, a) \in D$  do
3    $S_{ia}^{PC} \leftarrow \emptyset$ ;
4   forall  $C_{ij} \in \mathcal{C}$  do
5      $S_{ija}^{AC} \leftarrow \emptyset$ ;  $L_{ija} \leftarrow nil$ ; InitList  $\leftarrow$  InitList  $\cup$   $\{(i, j), a\}$ ;
6   while InitList  $\neq \emptyset$  or DeletionList  $\neq \emptyset$  or CheckPCList  $\neq \emptyset$  do
7     if DeletionList  $\neq \emptyset$  then
8       choose and delete  $(i, a)$  from DeletionList;
9       PropagDeletion( $i, a, DeletionList, CheckPCList$ );
10    else if CheckPCList  $\neq \emptyset$  then
11      choose and delete  $(i, a, j, b, k, c)$  from CheckPCList;
12      if  $a \in D_i$  and  $b \in D_j$  then
13        CheckPC( $i, a, j, b, k, c, DeletionList$ );
14      else choose and delete  $(i, j), a$  from InitList;
15       $b \leftarrow nil$ ;  $NbS \leftarrow TryToFindTwoSupports(i, a, j, b)$ ;
16      if  $NbS = 0$  then
17        remove( $D_i, a$ ); DeletionList  $\leftarrow$  DeletionList  $\cup$   $\{(i, a)\}$ ;
18      else if  $NbS = 1$  then
19        CheckPCList  $\leftarrow$  CheckPCList  $\cup$   $\{(i, a, j, b, nil, nil)\}$ ;

```

---

Fig. 4. RPC2

$S_{ija}^{AC}$  i.e. the list of the values supported by  $(i, a)$  on  $C_{ij}$ . If less than two supports have been found, RPC2 goes on with its search looking for the smallest supports in  $D_j$ . The array  $L$  allows to reduce the number of constraint checks performed.  $L_{ija}$  is used to determine the values of  $D_j$  that have not already been checked and we check  $C_{ij}(a, b)$  only if  $L_{jib} < a$ . Indeed, if  $L_{jib} \geq a$ , RPC2 has already checked if  $(i, a)$  is a support of  $(j, b)$ , and if it is a compatible value *TryToFindTwoSupport* has found  $b$  in  $S_{ija}^{AC}$ .

If  $(i, a)$  has an unique support  $(j, b)$  the path consistency of  $((i, a), (j, b))$  has to be checked and  $(i, a, j, b, nil, nil)$  is put in *CheckPCList*. To check the path consistency of a pair  $((i, a), (j, b))$  the procedure *CheckPC* uses the function *IsPathConsistent* to find the smallest common support of  $(i, a)$  and  $(j, b)$  in  $D_k$  for all  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ . If path consistency is proved, for each smallest common support  $(k, c)$  found, RPC2 stores that it is currently supporting  $((i, a), (j, b))$  by updating  $S_{kc}^{PC}$ , otherwise  $(i, a)$  is not restricted path consistent.

If a value  $(j, b)$  is deleted, *PropagDeletion* checks if the values currently supported by  $(j, b)$  (in  $S_{j*b}^{AC}$ ) are still restricted path consistent and if the pairs of values for which  $(j, b)$  is the smallest common support (in  $S_{j*b}^{PC}$ ) are still path consistent.

### 5.3 Complexity

Since *TryToFindTwoSupports* removes from  $S_{ija}^{AC}$  the values that are no longer in  $D_j$ , the test of line 4 is performed at most  $O(d)$  times for each arc-value pair.

In addition  $L_{ija}$  is bounded above by  $d$  and  $L_{ija}$  increases at each step of the second loop of *TryToFindTwoSupports*. Thus, the cost of this loop is  $O(d)$  for each arc-value pair and the complexity due to the calls to *TryToFindTwoSupports* is  $O(ed^2)$ . The pairs of values  $((i, a), (j, b))$  for which the path consistency has to be proved are such that  $(j, b)$  is the unique support of  $(i, a)$ . So, in the worst case path consistency has to be checked for  $O(ed)$  pairs of values. Whatever is the pair of values  $((i, a), (j, b))$  and  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ ,

---

```

function TryToFindTwoSupports(i, a, j, var b) : integer;
1 if b = nil then NbS  $\leftarrow$  0 else NbS  $\leftarrow$  1;
2 (b', a')  $\leftarrow$  first(SijaAC);
3 while NbS < 2 and b' ≠ nil do
4   if b' ∉ Dj then
5     delete (b', a') from SijaAC
6   else if b' ≠ b then
7     NbS  $\leftarrow$  NbS + 1;
8     if NbS = 1 then b  $\leftarrow$  b';
9   if NbS < 2 then
10    (b', a')  $\leftarrow$  next(SijaAC, (b', a'));
11 while NbS < 2 and Lija < last(Dj) do
12   b'  $\leftarrow$  next(Dj, Lija); Lija  $\leftarrow$  b';
13   if b' ≠ b and (Ljib' = nil or Ljib' < a) then
14     if Cij(a, b') then
15       NbS  $\leftarrow$  NbS + 1;
16       if NbS = 1 then b  $\leftarrow$  b';
17 if NbS = 2 then
18   add (a, b') in SjibAC, (a, b) in Sjib'AC and link them
19 else if NbS = 1 then
20   add (a, nil) in SjibAC;
21 return NbS;

procedure PropagDeletion(j, b, var DeletionList, var CheckPCList);
1 forall i  $\in$   $\mathcal{X}$  such that Cij  $\in$   $\mathcal{C}$  do
2   while SjibAC  $\neq \emptyset$  do
3     choose and delete (a, b') from SjibAC;
4     if a  $\in$  Dj then
5       if b' = nil then
6         remove(Di, a); DeletionList  $\leftarrow$  DeletionList  $\cup$  {(i, a)}
7       else delete (a, b) from Sjib'AC; {constant time}
8       if b' ∉ Dj then b'  $\leftarrow$  nil;
9       NbS  $\leftarrow$  TryToFindTwoSupports(i, a, j, b');
10      if NbS = 0 then
11        remove(Di, a); DeletionList  $\leftarrow$  DeletionList  $\cup$  {(i, a)}
12      else if NbS = 1 then
13        CheckPCList  $\leftarrow$  CheckPCList  $\cup$  {(i, a, j, b', nil, nil)};
14 while SjbPC  $\neq \emptyset$  do
15   choose and delete ((i, a), (k, c)) from SjbPC;
16   CheckPCList  $\leftarrow$  CheckPCList  $\cup$  {(i, a, k, c, j, b)};

procedure CheckPC(i, a, j, b, k, c, var DeletionList);
1 if k = nil then
2   PCconsistent  $\leftarrow$  true; Common  $\leftarrow$  {k  $\in$   $\mathcal{X}$  | Cik  $\in$   $\mathcal{C}$  and Cjk  $\in$   $\mathcal{C}$ };
3   forall k  $\in$  Common while PCconsistent do
4     c'  $\leftarrow$  nil; PCconsistent  $\leftarrow$  IsPathConsistent(i, a, j, b, k, c'); CS[k]  $\leftarrow$  c';
5   if PCconsistent then
6     forall k  $\in$  Common do
7       Sk,CSPC  $\leftarrow$  Sk,CSPC  $\cup$  {(i, a), (j, b)};
8   else PCconsistent  $\leftarrow$  IsPathConsistent(i, a, j, b, k, c);
9   if PCconsistent then
10    SkcPC  $\leftarrow$  SkcPC  $\cup$  {(i, a), (j, b)};
11 if not PCconsistent then
12   remove(Di, a); DeletionList  $\leftarrow$  DeletionList  $\cup$  {(i, a)};

function IsPathConsistent(i, a, j, b, k, var c) : boolean;
1 found  $\leftarrow$  false;
2 while (not found) and c  $\neq$  last(Dk) do
3   c  $\leftarrow$  next(Dk, c);
4   if Cik(a, c) and Cjk(b, c) then found  $\leftarrow$  true;
5 return found;

```

---

Fig. 5. The subprocedures used by RPC2

a value of  $D_k$  is never checked twice to prove the path consistency of  $((i, a), (j, b))$  w.r.t.  $k$ . Thus, the complexity due to the calls to *IsPathConsistent* is  $O(\epsilon nd^2)$  and the worst case time complexity of RPC2 is  $O(\epsilon nd^2)$ .

The worst case space complexity of  $S_{ij}^{AC}$  lists is  $O(\epsilon d)$  because a value  $(i, a)$  has at most two current supports on each constraint  $C_{ij}$ . The size of *InitList* is  $O(\epsilon d)$  since each arc-value pair is put in this list once. A pair of values  $((i, a), (j, b))$  such that  $(j, b)$  is the unique support of  $(i, a)$  has at most one current support in the domain of each variable linked to both  $i$  and  $j$ . Therefore the worst case space complexity of  $S_{jb}^{PC}$  lists is  $O(\epsilon nd)$ . There is at most  $O(\epsilon d)$   $(i, a, j, b, nil, nil)$  elements in *CheckPCList*, and whatever is the pair of values  $((i, a), (j, b))$  and the variable  $k$  linked to both  $i$  and  $j$ , there is at most one  $(i, a, j, b, k, c)$  element in *CheckPCList*. Thus the size of *CheckPCList* is  $O(\epsilon nd)$  and the worst case space complexity of RPC2 is  $O(\epsilon nd)$ .

## 6 Max-RPC

### 6.1 Bases of the algorithm

Max-RPC does not have to determine the set of weakly supported values i.e. those having at most  $k$  supports. It only has to ensure that all the values have at least one path consistent support on each constraint. The idea of AC-6 is used twice. First, to prove the Max-restricted path consistency of a value  $(i, a)$ , Max-RPC looks in the domain of each variable linked to  $i$  for the smallest path consistent support this value has. To determine if a value  $(j, b)$  compatible with  $(i, a)$  is a path consistent support, Max-RPC looks for the smallest common support of  $(i, a)$  and  $(j, b)$  in the domain of each variable  $k$  linked to  $i$  and  $j$ .

### 6.2 The algorithm

The data structures of Max-RPC are:

- the same representation of the domains as RPC2.
- the values for which  $(j, b)$  is the smallest path consistent support are stored in the list  $S_{jb}^{AC}$ .
- the path consistent pairs of values  $((i, a), (j, b))$  such that  $(k, c)$  is the smallest common support of  $(i, a)$  and  $(j, b)$  in  $D_k$  are stored in the list  $S_{kc}^{PC}$ .
- as in RPC2, the deleted values for which the deletion has not been propagated yet are put in *DeletionList*. An arc-value pair  $[(i, j), a]$  is in *InitList* if Max-RPC has not verified if  $(i, a)$  has a path consistent support in  $D_j$ .

*IsWithoutPCSupport* $(i, a, j, b)$  is used to determine if  $(i, a)$  has a path consistent support greater than  $b$  in  $D_j$ . It looks for the smallest support  $(j, b')$  of  $(i, a)$  such that  $((i, a), (j, b'))$  is path consistent. If such a support exists,  $(i, a)$  is put in  $S_{jb'}^{AC}$  in order to store that  $(j, b')$  is currently supporting  $(i, a)$ . In addition, for each  $k \in \mathcal{X}$  linked to  $i$  and  $j$ ,  $((i, a), (j, b'))$  is put in  $S_{kc}^{PC}$  where

---

```

procedure Max-RPC();
1 DeletionList  $\leftarrow$   $\emptyset$ ; InitList  $\leftarrow$   $\emptyset$ ;
2 forall  $(i, a) \in D$  do
3    $S_{ia}^{PC} \leftarrow \emptyset$ ;  $S_{ia}^{AC} \leftarrow \emptyset$ ;
4   forall  $C_{ij} \in \mathcal{C}$  do
5     InitList  $\leftarrow$  InitList  $\cup$   $\{(i, j), a\}$ ;
6 while InitList  $\neq \emptyset$  or DeletionList  $\neq \emptyset$  do
7   if DeletionList  $\neq \emptyset$  then
8     choose and delete  $(i, a)$  from DeletionList;
9     PropagDeletion( $i, a, DeletionList$ );
10  else choose and delete  $[(i, j), a]$  from InitList;
11  if IsWithoutPCSupport( $i, a, j, nil$ ) then
12    remove( $D_i, a$ ); DeletionList  $\leftarrow$  DeletionList  $\cup$   $\{(i, a)\}$ ;

function IsWithoutPCSupport( $i, a, j, b$ ) : boolean;
1 Common  $\leftarrow$   $\{k \in \mathcal{X} \mid C_{ik} \in \mathcal{C} \text{ and } C_{jk} \in \mathcal{C}\}$ ;
2 WithoutPCSupport  $\leftarrow$  true;  $b' \leftarrow b$ ;
3 while  $b' \neq last(D_j)$  and WithoutPCSupport do
4    $b' \leftarrow next(D_j, b')$ ;
5   if  $C_{ij}(a, b')$  then
6     PCconsistent  $\leftarrow$  true;
7     for  $k \in Common$  while PCconsistent do
8        $c \leftarrow nil$ ;
9       if IsPathConsistent( $i, a, j, b', k, c$ ) then
10         $CS[k] \leftarrow c$ ;
11        else PCconsistent  $\leftarrow$  false;
12  if PCconsistent then
13    forall  $k \in Common$  do
14       $S_{k,CS[k]}^{PC} \leftarrow S_{k,CS[k]}^{PC} \cup \{(i, a), (j, b')\}$ ;
15       $S_{jb'}^{AC} \leftarrow S_{jb'}^{AC} \cup \{(i, a)\}$ ;
16      WithoutPCSupport  $\leftarrow$  false;
17 return WithoutPCSupport;

procedure PropagDeletion( $j, b, var DeletionList$ );
1 while  $S_{jb}^{AC} \neq \emptyset$  do
2   choose and delete  $(i, a)$  from  $S_{jb}^{AC}$ ;
3   if  $a \in D_i$  and IsWithoutPCSupport( $i, a, j, b$ ) then
4     remove( $D_i, a$ ); DeletionList  $\leftarrow$  DeletionList  $\cup$   $\{(i, a)\}$ ;
5 while  $S_{jb}^{PC} \neq \emptyset$  do
6   choose and delete  $((i, a), (k, c))$  from  $S_{jb}^{PC}$ ;
7   if  $a \in D_i$  and  $c \in D_k$  and  $(i, a) \in S_{kc}^{AC}$  then
8      $b' \leftarrow b$ ;
9     if IsPathConsistent( $i, a, k, c, j, b'$ ) then
10       $S_{jb'}^{PC} \leftarrow S_{jb'}^{PC} \cup \{(i, a, k, c)\}$ ;
11    else remove  $(i, a)$  from  $S_{kc}^{AC}$ ;
12    if IsWithoutPCSupport( $i, a, k, c$ ) then
13      remove( $D_i, a$ ); DeletionList  $\leftarrow$  DeletionList  $\cup$   $\{(i, a)\}$ ;

```

---

**Fig. 6.** Max-RPC

$(k, c)$  is the smallest common support of  $(i, a)$  and  $(j, b')$  found in  $D_k$ . As long as  $c$  is in  $D_k$ ,  $((i, a), (j, b'))$  is path consistent w.r.t.  $k$ .

If a value  $(j, b)$  is deleted, for each value  $(i, a)$  currently supported by  $(j, b)$  another path consistent support has to be found in  $D_j$  to prove that Max-RPC holds for  $(i, a)$ . If such a support exists it is greater than  $b$  since this value was the smallest path consistent support. *PropagDeletion*( $j, b$ ) has also to check for each pair of value  $((i, a), (k, c))$  supported by  $(j, b)$  if  $(k, c)$  is still a path consistent support of  $(i, a)$ . If  $((i, a), (k, c))$  is still path consistent, there is a common support of  $(i, a)$  and  $(k, c)$  greater than  $b$  in  $D_j$ . Otherwise, we have to look for another path consistent support greater than  $c$  in  $D_k$  for  $(i, a)$ . If there is not any such support,  $(i, a)$  is not max-restricted path consistent and must be removed.

### 6.3 Complexity

If Max-RPC has to look for a path consistent support of a value  $(i, a)$  in  $D_j$ , it considers only the values it has not already checked i.e. those greater than the current support of  $(i, a)$  on  $C_{ij}$ . In the worst case, the path consistency of  $O(ed^2)$  pairs of values is checked. In addition, whatever is the pair of values  $((i, a), (j, b))$  and  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ , each value of  $D_k$  is checked by *IsPathConsistent* at most once to determine if it is the smallest support of  $((i, a), (j, b))$  in  $D_k$ . Thus, the worst case time complexity of Max-RPC is  $O(end^3)$ . Although the worst case time complexity of the best path consistency algorithm (PC-5 [11]) is  $O(n^3d^3)$ , enforcing Max-RPC is really less expensive than achieving PC. Indeed, the number of constraints  $e$  can be far from  $n^2$  on sparse CNs. Furthermore, Max-RPC looks for only one path consistent support for each value on each constraint. So, although in the worst case the path consistency of  $O(ed^2)$  pairs of values has to be checked, Max-RPC will check much less than this upper bound in practice.

A value  $(i, a)$  has at most one current support on each constraint  $C_{ij}$  and the size of the  $S_{j_b}^{A_C}$  lists is  $O(ed)$ . For each value  $(j, b)$  currently supporting  $(i, a)$ ,  $((i, a), (j, b))$  is in at most one  $S_{k_c}^{PC}$  list for each  $k \in \mathcal{X}$  linked to both  $i$  and  $j$ . Therefore, the size of the  $S^{PC}$  data structure is  $O(end)$  and the worst case space complexity of Max-RPC is  $O(end)$ .

## 7 Experimental evaluation

The generator of the section 3 has been used to evaluate the efficiency of RPC1, RPC2 and Max-RPC. All the generated CNs have 100 variables and 20 values in each initial domain. Fig.7 shows the results for both relatively sparse and dense CNs. For each tightness, 250 instances were generated and fig.7 presents mean values. To give an evaluation of the space required, we sum the number of counters used and the effective maximal size of each list used.

On dense CNs RPC2 outperforms RPC1 in cpu time. On sparse CNs, RPC2 requires less cpu time than RPC1, but when the lists of supported values of RPC1 are very short and when the path consistency of many pairs of values has to be checked, RPC1 can outperform RPC2. Such a situation arise for tightness between 0.63 and 0.87 at density 0.04. But if the path consistency of many pairs of values has to be checked to enforce RPC in a more dense CN, the size of the lists of supported values of RPC1 is more important and RPC2 outperforms RPC1. This can be observed for tightness greater than 0.65 at density 0.25 where the non optimal worst case time complexity of RPC1 is a real drawback. RPC2 always significantly overcomes RPC1 if we consider the number of constraint checks and list checks performed, or the space required.

Obviously in dense CNs, enforcing Max-RPC can be much more expensive than achieving RPC. But Max-RPC is stronger than RPC, and as soon as Max-RPC detects many inconsistent values it becomes less expensive than RPC2. In addition, the seven seconds required for tightness 0.6 at density 0.25 remain very small when compared to the twenty one minutes required to enforce singleton

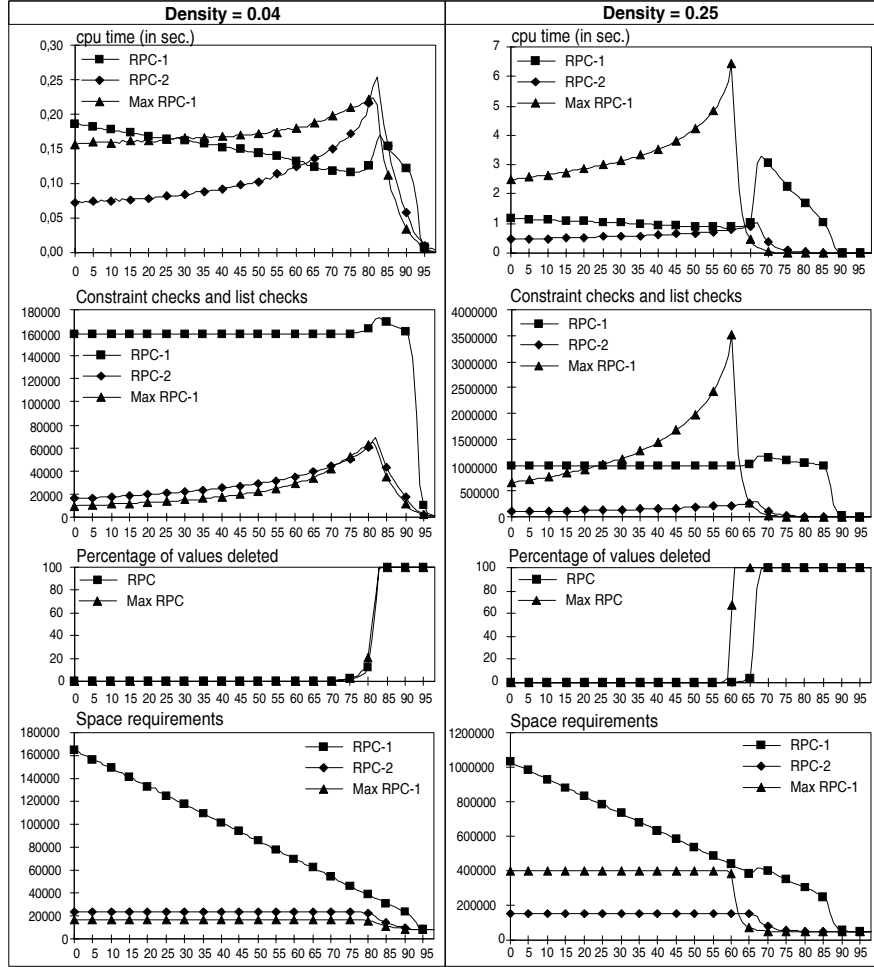


Fig. 7. Comparison of RPC1, RPC2 and Max-RPC on random CNs with  $n=100$  and  $d=20$

arc consistency at tightness 0.53 and the five hours required to achieve path consistency at tightness 0.52. This shows that the average time complexity of Max-RPC is far from its  $O(\epsilon nd^3)$  worst case time complexity. Moreover, Max-RPC has the same worst case space complexity as RPC1 or RPC2, and although it requires more space than RPC2, on all the generated CNs RPC1 is more expensive in space than Max-RPC.

## 8 Conclusion

In this paper we have extended restricted path consistency to  $k$ -RPC and Max-RPC. These new local consistencies are more pruningful than RPC while avoiding the drawbacks of path consistency. Two new algorithms have been proposed. A RPC algorithm called RPC2 with an  $O(\epsilon nd^2)$  worst case time complexity, and a

Max-RPC algorithm based on AC-6 with an  $O(end^3)$  worst case time complexity, both having an  $O(end)$  worst case space complexity. An experimental evaluation shows that RPC2 has better cpu time performances and performs less constraint checks than RPC1. Moreover, these experiments highlight that Max-RPC has good average cpu time performances in spite of its  $O(end^3)$  worst case time complexity and although it detects much more inconsistent values than RPC.

## References

1. Berlandier, P.: Improving Domain Filtering using Restricted Path Consistency. In proceedings of IEEE CAIA-95, Los Angeles CA (1995)
2. Bessière, C., Freuder, E.C., Régin, J.C.: Using inference to reduce arc-consistency computation. In proceedings of IJCAI-95, Montréal, Canada (1995)
3. Bessière, C.: Arc-consistency and arc-consistency again. *Artificial Intelligence* **65** (1984) 179–190
4. Debruyne, R., Bessière, C.: From Restricted Path Consistency to Max-Restricted Path Consistency. Technical Report 97036, Montpellier, France (1997)
5. Debruyne, R., Bessière, C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In proceedings of IJCAI-97, Nagoya, Japan (1997) (to appear)
6. Debruyne, R., Bessière, C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. Technical Report 97035, Montpellier, France (1997)
7. Freuder, E., Elfe, D.C.: Neighborhood Inverse Consistency Preprocessing. In proceedings of AAAI-96, Portland OR (1996) 202–208
8. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* **14** (1980) 263–313
9. Mohr, R., Henderson, T.C.: Arc and Path Consistency Revisited. *Artificial Intelligence* **28** (1986) 225–233
10. Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In Allan Borning, editor, PPCP'94: second workshop on Principles and Practice of Constraint Programming, Seattle WA (1994)
11. Singh, M.: Path Consistency Revisited. In proceedings of IEEE ICTAI-95, Washington D.C. (1995)