A strong local consistency for constraint satisfaction

Romuald Debruyne
Ecole des Mines de Nantes
4, rue Alfred Kastler, La Chantrerie
44307 Nantes cedex 03 - France
Email: debruyne@lirmm.fr

Abstract

Filtering techniques are essential to efficiently look for a solution in a constraint network (CN). However, for a long time it has been considered that to efficiently reduce the search space, the best choice is the limited local consistency achieved by forward checking [15, 17]. However, more recent works [18, 4, 16] show that maintaining arc consistency (which is a more pruningful local consistency) during search outperforms forward checking on hard and large constraint networks. In this paper, we show that maintaining a local consistency stronger than arc consistency during search can be advantageous. According to the comparison of the local consistencies more pruningful than arc consistency that can be used on large CNs in [9], Max-restricted path consistency (Max-RPC, [7]) is one of the most promising local consistencies. We propose a new local consistency, called Max-RPCEn, that is stronger than Max-RPC and that has almost the same cpu time requirements.

1 Introduction

Finding a solution in a constraint network (CN) involves looking for an assignment of values for the problem variables so that all the constraints are simultaneously satisfied. This task is NP-hard, and to avoid a combinatorial explosion, the search space has to be reduced by filtering techniques, which remove some local inconsistencies. Obviously, a given local consistency can advantageously be maintained during search only if it requires less cpu time to detect that a branch of the search tree does not lead to any solution than a search algorithm to explore this branch. For a long time, the only practicable local consistency was arc consistency (AC, namely 2-consistency or (1, 1)-consistency in the formalism of [11]). Indeed, higher levels of k-consistency, such as path consistency (k=3), are so expensive that they can be used only on very small CNs. In the last three years, new local consistencies have been proposed and a comparison of those that can be used on large CNs have been done in [9]. The conclusion of this comparison is that Max-RPC [7] is one of the most worthwhile local consistencies. Experiments show also that the cpu time required to enforce the local consistencies more pruningful than Max-RPC, such as neighborhood inverse consistency [13] and singleton consistencies [6], is not of the same order of magnitude. In this paper we show that Max-RPC is not the limit we have to do not exceed to guarantee a reasonable cpu time. We propose a new local consistency called Max-RPCEn that is significantly more pruningful than Max-RPC while requiring almost no additional cpu time.

2 Definitions and notations

A *network of binary constraints* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined by 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* for (i, a) on C_{ij}. Checking whether a pair of value is allowed by a constraint is called a *constraint check*. With each CN we associate a *constraint graph* in which nodes represent variables and arcs connect pairs of variables that are constrained explicitly. c is the number of 3-cliques in the constraint graph. The *neighborhood* of i is the set of variables linked to i in the constraint graph. An *instantiation* of a set of variables S is a set of value assignments $\{I_j\}_{j \in S}$, one for each variable belonging to S s.t. $\forall j \in S,\ \ I_j \in D_j$. 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 $\{i,\ j,\ k\}$. Checking whether a pair of values is path consistent is called a *path consistency check*. $(j,\ b)$ is a *path consistent support* for $(i,\ a)$ if $(a, b) \in C_{ij}$ and $((i,\ a),\ (j,\ b))$ 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* (or *completable* [12]) if there is a solution I such that $I_i = a$, and a CN is *consistent* if it has at least one solution.

3 From arc consistency to conservative path consistency

Let us recall that a k-consistency algorithm removes the instantiations of length $k-1$ that cannot be extended to a consistent instantiation including any additional k^{th} variable. So, an arc consistency algorithm ($k = 2$) deletes the values that have not at least one compatible value (a support) on each constraint. Higher levels of k-consistency, such as path consistency (PC, $k = 3$), are so expensive that they can be used only on very small CNs. A path consistency algorithm has to try to extend all the pairs of values, even those between two variables that are not linked by a constraint, to any third variable. Thus, even the most efficient PC algorithms [5, 19] are prohibitive. Obviously, enforcing an higher level of k-consistency is even more expensive. Furthermore, if $k > 2$ a k-consistency algorithm changes the structure of the network. Indeed, constraints involving $k-1$ variables may have to be added to store the deletion of the $(k-1)$-inconsistent instantiations. To avoid these important drawbacks, a restricted path consistency algorithm (RPC, [1]) performs only the most pruningful path consistency checks, namely those that can directly lead to the deletion of a value, and it deletes only values in order to keep unchanged the structure of the network. In addition to AC, an RPC algorithm checks the path consistency of the pairs of values $((i,\ a),\ (j,\ b))$ such that $(j,\ b)$ is the only support for $(i,\ a)$ in D_j. If such a pair of values is path inconsistent, its deletion would lead to the arc inconsistency of $(i,\ a)$, and thus $(i,\ a)$ can be removed. So, these few path consistency checks allow to remove more values than arc consistency while leaving unchanged the set of constraints. We can extend the idea of RPC to remove more values by checking the existence of a path consistent support on a constraint not only for the values that have only one support on this constraint as in RPC, but also for the values having at most k supports on this constraint (k-restricted path consistency, k-RPC [7]), or for all the values, whatever is the number of supports they have (Max-restricted path consistency). Considering the pruning efficiency, Max-RPC is an upper bound for k-RPC and a Max-RPC algorithm removes all the k-restricted path inconsistent values for all k. However, we can delete even more values than Max-RPC, still without adding any constraint in the network.

The limit in terms of pruning efficiency of checking the path consistency of pairs of values while keeping the structure of the network is conservative path consistency (CPC). CPC is the restriction of strong path consistency (a strong PC algorithm enforces both arc and path consistency) to the explicit constraints of the network. If there is no constraint between two variables i and j, any pair of values $((i,\ a),\ (j,\ b))$ is conservative path consistent. If i and j are linked by a constraint $C_{ij} \in \mathcal{C}$, a pair of values $((i,\ a),\ (j,\ b))$ allowed by this constraint is conservative path consistent if, and only if, for any third variable $k \in \mathcal{X}$ linked to both i and j, $((i,\ a),\ (j,\ b))$ can be extended to a consistent instantiation including k. A constraint network is conservative path consistent iff it is arc consistent and all the pairs of values allowed by the constraints explicitly present in the network are conservative path consistent (see Fig. 1). CPC is more pruningful than Max-RPC (see Section 5) and does not change the structure of the network, but on complete constraint networks it is as expensive as strong path consistency. Furthermore, in real applications, a constraint is seldom represented by a Boolean matrix or a set of compatible pairs of values. They are often represented by a predicate which has a particular semantics ($\neq, \leq, ...$). Enforcing CPC leads to the generation of the Boolean matrix to store the deletion of the conservative path inconsistent pairs of values and the semantics of the constraints is lost. The drawbacks of CPC are too important, and enforcing it is too expensive to be worthwhile. So, in the following, we study the pruning efficiency of CPC, but we do not try to know the cpu time required to achieve it.

To avoid the drawbacks of CPC while removing more values than an algorithm achieving Max-RPC, the new filtering algorithm proposed in the next section, called Max-RPCEn1, does not try to check the conservative path consistency of all the pairs of values. Max-RPCEn1 does not delete any pair of values and it performs more value deletions than an algorithm achieving Max-RPC only when the enforce-

- A binary CN is *(i, j)-consistent* iff $\forall i \in \mathcal{X}$, $D_i \neq \emptyset$ and any consistent instantiation of i variables can be extended to a 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 $\forall D_i \in D$, $D_i \neq \emptyset$ 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 *restricted path consistent* iff
  $\forall i \in \mathcal{X}$, D_i is a non empty arc consistent domain and,
  $\forall (i,\ a) \in D$, $\forall j \in \mathcal{X}$ s.t. $(i,\ a)$ has only one support b in D_j,
  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 *max restricted path consistent* iff
  $\forall i \in \mathcal{X}$, D_i is a non empty arc consistent domain and,
  $\forall (i,\ a) \in D$, 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 pair of values $((i,\ a),\ (j,\ b))$ such that there is no constraint $C_{ij} \in \mathcal{C}$ is conservative path consistent. If $\exists C_{ij} \in \mathcal{C}$, a pair of values $((i,\ a),\ (j,\ b))$ is conservative path consistent iff $C_{ij}(a,\ b)$ and $\forall k \in \mathcal{X}$ linked to both i and j, $\exists c \in D_k$ s.t. $C_{ik}(a,\ c) \wedge C_{jk}(b,\ c)$. A constraint $C_{ij} \in \mathcal{C}$ is conservative path consistent iff all the pairs of values $((i,\ a),\ (j,\ b))$ s.t. $C_{ij}(a,\ b)$ are conservative path consistent. A CN is *conservative path consistent* iff $\forall (i,\ a) \in \mathcal{D}$, $(i,\ a)$ is arc consistent and $\forall C_{ij} \in \mathcal{C}$, C_{ij} is conservative path consistent.

Figure 1. The mentioned local consistencies

ment of Max-RPC allows it to detect some conservative path inconsistent values.

4 Max-RPCEn1

4.1 Bases of the algorithm

Max-RPCEn1 is an improvement on Max-RPC1. Max-RPC1 is based on the idea of AC6 [2] which is a very efficient arc consistency algorithm but that does not use the bidirectionality of the constraints, namely the property that a value $(i,\ a)$ is a support of another value $(j,\ b)$ if and only if $(j,\ b)$ is a support for $(i,\ a)$. Like AC7 [3], Max-RPCEn1 uses this property of the constraints to infer the existence, or the non-existence of supports. This substantially reduces the number of constraint checks and the cpu time required to enforce the local consistency on most of the CNs. Since all the constraints of a CN are bidirectional, if we find that a value $(j,\ b)$ is a support for $(i,\ a)$, we can infer that $(i,\ a)$ is a support for $(j,\ b)$. Furthermore, if we have found that $(i,\ a)$ is not compatible with any value lower than b in D_j, it is useless to check whether $(i,\ a)$ is compatible with $(j,\ b')$ (with $b' < b$) when we look for a support for $(j,\ b')$ in D_i. Taking advantage of the bidirectionality to achieve Max-RPC leads to even more savings than during the enforcement of AC. Indeed, $(j,\ b)$ is a path consistent support for $(i,\ a)$ if and only if $(j,\ b)$ is a path consistent support for $(i,\ a)$. So, the bidirectionality of the constraints allows not only avoids some constraint checks, but also some path consistency checks.

The second improvement of Max-RPCEn1 on Max-RPC1 is an enhancement of the pruning efficiency. We say that a pair of values $((i,\ a),\ (j,\ b))$ is *valid path consistent* if for all the 3-cliques $\{i,\ j,\ k\}$ of the constraint graph there exists a value $c \in D_k$ compatible with both $(i,\ a)$ and $(j,\ b)$ such that Max-RPCEn1 has not found the conservative path inconsistency of $((i,\ a),\ (k,\ c))$ or $((j,\ b),\ (k,\ c))$. A value $(j,\ b)$ is a valid path consistent support for $(i,\ a)$ if $(j,\ b)$ is compatible with $(i,\ a)$ and $((i,\ a),\ (j,\ b))$ is valid path consistent. The values deleted by Max-RPCEn1 are those that do not have any valid path consistent support on a constraint. So, Max-RPCEn1 removes the Max-restricted path inconsistent values and some of the conservative path inconsistent values. The data structure that allows Max-RPCEn1 to detect the conservative path inconsistency of some pairs of values is the array of counters M. $M_{ija} = b$ if there is no valid path consistent support of $(i,\ a)$ in D_j lower than b. To know whether a pair of value $((i,\ a),\ (j,\ b))$ allowed by a constraint C_{ij} is path consistent w.r.t. a third variable k linked to both i and j, we have to look for a value $c \in D_k$ that is compatible with $(i,\ a)$ and $(j,\ b)$.

```
procedure Max − RPCEn1();
DeletionList ← ∅; InitList ← ∅;
forall (i, a) ∈ D do
    S_ia^PC ← ∅;
    forall C_ij ∈ C do
        S_ija ← ∅; M_ija ← nil;
        InitList ← InitList ∪ {[(i, j), a)]};
forall C_ij ∈ C s.t. i < j do
    Common[C_ij] ← ∅;
    forall C_jk ∈ C do
        if ∃C_ik ∈ C then
            Common[C_ij] ← Common[C_ij] ∪ {k}
    Common[C_ji] ← Common[C_ij];
while IniList ≠ ∅ or DeletionList ≠ ∅ do
    if DeletionList ≠ ∅ then
        choose and delete (i, a) from DeletionList;
        PropagDeletion(i, a, DeletionList);
    else
        choose and delete [(i, j), a] from InitList;
        if (i, a) ∈ D and not
        HasAValidPCSupport(i, a, j) then
            remove(D_i, a);
            DeletionList ← DeletionList ∪{(i, a)};

procedure PropagDeletion(j, b, in out DeletionList);
while S_jb ≠ ∅ do
    choose and delete (i, a) from S_jb;
    if a ∈ D_i and not HasAValidPCSupport(i, a, j) then
        remove(D_i, a);
        DeletionList ← DeletionList ∪ {(i, a)};
while S_jb^PC ≠ ∅ do
    choose and delete ((i, a), (k, c)) from S_jb^PC;
    if a ∈ D_i and c ∈ D_k and a ∈ S_kic then
        b' ← b;
        if IsValidPathConsistent(i, a, k, c, j, b') then
            S_jb'^PC ← S_jb'^PC ∪ {((i, a), (k, c))};
        else
            remove a from S_kic;
            if c ∈ S_ika then
                remove c from S_ika;
                if not HasAValidPCSupport(k, c, i) then
                    remove(D_k, c);
                    DeletionList ← DeletionList ∪ {(k, c)};
            if not HasAValidPCSupport(i, a, k) then
                remove(D_i, a);
                DeletionList ← DeletionList ∪ {(i, a)};
```

Figure 2. Max-RPCEn1.

However, even if such a value c exists, a CPC algorithm can delete $((i,\ a),\ (j,\ b))$ if $((i,\ a),\ (k,\ c))$ or $((j,\ b),\ (k,\ c))$ is conservative path inconsistent. If a support $c \in D_k$ of $(i,\ a)$ is lower than M_{ika} or if $a < M_{ikc}$ then $((i,\ a),\ (k,\ c))$ is conservative path inconsistent. This is the property used by Max-RPCEn1 to detect some conservative path inconsistent pairs of values. Therefore, a pair of values $((i,\ a),\ (j,\ b))$ is valid path consistent if for all the 3-cliques $\{i,\ j,\ k\}$ of the constraint graph there exists a value $c \in D_k$ compatible with both $(i,\ a)$ and $(j,\ b)$ such that $(c \geq M_{ika}) \wedge (c \geq M_{jkb}) \wedge (a \geq M_{kic}) \wedge (b \geq M_{kjc})$.

4.2 The Algorithm

The data structures of Max-RPCEn1 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:
  - $next(D_i, nil)$ returns the smallest value of D_i if $D_i \neq \emptyset$ and ∞ otherwise. $next(D_i, a)$ with $a \neq nil$ returns the smallest value in D_i greater than a if a is not the greatest value of D_i, and ∞ otherwise.
  - $remove(D_i,\ a)$ removes the value a from D_i.
- A value a is in S_{jib} if $(i,\ a)$ is currently supported by $(j,\ b)$ i.e. b is the current valid path consistent support of $(i,\ a)$ on C_{ij}. In other words, S_{jib} is the set of the values that may no longer have a valid path consistent support in D_j if we delete $(j,\ b)$. S_{jb} is the set of the values $(i,\ a)$ such that $a \in S_{jib}$ for some $C_{ij} \in \mathcal{C}$. If $S_{jib} \neq \emptyset$, $first(S_{jib})$ returns the first value in S_{jib} and ∞ otherwise.
- To take advantage of the bidirectionality of the constraints and to detect the conservative path inconsistency of some pairs of values, Max-RPCEn1 uses the array M. $M_{ija} = b$ if $\forall b' \in D_j$ s.t. $b' < b$, $(j,\ b')$ is not a valid path consistent support for $(i,\ a)$, namely $(j,\ b')$ is not compatible with $(i,\ a)$, $((i,\ a),\ (j,\ b'))$ is path inconsistent, or Max-

```
function HasAValidPCSupport(i, a, j) : boolean;
while S_ija ≠ ∅ do
    b ← first(S_ija)
    if b ∉ D_j then remove b from S_ija
    else S_jib ← S_jib ∪ {a}; return true;
M_ija ← next(D_j, M_ija);
while M_ija ≠ ∞ do
    if (a > M_jiM_ija) and (C_ij(a, M_ija)) then
        ValidPC ← true;
        forall k ∈ Common[C_ij] while ValidPC do
            c ← nil;
            if IsValidPathConsis−
            tent(i, a, j, M_ija, k, c) then CS[k] ← c;
            else ValidPC ← false;
        if ValidPC then
            S_jiM_ija ← S_jiM_ija ∪ {a};
            forall k ∈ Common[C_ij] do
                S_kCS[k]^PC ← S_kCS[k]^PC ∪ {((i, a), (j, M_ija))}
            return true;
    M_ija ← next(D_j, M_ija);
return false;

function IsValidPathConsistent(i, a, j, b, k, in out c) :
boolean;
if (M_ika = M_jkb) then
    if M_ika ∈ D_k then
        c ← M_ika; return true;
    else c ← max(c, M_ika);
else c ← max(c, M_ika, M_jkb);
if c ∉ D_k then c ← next(D_k, c);
while c ≠ ∞ do
    if (a ≥ M_kic) ∧ (b ≥ M_kjc) ∧ (C_ik(a, c))
        ∧(C_jk(b, c)) then return true;
    c ← next(D_k, c);
return false;
```

Figure 3. The functions of Max-RPCEn1.

RPCEn1 has found the conservative path inconsistency of $((i,\ a),\ (j,\ b'))$.

- If $((i,\ a),\ (j,\ b)) \in S^{PC}_{kc}$ and $(a \in D_i \wedge b \in D_j)$, b is the current valid path consistent support of $(i,\ a)$ in D_j and $(k,\ c)$ is currently supporting $((i,\ a),\ (j,\ b))$, i.e. $(k,\ c)$ is such that $(C_{ik}(a,\ c) \wedge C_{jk}(b,\ c))$ and Max-RPCEn1 has not found that $((i,\ a),\ (k,\ c))$ or $((j,\ b),\ (k,\ c))$ are conservative path inconsistent. So, S^{PC}_{kc} is the set of the pairs of values that may be no longer valid path consistent w.r.t. k if we delete $(k,\ c)$.
- $Common[C_{ij}]$ is the set of the variables k that are linked to both i and j, i.e. the variables k such that $\{i,\ j,\ k\}$ is a 3-clique in the constraint graph.
- An arc-value pair $[(i,\ j),\ a]$ is in $InitList$ if Max-RPCEN1 has not yet checked whether $(i,\ a)$ has a valid path consistent support 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 yet.

For each arc-value pair $[(i,\ j),\ a]$ Max-RPCEn1 (see Fig. 2) uses the function $HasAValid$-$PCSupport$ (Fig. 3) to know whether $(i,\ a)$ has a valid path consistent support on C_{ij}. This function tries first (lines 1 to 4) to infer a valid path consistent support looking for an undeleted value in S_{ija}, i.e. in the list of the values supported by $(i,\ a)$ on C_{ij}. If no valid path consistent support can be inferred, Max-RPCEn1 goes on with its search looking for the smallest valid path consistent support in D_j. The array M allows to reduce the number of constraint checks performed. $HasAValidPCSupport$ does not check $(j,\ M_{ija})$ (see line 5) because it is not a valid path consistent support for $(i,\ a)$. Indeed, if it is the first time that $HasAValidPCSupport$ is called for the pair arc value $[(i,\ j),\ a]$, $M_{ija} = nil$, otherwise $HasAVAlidPCSupport$ has been called by $PropagDeletion$ (Fig. 2) because the current valid path consistent support of $(i,\ a)$ on C_{ij} has been deleted and $(j,\ M_{ija})$ is no longer in D_j. Furthermore, by definition of M_{ija}, there is no valid path consistent support of $(i,\ a)$ in D_j lower than M_{ija}. So, if $(i,\ a)$ has a valid path consistent support in D_j, it is greater than M_{ija}. Furthermore, if $b \in D_j$ is a valid path consistent support for $(i,\ a)$, $a > M_{jib}$ since if $a < M_{jib}$ $(i,\ a)$ is not a valid path consistent support for $(j,\ b)$ and if $a = M_{jib}$, b would have been found in S_{ija}. $IsValidPathConsistent$ (Fig. 3) is used to know whether a pair of values $((i,\ a),\ (j,\ b))$ is valid path consistent w.r.t. a third variable k. This function looks for the smallest value c in D_k supporting $((i,\ a),\ (j,\ b))$, namely such that $((i,\ a),\ (k,\ c))$ and $((j,\ b),\ (k,\ c))$ are allowed pairs of values not detected conservative path inconsistent by Max-RPCEn1. If we say that a value $c \in D_k$ *supports* the pair of values $((i,\ a),\ (j,\ b))$ if it is compatible with both $(i,\ a)$ and $(j,\ b)$ and if it is such that $(c \geq M_{ika}) \wedge (c \geq M_{jkb}) \wedge (a \geq M_{kic}) \wedge (b \geq M_{kjc})$, then $IsValidPathConsistent(i,\ a,\ j,\ b,\ k,\ c)$ looks for the smallest value $c \in D_k$ supporting $((i,\ a),\ (j,\ b))$. If at the call of this function the parameter c is not nil, c is the value of D_k that was supporting $((i,\ a),\ (j,\ b))$, and so, no value lower than c in D_k supports $((i,\ a),\ (j,\ b))$. This allows to never check a value of D_k twice to know whether $((i,\ a),\ (j,\ b))$ is valid path consistent w.r.t. k. If $M_{ika} = M_{jkb}$ and $M_{ika} \in D_k$, $IsValidPathConsistent$ infers that M_{ika} supports $((i,\ a),\ (j,\ b))$. We could infer a support of $((i,\ a),\ (j,\ b))$ in D_k looking for a value in $(S_{ika} \cap S_{jkb} \cap D_k)$ but this is not cost effective.

If for all the 3-clique $\{i,\ j,\ k\}$ a value $c \in D_k$ supporting $((i,\ a),\ (j,\ b))$ has been found, a is added in S_{jib} to store that b is the current valid path consistent support of $(i,\ a)$ in D_j, and $((i,\ a),\ (j,\ b))$ is added in S^{PC}_{kc} to store that c is currently supporting $((i,\ a),\ (j,\ b))$. Obviously, if $(i,\ a)$ has no valid path consistent support in D_j, $(i,\ a)$ is deleted and Max-RPCEn1 adds this value in $DeletionList$ to propag its deletion. $PropagDeletion$ propagates the deletion of the values of $DeletionList$. For each value $(j,\ b)$ in $DeletionList$, we have to delete all the values supported by $(j,\ b)$ (the values of S_{jb}) that no longer have any valid path consistent support in D_j. Furthermore, for all the pairs of values $((i,\ a),\ (k,\ c))$ in S^{PC}_{jb} (the pairs of values that were supported by $(j,\ b)$), $PropagDeletion$ has to check whether $((i,\ a),\ (k,\ c))$ is still valid path consistent w.r.t. j. If no supporting value can be found in D_j, $(k,\ c)$ is no longer a valid path consistent support for $(i,\ a)$ and $PropagDeletion$ has to look for another valid path consistent support for $(i,\ a)$ in D_k to know whether $(i,\ a)$ has to be deleted.

4.3 Complexity

To check whether a value a is in S_{kic} (line 8 of $PropagDeletion$) and to remove this value from S_{kic} (line 13) in constant time, we use in our implementation an array called $SupportedBy$ such that $SupportedBy_{ika}$ is the current valid path consistent support of $(i,\ a)$ in D_k, and an array $PtSupportedBy$ s.t. $PtSupportedBy_{ika}$ points at the value a in S_{kic}, where c is the current valid path consistent support of $(i,\ a)$ in D_k. The space required by these data structures is in $O(ed)$ and so, they do not change the worst case space complexity of Max-RPCEn1 (see below).

The cost of the initialization of the data structures S, S^{PC}, M, and $InitList$ (lines 1 to 6 of Max-RPCEn1) is in $O(ed)$. The time required to determine the 3-cliques of the constraint graph (lines 7 to 12) is in $O(en)$. Since $HasAValidPC$-$Support$ removes from S_{ija} the values that are no longer in D_j, the test of line 3 is performed at most $O(d)$ times for each arc-value pair. Furthermore, the value of M_{ija} (which never decreases) is increased at each step of the second loop (lines 6 to 19) of $HasAValidPCSupport$ and if M_{ija} has reached the last value of D_j, M_{ija} is set to ∞ at line 19 to stop the loop. Therefore, for each arc-value pair $[(i,\ j),\ a]$, and each value $b \in D_j$, $HasAValidPCSupport$ checks at most once whether $(j,\ b)$ is a valid path consistent support for $(i,\ a)$. So, Max-RPCEn1 checks the valid path consistency of at most $O(ed^2)$ pairs of values. To know whether a pair of value $((i,\ a),\ (j,\ b))$ is valid path consistent, for each 3-clique $\{i,\ j,\ k\}$ in the constraint graph, Max-RPCEn1 calls $IsValidPathConsistent$ to look for the smallest value $c \in D_k$ supporting $((i,\ a),\ (j,\ b))$. When $IsValidPath$-$Consistent$ checks the valid path consistency of $((i,\ a),\ (j,\ b))$ w.r.t. k, it checks only the values of D_k greater than the previous support of $((i,\ a),\ (j,\ b))$ in D_k. So, for each support $(j,\ b)$ of a value $(i,\ a)$, and each 3-clique $\{i,\ j,\ k\}$, a value of D_k is checked at most once to know whether $(j,\ b)$ is a valid path consistent support for $(i,\ a)$ w.r.t. k. Therefore, the complexity due to the calls to $IsValidPathConsistent$ is $O(cd^3)$ where c is the number of 3-cliques in the constraint graph, and the worst case time complexity of Max-RPCEn1 is $O(en + ed^2 + cd^3)$.

The size of $InitList$ is $O(ed)$ since each arc-value pair is put in this list once. When a value is deleted, it is put in $DeletionList$ and it will not be put in this list any more. So, the worst case space complexity of $DeletionList$ is $O(nd)$. A value a is in S_{jib} if $(j,\ b)$ is the current valid path consistent support of $(i,\ a)$. Since Max-RPCEn1 looks for only one valid path consistent support for each value on each constraint, the size of the data structure S is in $O(ed)$. The data structure M is an array of $O(ed)$ counters. If a pair $((i,\ a),\ (j,\ b))$ is in S^{PC}_{kc}, b is the current valid path consistent support of $(i,\ a)$ in D_j, $\{i,\ j,\ k\}$ is a 3-clique, and $(k,\ c)$ is currently supporting $((i,\ a),\ (j,\ b))$. The worst case space complexity of the data structure S^{PC} is $O(cd)$ since for each value $(i,\ a) \in D$ and each 3-clique $\{i,\ j,\ k\}$ there is only one value $c \in D_k$ such that a pair $((i,\ a),\ (j,\ *))$ is in S^{PC}_{kc} and there is only one such a pair of values in S^{PC}_{kc}. Therefore, the worst case space complexity of Max-RPCEn1 is $O(ed + cd)$.

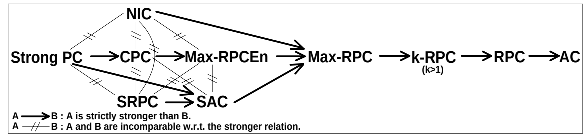

Figure 4. Relations between the mentioned local consistencies.

Consequently, Max-RPCEn1 has the same worst case time and space complexities as Max-RPC1.

5 Pruning efficiency

We call Max-restricted path consistency enhanced (Max-RPCEn) the local consistency achieved by Max-RPCEn1. Like directional arc consistency [10], which depends on the variable ordering used, Max-RPCEn depends on the orderings used to handle the domains, $InitList$, and $DeletionList$. In the following experiments, the arc-value pairs of $InitList$ are checked in the lexicographic order and $DeletionList$ is a "Last In, First Out" list. Fig. 4 summarizes the relations between Max-RPCEn, CPC, AC, RPC, k-RPC, Max-RPC, path inverse consistency (PIC, [13]), neighborhood inverse consistency (NIC, [13]), singleton arc consistency (SAC, [6]), singleton restricted path consistency (SRPC, [6]), and strong path consistency (strong PC [5, 19]). There is an arrow from LC to LC' iff LC is strictly stronger [6] than LC', namely if on any CN on which LC holds, LC' holds too and there is at least one CN in which LC' holds and LC does not. A crossed line between two local consistencies means that they are not comparable w.r.t. the stronger relation. A proof of these relations can be found in [8].

The stronger relation does not induce a total ordering. Especially, CPC and Max-RPCEn are not comparable to SAC, NIC and SRPC w.r.t. the stronger relation. Furthermore, Fig. 4 does not show if a local consistency is far more pruningful than another or if it performs only few additional value deletions.

Therefore an experimental evaluation has been done. We used the random uniform CN generator of [14]. It involves four parameters: n the number of variables, d the common size of the initial domains, $p1$ the proportion of constraints in the network

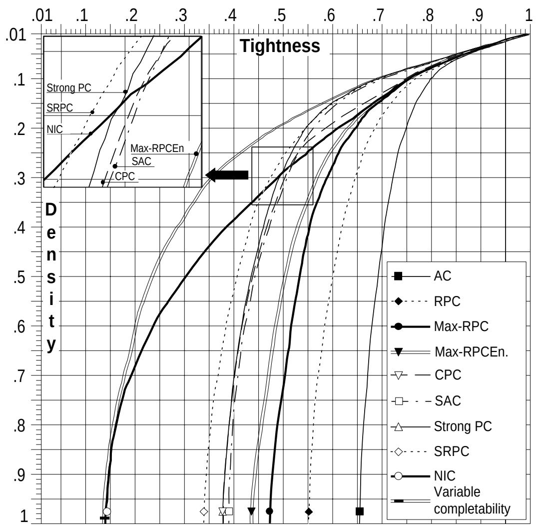

Figure 5. The T_{all} bounds for random CNs with n=40 and d=15.

(the density $p1$=1 corresponds to the complete graph) and $p2$ the proportion of forbidden pairs of values in a constraint (the tightness). The generated problems have 40 variables and 15 values in each domain. For each local consistency and each density $p1$, we have determined $T_{all}(p1)$, namely the tightness such that the local consistency finds the inconsistency of 50% of the CNs generated with tightness $T_{all}(p1)$ and density $p1$. For all the mentioned local consistencies, the value $T_{all}(p1)$ for any density $p1$ is given in Fig. 5. We also show these bounds for the variable completability filtering [12], which removes the values that do not belong to any solution, and thus is the strongest filtering we can have when we limit filtering to the domains. Many instances have to be considered to determine the T_{all} bound. This explains that the generated problems are relatively small.

On sparse random uniform CNs, conservative path consistency is less prunigful than singleton arc consistency, but on more dense CNs, CPC has a better pruning efficiency than SAC. These two local consistencies removes almost all the strong path inconsistent values. Obviously, on complete CNs, CPC is strong path consistency. However, even on relatively sparse CNs, the T_{all} bounds of CPC and strong PC are very close. Compared to Max-RPC, Max-RPCEn is a substantial enhancement w.r.t. the pruning efficiency.

6 Time efficiency

The same random uniform CN generator is used to compare the time efficiency. Fig. 6 shows the results on relatively sparse CNs ($p1$=5%) having 1000 variables and 20 values in each initial domain, and Fig. 7 presents performances on complete CNs with n=100 and d=30. For each tightness, 50 instances were generated, and Fig. 6 and Fig. 7 show mean values obtained on a Pentium II-266 Mhz with 64 Mo of memory under Linux.

The advantage of using Max-RPCEn1 on sparse CNs is obvious. The bidirectionality allows substantial constraint check savings and although Max-RPCEn1 deletes more values than Max-RPC1, it requires less cpu time. There is "few" 3-cliques in the constraint graph of these sparse CNs, and Max-RPC1 and Max-RPCEn1 requires at most 108 seconds (On all the CNs generated with a tightness lower than 0.69, PC8, the PC algorithm presented in [5], requires more than 68 hours).

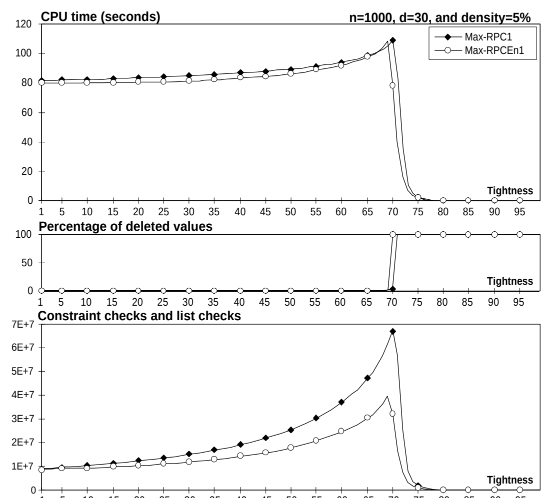

Figure 6. Experimental evaluation on CNs with n=1000, d=30, and density=0.05.

On dense CNs, Max-RPCEn1 still requires less constraint checks and list checks than Max-RPC1. However, there is many 3-cliques in the constraint graph, and Max-RPCEn1 detects the conservative path inconsistency of many pairs of values. Therefore, for each value $(i,\ a)$ and each constraint C_{ij}, Max-RPCEn1 checks more values of D_j to find a valid path consistent support for $(i,\ a)$ than Max-RPC1 to find a path consistent support. So, Max-RPCEn1 requires more cpu time than Max-RPC1 as long as it removes only few values. However, the cpu time performances of Max-RPCEn1 and Max-RPC1 remain of the same order of magnitude, and the improvement of the pruning efficiency is significant on these complete CNs.

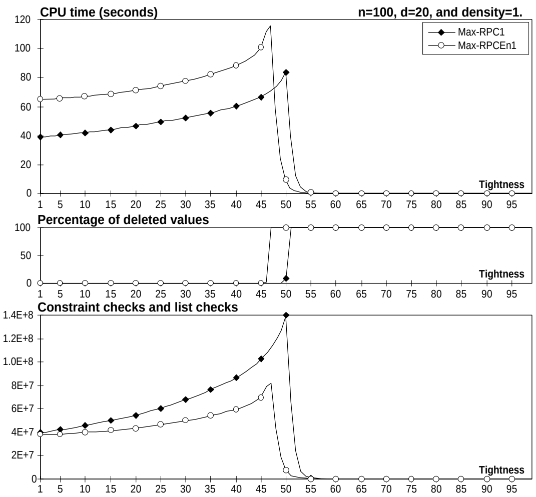

Figure 7. Experimental evaluation on complete CNs with n=100, and d=20.

7 Conclusion

In this paper, we proposed a new local consistency called Max-restricted path consistency enhanced that removes far more values than Max-RPC. An algorithm, called Max-RPCEn1 is also proposed to achieve it. This algorithm has cpu time performances that remain comparable to those of Max-RPC1, and it requires less constraint checks. Therefore, while considering both time and pruning efficiences, Max-restricted path consistency enhanced is one of the most insteresting local consistencies.

References

[1] Berlandier, P.: Improving Domain Filtering using Restricted Path Consistency. In proceedings of IEEE CAIA-95, Los Angeles CA (1995)

[2] Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence **65** (1984) 179–190

[3] Bessière, C., Freuder, E.C., Régin, J.C.: Using inference to reduce arc-consistency computation. In proceedings of IJCAI-95, Montréal, Canada (1995) 592-598

[4] Bessière, C., Régin, J.C.: MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. in proceedings of CP-96, Cambridge, MA (1996) 61–75

[5] Chmeiss, A., and Jégou, P.: Two New Constraint Propagation Algorithms Requiring Small Space Complexity, in proceedings of ICTAI-96, Toulouse, France (1996) 286–289

[6] Debruyne, R., Bessière, C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In proceedings of IJCAI-97, Nagoya, Japan (1997) 412–417

[7] Debruyne, R., Bessière, C.: From Restricted Path Consistency to Max-Restricted Path Consistency. In proceedings of CP-97, Linz, Austria (1997) 312–326

[8] Debruyne, R.: Removing more values than Max-restricted path consistency for the same cost. Technical Report 98041, Montpellier, France (1998)

[9] Debruyne, R., Bessière, C.: Which local consistency has to be used on large constraint networks?. Technical Report 98037, Montpellier, France (1998)

[10] Dechter, R., Pearl, J.: Network-Based Heuristics for Constraint-Satisfaction Problems. Artif. Intell. **34(1)** (1988) 1-38

[11] Freuder, E.: A sufficient condition for backtrack-bounded search. Journal of the ACM **32(4)** (1985) 755–761

[12] Freuder, E.: Completable Representations of Constraint Satisfaction Problems. in proceedings of KR-91, Cambridge, MA (1991) 186–195

[13] Freuder, E., Elfe, D.C.: Neighborood Inverse Consistency Preprocessing. In proceedings of AAAI-96, Portland OR (1996) 202–208

[14] Frost, D., Bessière, C., Dechter, R., and Régin, J.C.: Random Uniform CSP Generators. http://www.ics.uci.edu/~dfrost/csp/generator.html (1996)

[15] Golomb, S.W., Baumert, I.D.: Backtrack programming. Journal of the ACM **12(4)** (1965) 516–524

[16] Grant, S.A., Smith, B.M.: The phase transition behaviour of maintaining arc consistency, in proceedings of ECAI-96, Budapest, Hungary (1996) 175–179

[17] Haralick, R.M., Elliot, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. **14** (1980) 263–313

[18] Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In proceedings PPCP-94, Seattle, WA (1994)

[19] Singh, M.: Path Consistency Revisited. In proceedings of IEEE ICTAI-95, Washington D.C. (1995)