A Property of Path Inverse Consistency Leading to an Optimal PIC Algorithm

Romuald Debruyne[1]

Abstract. In constraint networks, the efficiency of a search algorithm is strongly related to the local consistency maintained during search. For a long time, it has been considered that forward checking was the best compromise between the pruning effect and the amount of overhead involved. But recent works, comparing the search algorithms on a large range of networks, show that maintaining arc consistency during search (MAC) outperforms forward checking on large and hard problems. It is conceivable that on very difficult instances, using an even more pruningful local consistency may pay off. To know which local consistency is the most promising, a study comparing both their pruning efficiency and the time needed to achieve them has been done [4]. This work shows that PIC1, the path inverse consistency algorithm presented in [6], has very bad average and worst case time complexities. In this paper, we give a property of PIC and we propose and evaluate a PIC algorithm based on this property that has an optimal worst case time complexity. Experiments show that maintaining PIC during search outperforms MAC on hard sparses CNs.

1 Introduction

Finding a solution in a constraint network (CN), i.e. looking for an assignment of values for the problem variables that satisfies all the constraints of the network, is NP-hard. A blind search often leads to a combinatorial explosion, the algorithm thrashing because of some local inconsistencies. Therefore, filtering techniques are essential to remove once and for all some local inconsistencies during a preprocessing step or to efficiently prune the search tree during search. To solve small and "easy" problems, it is sufficient to maintain a low level of local consistency, such as forward checking. The additional cost of maintaining a more pruningful local consistency cannot be outweighed. But on large and hard problems, a more pruningful filtering technique is essential to avoid combinatorial explosion. A good illustration is that MAC outperforms forward checking on hard problems. The question is, "Is there a local consistency that outperforms arc consistency (AC) on very hard CNs?". To give a first answer, some comparisons between the local consistencies more pruningful than AC have been carried out [3][4] considering both pruning and time efficiencies. From that work, retricted path consistency (RPC, [1]) and Max-RPC seem be victorious. Path inverse consistency was indeed more expensive than Max-RPC while not pruning far more than RPC which is weaker and cheaper. PIC1, the PIC algorithm used for the comparison is in $O(en^2d^4)$ when RPC2 and Max-RPC1 (the Max-RPC algorithm proposed in [3]) are in $O(ed^2+cd^2)$[2] and $O(ed^2+cd^3)$ respectively. But PIC is a recent filtering technique and has not yet been much studied. PIC1 is the first and only one algorithm to achieve it. PIC1 has bad time performances because it does not store anything else than the deleted values. Therefore, it has a linear worst case space complexity that would allow using it on very large CNs but it is too time expensive for this use. In this paper, we propose a new PIC algorithm, called PIC2, that has an optimal $O(ed^2+cd^3)$ worst case time complexity. Furthermore, PIC2 has an $O(cd)$ worst case space complexity, but like RPC2 [3] and Max-RPC1, its average space complexity is far from this limit. Experiments show that PIC2 is far more efficient than PIC1 and that maintaining PIC during search can lead to better time performances than MAC on hard sparse CNs.

[1] Ecole des Mines de Nantes, 4, rue Alfred Kastler, La Chantrerie, 44307 Nantes cedex 03 - France, email: romuald.debruyne@emn.fr

[2] c is the number of 3-cliques in the network

2 Definitions and notations

A *network of binary constraints* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined by a set $\mathcal{X} = \{i, j, \ldots\}$ of *n variables*, each taking value in its respective finite *domain* $D_i, D_j, \ldots$ 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}. With each CN we associate a *constraint graph* in which nodes represent variables and arcs connect pairs of variables which are constrained explicitly. c is the number of 3-cliques in the constraint graph. An *instantiation* of a set of variables S is an indexed set of values $\{I_j\}_{j \in S}$ s.t. $\forall j \in S \;\; I_j \in D_j$. 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 CN is *consistent* if it has at least one solution.

3 Path Inverse Consistency

Before the presentation of PIC, let us recall what RPC and Max-RPC are because there are some relations between PIC and these two local consistencies.

The aim of Berlandier when he proposed RPC was to remove more inconsistent values than AC while avoiding the drawbacks of path consistency (PC). Indeed, a PC algorithm has to try to extend all the pairs of values, even those between two independent variables, to any third variable. This is very expensive and the best PC algorithm [9] has an $O(n^3d^3)$ worst case time complexity with an $O(n^3d^2)$

worst case space complexity. In addition, removing a pair of values can lead to the addition of a constraint in the network. To avoid these drawbacks, a RPC algorithm only removes values, leaving unchanged the set of constraints. Furthermore, to have a better worst case time complexity, RPC checks the path consistency of a pair of values only if it can directly lead to the deletion of a value. 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 of $(i,\ a)$ in D_j. If such a pair is path inconsistent, its deletion would lead to the arc inconsistency of $(i,\ a)$. So these few additional path consistency checks allow detecting more inconsistent values than AC without having to delete any pair of values. Furthermore, the number of pairs that have to be checked is greatly reduced compared to PC, and the worst case time complexity of RPC2 is $O(ed^2 + cd^2)$.

Max-RPC is an extension of RPC. RPC checks the path consistency of a support $(j,\ b)$ only when it is the only support of a value $(i,\ a)$. We can remove more values by checking the existence of a path consistent support for all the values on each constraint, whatever is the number of supports the value has. This is the base of Max-RPC1 [3] which has an $O(ed^2 + cd^3)$ worst case time complexity.

When they proposed inverse consistency in [6], the aim of Freuder and Elfe was to achieve high order local consistencies with a good space complexity. For a long time, the only studied local consistencies were k-consistencies (i.e., (k-1, 1)-consistencies in the formalism of [5]) which remove the consistent instantiations of length k-1 that cannot be extended to a consistent instantiation including any additional k^{th} variable. k-consistency has a time complexity polynomial with the exponent dependent on k and requires $O(n^{k-1}d^{k-1})$ space to store the deleted instantiations. So, only 2-consistency (i.e., arc consistency) can be used in practice. Path consistency (3-consistency) and higher levels of k-consistency are too expensive, and change the structure of the network. Space requirements are no longer a problem with k inverse consistency ((1, k-1)-consistency), which removes the values that cannot be extended to a consistent instantiation including any k-1 additional variables. It has a linear space complexity. But the time complexity remains polynomial with the exponent dependent on k. So, the inverse consistencies that can be used in practice are path inverse consistency (k=3) and neighborhood inverse consistency (which removes the values that cannot be extended to a consistent instantiation including all the variables linked to it) on sparse CNs. By definition, a PIC algorithm has to remove the values $(i,\ a)$ that cannot be extended to a consistent instantiation including any 3-tuple of variables including i. In [6], the authors remark that not all 3-tuples need to be checked. Only one of the tuples $(i,\ j,\ k)$ and $(i,\ k,\ j)$ has to be checked and we do not have to check the 3-tuples $(i,\ j,\ k)$ such that i is linked to neither j nor k. Indeed, in this last case, $(i,\ a)$ can be removed only if PIC has deleted all the values of j or k and so PIC would have already detected the inconsistency of the network.

On some of the 3-tuples we still have to check, PIC is nothing more than AC. If the CN is arc consistent, any value $(i,\ a)$ can be extended to a 3-tuple $(i,\ j,\ k)$ such that there is no constraint between j and k. If AC holds, there is $b \in D_j$ and $c \in D_k$ s.t. $C_{ij}(a,\ b)$ and $C_{ik}(a,\ c)$, and since j is not linked to k, $((i,\ a),\ (j,\ b),\ (k,\ c))$ is consistent. Furthermore, AC is a sufficient condition to prove that a value $(i,\ a)$ can be extended to $(i,\ j,\ k)$ if there is no constraint between i and k (resp. between i and j). If AC holds, $(i,\ a)$ has a support b in D_j (resp. c in D_k) and this value being arc consistent too, it has a support c in D_k (resp. b in D_j). So $((i,\ a),\ (j,\ b),\ (k,\ c))$ is consistent. Consequently, if the constraint network is arc consistent, the only 3-tuples that have to be checked to achieve PIC correspond to the 3-cliques of the constraint graph. Furthermore, the definition of PIC shows that any constraint network involving less than three variables is path inverse consistent, even though it is not arc consistent. These remarks lead to the following property.

- 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 **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 binary CN is **path inverse consistent** iff it is $(1, 2)$-consistent i.e. $\forall (i,\ a) \in D$ $\forall j, k \in \mathcal{X}$ s.t. $j \neq i \neq k \neq j$, $\exists (j,\ b) \in D$ and $(k,\ c) \in D$ s.t. $C_{ij}(a,\ b) \wedge C_{ik}(a,\ c) \wedge C_{jk}(b,\ c)$

Figure 1. The mentionned local consistencies

Property 1. A CN is path inverse consistent iff

- it involves less than three variables, or
- it is arc consistent and for each value (i, a) in $\mathcal{D}$, for any 3-clique $\{i, j, k\}$, $(i,\ a)$ can be extended to a consistent instantiation of $\{i, j, k\}$.

This property highlights the relations between PIC, RPC and Max-RPC. Let us consider a value $(i,\ a)$, a constraint C_{ij} and the set S of the variables s s.t. $\{i, j, s\}$ is a 3-clique. Obviously, if $(i,\ a)$ is not arc consistent, the three local consistencies delete it. If $(i,\ a)$ has only one support b in D_j, PIC, RPC and Max-RPC have the same behavior. They delete $(i,\ a)$ because of C_{ij} if $((i,\ a),\ (j,\ b))$ is path inconsistent. If $(i,\ a)$ has at least two supports in D_j, $(i,\ a)$ is restricted path consistent w.r.t. C_{ij} but PIC can delete it if there is $s \in S$ such that all the supports of $(i,\ a)$ in D_j are path inconsistent because of s. This explains that the pruning efficiency of PIC is close to the one of RPC because it is infrequent that all the supports of a value are path inconsistent because of the same third variable. A Max-RPC algorithm removes much more values since it deletes a value $(i,\ a)$ because of C_{ij} if it has no path consistent support in D_j. In other words, $(i,\ a)$ is deleted if all its supports in D_j are path inconsistent, even if they are not path inconsistent because of the same third variable.

4 PIC2

4.1 Bases of the algorithm

PIC2 uses Property 1. To remove the arc inconsistent values, PIC2 uses AC7 [2], the most efficient AC algorithm. As soon as AC holds, achieving PIC is only checking whether the values can be extended to any 3-clique involving their variable. This greatly reduces the set of 3-tuples that have to be checked, especially on sparse CNs.

This enhancement is not sufficient. The reason why PIC1 has an $O(en^2d^4)$ worst case time complexity is that it stores nothing else than the deleted values. If PIC1 has found that (i, a) can be extended to $((i, a), (j, b), (k, c))$ it does not store this information. Therefore, if a value (j, b) is deleted, PIC1 does not know which values may be no longer path inverse consistent because of this deletion. It can only overestimate this set by the values of all the variables linked to j. To avoid this drawback, when PIC2 finds that (i, a) can be extended to $((i, a), (j, b), (k, c))$ it stores that the path inverse consistency of (i, a) depends on (j, b) and (k, c). While $b \in D_j$ and $c \in D_k$, PIC holds for (i, a) w.r.t. $\{i, j, k\}$. PIC2 will try to extend (i, a) to $\{i, j, k\}$ again only if (j, b) or (k, c) is deleted. Storing this information has another advantage. When PIC2 looks for a pair $(b, c) \in D_j \times D_k$ consistent with (i, a), it uses a lexicographic order. So, if (i, a) is supported by a pair $((j, b), (k, c))$ we are sure that this is the first pair compatible with (i, a) according to this order. If (j, b) or (k, c) is deleted, we have to look for another pair of values in $D_j \times D_k$ supporting (i, a) but it is useless to check the pairs $((j, b'), (k, c'))$ such that $b' < b$ or $((b' = b)$ and $(c' \leq c))$. Therefore, in the worst case PIC2 checks each pair of $D_j \times D_k$ only once to check the path inverse consistency of a value (i, a) w.r.t. $\{i, j, k\}$.

4.2 The Algorithm

The data structures of PIC2 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:
  - $first(D_i)$ returns the smallest value of D_i if $D_i \neq \emptyset$ and ∞ otherwise.
  - $last(D_i)$ returns the greatest value of D_i if $D_i \neq \emptyset$ and nil otherwise.
  - if $a \in D_i \backslash last(D_i)$, $next(D_i, a)$ returns the smallest value in D_i greater than a. $next(D_i, nil)$ returns the smallest value of D_i if $D_i \neq \emptyset$ and ∞ otherwise.
  - $remove(D_i, a)$ removes the value a from D_i.
- S^{AC} is the array of lists of supported values of AC7. A value a is in $S^{AC}_{jb}[i]$ if (j, b) is currently supporting (i, a). If a value (j, b) is deleted, PIC2 has to check whether the values (i, a) in S^{AC}_{jb} are still arc consistent w.r.t. C_{ij}.
- Like in AC7, L_{ija} is the last value of D_j checked to find a support for (i, a) i.e. $\forall b \in D_j$ s.t. $b \leq L_{ija}$, $C_{ij}(a, b)$ has already been checked.
- The list S^{PIC}_{jb} is used to know the values for which the path inverse consistency depends on (j, b). $((i, a), (k, c)) \in S^{PIC}_{jb}$ if $((j, b), (k, c))$ is the current "pic support" of (i, a). In other words, if $(j < k)$ (resp. $k < j$) and (i, a), (j, b), and (k, c) are in $\mathcal{D}$, (b, c) (resp. (c, b)) is the first allowed pair of $D_j \times D_k$ (resp. $D_k \times D_j$) w.r.t. the lexicographic order that is consistent with (i, a). As long as $b \in D_j$ and $c \in D_k$, (i, a) is path inverse consistent w.r.t. $\{i, j, k\}$. If (j, b) (or (k, c)) is deleted, another pair of $D_j \times D_k$ supporting (i, a) has to be found but since (b, c) was the first in $D_j \times D_k$, only the pairs greater than (b, c) have to be checked. If a pair $((i, a), (k, c)) \in S^{PIC}_{jb}$, $((i, a), (j, b))$ is in S^{PIC}_{kc} since the path inverse consistency of (i, a) depends on (k, c) too. These two elements must be linked to allow performing the line 9 of $PropagDeletion$ in constant time.
- An arc-value pair $[(i, j), a]$ is in $InitACList$ if PIC2 has not checked the arc consistency of (i, a) w.r.t. C_{ij}. $[(i, a), j, k]$ is in $InitPICList$ if PIC2 has not checked the path inverse consistency of (i, a) w.r.t. the clique $\{i, j, k\}$. 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 $[(i, j), a]$ PIC2 uses the function $SeekACSupport$ to know whether (i, a) is arc consistent w.r.t. C_{ij}. This function has the behavior of AC7. First, it tries to infer a support looking for an undeleted value in $S^{AC}_{ia}[j]$, i.e. the list of the values supported by (i, a) on C_{ij}. If no support can be inferred, PIC2 goes on with its search looking for the smallest support in D_j. The array L is used to never perform a constraint check twice during AC achievement. Only the values greater than L_{ija} have to be checked. Furthermore, it is useless to check $C_{ij}(a, b)$ if $L_{jib} \geq a$ since in such a case PIC2 has already checked whether (i, a) is a support of (j, b), and if it is a compatible value $SeekACSupport$ would have found b in $S^{AC}_{ia}[j]$.

In addition to AC, PIC2 tries to extend each value (i, a) to any 3-clique $\{i, j, k\}$ using $SeekPIC\text{-}Support(i, a, j, b, k, c)$. This function looks for the first pair in $D_j \times D_k$ according to the lexicographic order that is consistent with (i, a) and greater than (b, c). If this pair of values exists, $SeekPICSupport$ uses S^{PIC} to store (line 22) that the path inverse consistency of (i, a) depends on this pair.

If a value (j, b) is deleted, $PropagDeletion$ checks whether the values in S^{AC}_{jb} (the values currently supported by (j, b)) still have a support in D_j. Furthermore, for each pair of values $((i, a), (k, c)) \in S^{PIC}_{jb}$, $PropagDeletion$ uses $SeekPICSupport$ to know whether (i, a) is still path inverse consistent w.r.t. $\{i, j, k\}$.

5 Complexity

A PIC algorithm has to remove all the arc inconsistent values and since the optimal worst case time complexity of achieving arc consistency is $O(ed^2)$, $O(ed^2)$ is a lower bound time complexity for PIC. Furthermore, for each value (i, a) and each 3-clique $\{i, j, k\}$, a PIC algorithm has to check whether (i, a) can be extended to a consistent instantiation including j and k. In the worst case, all the pairs of values of $D_j \times D_k$ have to be checked to know whether (i, a) is path inverse consistent w.r.t. $\{i, j, k\}$. So, $O(ed^2 + cd^3)$ is a lower bound for the worst case time complexity of any PIC algorithm.

Since $SeekACSupport$ removes from $S^{AC}_{ia}[j]$ 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, L_{ija} is bounded above by d and L_{ija} increases at each step of the second loop of $SeekACSupport$. Thus, the cost of this loop is $O(d)$ for each arc-value pair and the complexity due to the calls to $SeekACSupport$ is $O(ed^2)$. If PIC2 has to check the path inverse consistency of (i, a) w.r.t. $\{i, j, k\}$ (with $j < k$), it checks only the pairs of values of $D_j \times D_k$ it has not already checked, i.e. those greater (w.r.t. the lexicographic order) than the pair currently supporting (i, a). So, a pair of $D_j \times D_k$ is checked at most once to check the path inverse consistency of a value (i, a) w.r.t. $\{i, j, k\}$. Thus, in the worst case $SeekPICSupport$ checks $O(cd^3)$ times whether a pair of values is consistent with a third value and PIC2 has an optimal $O(ed^2 + cd^3)$ worst case time complexity. Since there is at most $O(en)$ 3-cliques in the network, $O(end^3)$ is an upper bound time complexity for PIC2. However this is a rough upper bound since there can be much less than $O(en)$ 3-cliques, especially on sparse CNs.

Only the deleted values for which the deletion has not yet been propagated are in $DeletionList$. So, there is at most $O(nd)$ values in $DeletionList$. The line 5 of the function $PIC2$ adds all the arc-value pairs in $InitACList$. Since after this initialization phase no arc-value pair is added in $InitACList$, the size of this list is

```
function $PIC2()$ : boolean;
  $DeletionList \leftarrow \emptyset; InitACList \leftarrow \emptyset; InitPICList \leftarrow \emptyset;$
  forall $(i,\ a) \in \mathcal{D}$ do
      $S^{PIC}_{ia} \leftarrow \emptyset;\ S^{AC}_{ia} \leftarrow \emptyset;$
      forall $C_{ij} \in \mathcal{C}$ do
          $InitACList \leftarrow InitACList \cup \{[(i,\ j),\ a)]\}; L_{ija} \leftarrow nil;$
          forall $C_{jk} \in \mathcal{C}$ s.t. $(k > j$ and $C_{ik} \in \mathcal{C})$ do
              $InitPICList \leftarrow InitPICList \cup \{[(i,\ a),\ j,\ k]\};$
  while $InitACList \neq \emptyset$ or $InitPICList \neq \emptyset$ or
   $DeletionList \neq \emptyset$ do
      if $DeletionList \neq \emptyset$ then
          choose and delete $(i,\ a)$ from $DeletionList;$
          if not $PropagDeletion(i,\ a,\ DeletionList)$ then
              return $false$
          else if $InitACList \neq \emptyset$ then
                  choose and delete $[(i, j), a]$ from $InitACList;$
                  if $a \in D_i$ and not $SeekACSupport(i, a, j)$ then
                      $remove(D_i, a);$
                      if $D_i = \emptyset$ then return $false;$
                      $DeletionList \leftarrow DeletionList \cup \{(i, a)\};$
              else choose and delete $[(i, a), j, k]$ from $InitPICList;$
                  if $a \in D_i$ and not
                   $SeekPICSupport(i, a, j, nil, k, nil)$ then
                      $remove(D_i, a);$
                      if $D_i = \emptyset$ then return $false;$
                      $DeletionList \leftarrow DeletionList \cup \{(i, a)\};$
  return $true;$

function $PropagDeletion(j, b,$ in out $DeletionList)$ : boolean;
  while $S^{AC}_{jb} \neq \emptyset$ do
      choose and delete $(i, a)$ from $S^{AC}_{jb};$
      if $a \in D_i$ and not $SeekACSupport(i, a, j)$ then
          $remove(D_i, a);$
          if $D_i = \emptyset$ then return $false;$
          $DeletionList \leftarrow DeletionList \cup \{(i, a)\};$
  while $S^{PIC}_{jb} \neq \emptyset$ do
      choose and delete $((i, a), (k, c))$ from $S^{PIC}_{kc};$
      remove $((i, a), (j, b))$ from $S^{PIC}_{kc};$
      if $a \in D_i$ and $((j < k$ and not $SeekPICSupport(i, a, j, b, k, c))$
       or $(j > k$ and not $SeekPICSupport(i, a, k, c, j, b)))$ then
          $remove(D_i, a);$
          if $D_i = \emptyset$ then return $false;$
          $DeletionList \leftarrow DeletionList \cup \{(i, a)\};$

function $SeekACSupport(i,\ a,\ j)$ : boolean;
  while $S^{AC}_{ia}[j] \neq \emptyset$ do
      $b \leftarrow first(S^{AC}_{ia}[j]);$
      if $b \notin D_j$ then delete $b$ from $S^{AC}_{ia}[j]$
      else $S^{AC}_{jb}[i] \leftarrow S^{AC}_{jb}[i] \cup \{a\};$ return $true;$
  while $L_{ija} \leq last(D_j)$ do
      $b \leftarrow next(D_j, L_{ija});$
      $L_{ija} \leftarrow b;$
      if $L_{jib} < a$ then
          if $C_{ij}(a,\ b)$ then
              $S^{AC}_{jb}[i] \leftarrow S^{AC}_{jb}[i] \cup \{a\};$
              return $true;$
  return $false;$

function $SeekPICSupport(i,\ a,\ j,\ b,\ k,\ c)$ : boolean;
  if $b \notin D_j$ then
      $b \leftarrow max(L_{ija}, next(D_j, b));$
      if $b \notin D_j$ then $b \leftarrow next(D_j, b);$
      while $b \neq \infty$ and $(L_{jib} > a$ or not $C_{ij}(a, b))$ do
          $b \leftarrow next(D_j, b);$
      if $b \neq \infty$ then
          $c \leftarrow max(L_{ika}, L_{jkb}, first(D_k));$
          if $c \notin D_k$ then $c \leftarrow next(D_k, c);$
  else
      $c \leftarrow max(L_{ika}, L_{jkb}, next(D_k, c));$
      if $c \notin D_k$ then $c \leftarrow next(D_k, c);$
  while $b \neq \infty$ and $(c = \infty$ or $L_{kjc} > b$ or $L_{kic} > a$ or
   not $C_{ik}(a, c)$ or not $C_{jk}(b, c))$ do
      $c \leftarrow next(D_k, c);$
      if $c = \infty$ then
          $b \leftarrow next(D_j, b);$
          while $b \neq \infty$ and $(L_{jib} > a$ or not $C_{ij}(a, b))$ do
              $b \leftarrow next(D_j, b);$
          if $b \neq \infty$ then
              $c \leftarrow max(L_{ika}, L_{jkb}, first(D_k));$
              if $c \notin D_k$ then $c \leftarrow next(D_k, c);$
  if $b \neq \infty$ then
      add $((i, a), (k, c))$ in $S^{PIC}_{jb}$, $((i, a), (j, b))$ in $S^{PIC}_{kc}$
       and link them
      return $true;$
  return $false;$
```

Figure 2. PIC2.

$O(ed)$. For each value $(i,\ a)$ and each 3-clique $\{i,\ j,\ k\}$ $(j < k)$, $[(i,\ a),\ j,\ k]$ is added in $InitPICList$ exactly once. Therefore, $InitPICList$ has an $O(cd)$ worst case space complexity. For each arc-value pair $[(i,\ j),\ a]$ there is a counter L_{ija} and the size of the L data structure is $O(ed)$. A value $(i,\ a)$ has at most one current support on each constraint C_{ij} and the size of the S^{AC}_{jb} lists is $O(ed)$. $((i,\ a),\ (k,\ c)) \in S^{PIC}_{jb}$ only if $((j,\ b),\ (k,\ c))$ is the current "PIC support" of $(i,\ a)$ on $\{i,\ j,\ k\}$ and no pair of value $((i,\ a),\ (k,\ *))$ is added in S^{PIC}_{j*} as long as $((i,\ a),\ (k,\ c))$ is in S^{PIC}_{jb}. So, the size of the S^{PIC} data structure is $O(cd)$ and the worst case space complexity of PIC2 is $O(ed + cd)$.

6 Experimental Evaluation

We used the CN generator of [7]. 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 (the density $p1$=1 corresponds to the complete graph) and $p2$ the proportion of forbidden pairs of values in a constraint (the tightness). Fig. 3 shows the efficiency of AC7, RPC2, PIC1, PIC2 and Max-RPC1 on complete CNs having 60 variables and 10 values in each initial domain. For each tightness, 300 instances were generated and Fig. 3 presents mean values. These experiments on small complete CNs show the computational advantage of using bidirectionality and storing information to enhance deleted values propagation. Indeed, since these networks are complete, the number of 3-cliques is maximal and PIC2 has to try to extend each value to any two additional variables like PIC1. When no value is deleted, PIC1 is about twice as fast as PIC2 but as soon as some value deletions occur, the advantage of an efficient propagation becomes significant and PIC2 is about seven times as fast as PIC1 for tightness 0.40. So even on small and complete CNs, PIC2 has better behavior than PIC1.

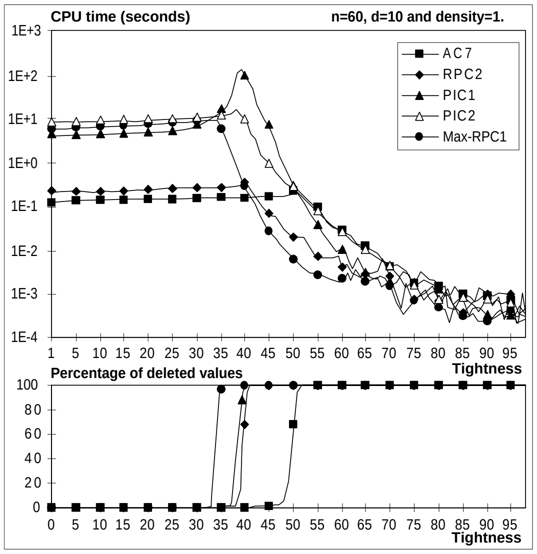

Figure 3. Experimental evaluation on complete CNs with n=60 and d=10.

The advantage of using PIC2 instead of PIC1 becomes obvious on sparse and large CNs. Fig. 4 shows the results on CNs having 250 variables, 30 values in each initial domain and a 5 percents density.

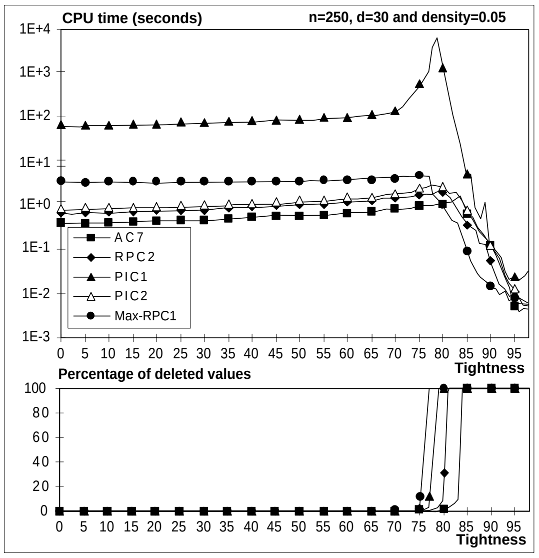

Figure 4. Experimental evaluation on CNs with n=250, d=30 and density=0.05.

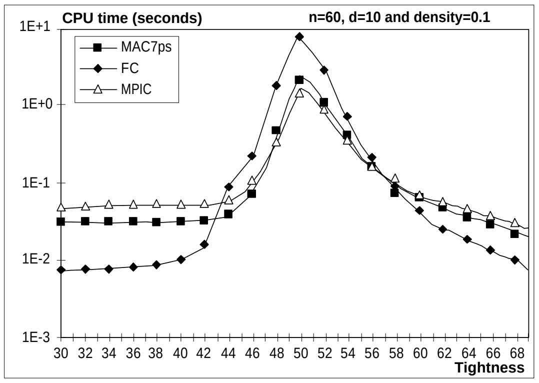

Figure 5. Experimental evaluation on CNs with n=60, d=10 and density=0.1.

There is much more 3-tuples checked by PIC1 than 3-cliques in these relatively sparse CNs (each variable is linked to 12.5 variables in average). PIC2 has better CPU time performances than PIC1 whatever the tightness is. When no value is deleted, PIC2 is already seventy times as fast as PIC1 and obviously PIC2 overcomes even more PIC1 when the number of deleted values increases. At tightness 0.80, PIC2 is more than 1750 times faster than PIC1.

If we consider the other local consistencies, Fig. 3 shows that PIC has bad performances, PIC2 being more expensive than Max-RPC1 while removing few additional values compared to RPC. However on the CNs of Fig. 4 PIC2 has better behavior. Considering the pruning efficiency, PIC is halfway between RPC and Max-RPC while PIC2 requires few additional cpu time compared to RPC2.

Local consistencies can be used during a preprocessing step but it is even more advantageous to use them during search. However, the time required to achieve them must not be prohibitive w.r.t. their pruning efficiency. To know whether it can be advantageous to maintain PIC during search (MPIC), we have compared a search algorithm maintaining PIC based on PIC2 with forward checking and MAC[3] on CNs having 60 variables with 10 values in each initial

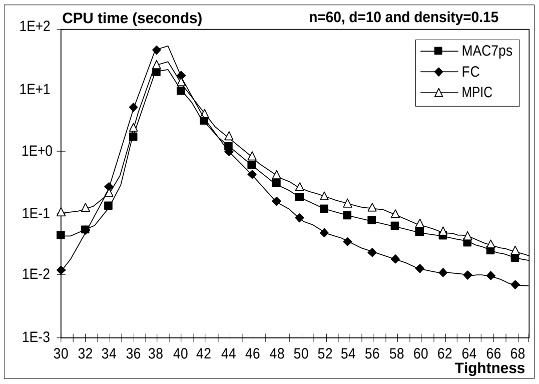

Figure 6. Experimental evaluation on CNs with n=60, d=10 and density=0.15.

domain. The same generator has been used. For each tightness 300 instances were generated and Fig. 5 presents the mean value for density .10. Obvisouly, on easy CNs forward checking has the best cpu time performances but for tightness 0.5 were most of the hard CNs are, maintaining PIC is more time efficient than FC and MAC. Additional experiments show that this advantage is even more important on CNs involving more variables. Fig. 6 shows the results for density .15. The number of 3-cliques in these CNs is more important and maintaining PIC becomes more costly. At this density, maintaining PIC still has better cpu time performances than FC on the hard CNs but MAC is more efficient. Maintaining PIC must therefore be used only when there is few 3-cliques in the constraint graph.

7 Conclusion

In this paper we gave a property of path inverse consistency, which highlights the relations between PIC, RPC and Max-RPC, and which shows the reason why PIC removes only few additional values compared to RPC while Max-RPC is far more pruningful. We proposed an optimal PIC algorithm, called PIC2, based on this property. An experimental evaluation shows that it is far more efficient than PIC1. Furthermore, maintaining PIC during search using PIC2 leads to better cpu time performances than FC and MAC on hard sparse CNs.

REFERENCES

[1] P. Berlandier, *Improving Domain Filtering using Restricted Path Consistency*, In IEEE CAIA-95, Los Angeles CA, 1995.

[2] C. Bessière, E.C. Freuder, and J.C. Régin, *Using inference to reduce arc-consistency computation*, 592–598, In IJCAI-95, Montréal, Canada, 1995.

[3] R. Debruyne and C. Bessière, *From Restricted Path Consistency to Max-Restricted Path Consistency*, 312–326, In CP-97, Linz, Austria, 1997.

[4] R. Debruyne and C. Bessière, *Some Practicable Filtering Techniques for the Constraint Satisfaction Problem*, 412–417, In IJCAI-97, Nagoya, Japan, 1997.

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

[6] E. Freuder and D.C. Elfe, *Neighborood Inverse Consistency Preprocessing*, 202–208, In AAAI-96, Portland OR, 1996.

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

[8] J.-C. Régin, *Développement d'outils algorithmiques pour l'intelligence artificielle. Application à la chimie organique.*, Thèse de doctorat, Université de Montpellier II. In French., 1995.

[9] M. Singh, *Path Consistency Revisited*, In ICTAI-95, Washington D.C., 1995.

[3] We used the MAC7ps version of MAC [8].