International Journal on Artificial Intelligence Tools, Vol. 7, No. 2 (1998) 121–142

EFFICIENT PATH-CONSISTENCY PROPAGATION*

ASSEF CHMEISS
CRIL
Université d'Artois
Rue de l'université - SP 16
62307 Lens Cedex
FRANCE
chmeiss@cril.univ-artois.fr

PHILIPPE JEGOU
LIM, ESA-CNRS 6077
Université de Provence-CMI
39, rue Joliot Curie
13453 Marseille Cedex 13
FRANCE
jegou@lim.univ-mrs.fr

Received 20 October 1997
Accepted 25 January 1998

ABSTRACT

Recently, efficient algorithms have been proposed to achieve arc- and path-consistency in constraint networks. For example, for arc-consistency, there are linear time algorithms (in the size of the problem) which are efficient in practice (e.g. AC-6 and AC-7). The best path-consistency algorithm proposed is PC-$\{5|6\}$ which is a natural generalization of AC-6 to path-consistency. While its theoretical complexity is the best, experimentations show clearly that it is not very efficient in practice. In this paper, we propose two algorithms, one for arc-consistency, AC-8, and the second for path-consistency, PC-8. These algorithms are based on the same principle: to exploit minimal supports as AC-6 and PC-$\{5|6\}$ do, but without recording them. While for AC-8, this approach is of limited interest, we show that for path-consistency, this new approach allows to outperform significantly existing algorithms.

Keywords: Constraint Satisfaction, CSPs, Arc-Consistency, Path-Consistency, Constraint Propagation.

*A preliminary version of this work has been presented at the International Conference on Tools with Artificial Intelligence IEEE-ICTAI'96 [1].

1. Introduction

The (finite) Constraint Satisfaction Problem (CSP) formalization has been a productive tool with Artificial Intelligence and related areas. Both within the framework of Constraint Programming and CSPs, the techniques of preprocessing phase based on filtering algorithms were shown to be very important for the search phase. In particular, two filtering methods have been studied, these methods exploit two properties of local consistency: arc-consistency and path-consistency. Concerning arc-consistency methods, there is a linear time algorithm (AC-6 of Bessière [2]) which is efficient in practice [3]. So, arc-consistency can be considered now as a basic tool in the fields of Constraint Programming and Constraint Reasonning. Nevertheless, for some application, the filtering corresponding to arc-consistency is limited because it is less powerfull than filtering associated to path-consistency. The best path-consistency algorithm proposed is PC-{5|6} (for PC-5 of [4] [5] and PC-6 of [6] [7]), a natural generalization of AC-6 to path-consistency. Its time complexity is $O(n^3d^3)$ and its space complexity is $O(n^3d^2)$, where n is the number of variables and d is the size of domains. Unfortunately, we have remarked that PC-{5|6}, though it is widely better than PC-4 [8] [9], is not very efficient in practice, specialy for those classes of problems that require a large space to be run. So, it seems that a such filtering cannot be used in practical applications or to be integrated as a basic tool in constraint solvers [10]. A possible way to avoid this problem is in defining new properties of consistency between arc- and path-consistency (see [11] or [12]). In this paper, we consider a different way, trying to optimize path-consistency filtering, proposing a new algorithm.

One of the conclusion of [7] indicates a way to optimize path-consistency filterings: to make a compromise between time and space in order to find a new algorithm with real practical efficiency. The algorithm we present in this paper, called PC-8, seems to be able to realize this goal. We relaxed the constraint of time complexity to limit space complexity. PC-8 deals with the notion of supports, but contrary to PC-{5|6}, PC-8 does not record supports. While for each pair of compatible values, PC-{5|6} records $n-2$ minimal supports, PC-8 only looks for supports when it is necessary. So, space complexity of PC-8 is only $O(n^2d)$. As a consequence, the time complexity of PC-8 is worse than the one of PC-{5|6} since it is $O(n^3d^4)$. Nevertheless, the simplicity of the algorithm and its data structures induce a really better practical efficiency for PC-8. Moreover, when the size of domains is a constant for the considered application (it is possible for real life problems), PC-8 becomes optimal in time complexity, $O(n^3)$, and in space, $O(n^2)$.

The principle used in PC-8 can be applied to arc-consistency. So, we introduce a new algorithm achieving arc-consistency, called AC-8. Its space complexity is $O(n)$ while its time complexity is $O(ed^3)$ where e is the number of constraints.

This paper is organized as following. Section 2 recalls some definitions and

notations on CSPs, and then presents the principles of constraint propagation based on supports for arc-consistency and path-consistency, i.e. for algorithms AC-4, AC-6, PC-4 and PC-{5|6}. Sections 3 and 4 describe respectively AC-8 and PC-8 while section 5 presents experimental results about the comparison between AC-3, AC-4, AC-6 and AC-8, and then between PC-2, PC-{5|6} and PC-8 on random CSPs.

2. Preliminaries

Definition 1 *A binary CSP is defined by* (X, D, C, R)*, where* X *is a set of* n *variables* $\{x_1, \ldots x_n\}$ *and* D *is a set of* n *finite domains* $\{D_1, \ldots D_n\}$ *such that* D_i *is the set of the possible values for variable* x_i*.* C *is a set of* e *binary constraints where each* $C_{ij} \in C$ *is a constraint between the variables* x_i *and* x_j *defined by its associated relation* R_{ij}*. So,* R *is a set of* e *relations, a binary relation* R_{ij} *between variables* x_i *and* x_j*, being a subset of the Cartesian product of their domain that defines the allowed pairs of values for* x_i *and* x_j *(i.e* $R_{ij} \subset D_i \times D_j$*).*

For the networks of interest here, we require that $(b, a) \in R_{ji} \Leftrightarrow (a, b) \in R_{ij}$. The fact that $(a, b) \in R_{ij}$ will be denoted by $R_{ij}(a, b)$ is true. If there is no constraint between the variables x_i and x_j, we consider the universal relation $R_{ij} = D_i \times D_j$. A CSP may be represented by a constraint graph (X, C) in the form of a network in which nodes represent variables and arcs connect variables that appear in the same constraint. An *instantiation* of the variables in X is an n-tuple $(v_1, v_2, \ldots, v_n)$ representing an assignement of $x_i \in X$ to v_i. A *consistent instantiation* of a network is an instantiation of the variables such that the constraints between variables are satisfied, i.e $\forall i, j : 1 \leq i < j \leq n, C_{ij} \in C \Rightarrow R_{ij}(v_i, v_j)$. A consistent instantiation is also called a *solution*. For a given CSP, the problem is either to know if there exists any solution or to find one solution or all solutions. The decision problem is known to be NP-complete. Since CSPs are NP-Complete, many techniques have been defined to restrict the size of search space. For example, preprocessing methods based on network consistency algorithms are of great interest in the field of CSPs. These algorithms are based on consistency properties like arc-consistency or path-consistency. We recall below their definitions.

Definition 2 *A domain* D_i *of* D *is arc-consistent iff,* $\forall a \in D_i, \forall x_j \in X$ *such that* $C_{ij} \in C$*, there exists* $b \in D_j$ *such that* $R_{ij}(a, b)$*. A CSP is arc-consistent iff* $\forall D_i \in D, D_i \neq \emptyset$ *and* D_i *is arc-consistent.*

Definition 3 *A pair of variables* $\{x_i, x_j\}$ *is path-consistent iff* $\forall (a, b) \in R_{ij}, \forall x_k \in X$*, there exists* $c \in D_k$ *such that* $R_{ik}(a, c)$ *and* $R_{jk}(b, c)$*. A CSP is path-consistent iff* $\forall x_i, x_j \in X$ *the pair* $\{x_i, x_j\}$ *is path-consistent.*

While filterings based on arc-consistency remove values from domains if they do not satisfy arc-consistency, filterings based on path-consistency remove pairs of values from relations if they do not satisfy path-consistency. So, if a constraint is

not defined between a pair of variables, the universal relation is then considered. As a consequence, the constraint graph can be completed in such cases. Different algorithms achieving arc-consistency or path-consistency in constraint networks have been proposed. For arc-consistency, these algorithms are AC-1 [13], AC-3 [14], AC-4 [8], AC-5 [15], AC-6 [2] and AC-7 [3], and for path-consistency, these algorithms are PC-1 [16], PC-2 [14], PC-4 [8] [9], PC-5 [4] [5], PC-6 [6] [7]. Note that PC-5 and PC-6 correspond to the same algorithm which has been proposed independently by their authors; so, in the sequel, these algorithms will be denoted PC-{5|6}. In Table 1. and Table 2., we recall their time and space complexities. Note that here, space complexity is related to the size of additionnal data-structures induced by algorithms to run. This remark is important since the size of the problem for path-consistency is in the worst case $O(n^2d^2)$ because all the constraints must be represented, that is $O(n^2)$, and the size of space required for representing a constraint is $O(d^2)$.

Table 1. Complexity of arc-consistency algorithms

Algorithm	Time	Space
AC-1	$O(ned^5)$	$O(e+nd)$
AC-3	$O(ed^3)$	$O(e+nd)$
AC-4	$O(ed^2)$	$O(ed^2)$
AC-5	$O(ed^3)$	$O(e+nd)$
AC-6	$O(ed^2)$	$O(ed)$

Table 2. Complexity of path-consistency algorithms

Algorithm	Time	Space
PC-1	$O(n^5d^5)$	$O(n^3d^2)$
PC-2	$O(n^3d^5)$	$O(n^3)$
PC-4	$O(n^3d^3)$	$O(n^3d^3)$
PC-{5|6}	$O(n^3d^3)$	$O(n^3d^2)$

Efficient algorithms for arc-consistency and path-consistency propagation are based on the notion of supports: AC-4 and AC-6 for arc-consistency, PC-4 and PC-{5|6} for path-consistency. The notion of support in constraint propagation has been introduced and exploited efficiently by Mohr and Henderson in [8]. This principle has been optimized by Bessière for AC-6, by introducing the notion of minimal support. Below, we formalize these notions.

Definition 4 *For arc-consistency, a support for a value $a \in D_i$ w.r.t. a constraint C_{ij} is a value $b \in D_j$ compatible with a, i.e. such that $R_{ij}(a,b)$ is true. Given an arbitrary ordering of the values in each domain, the minimal support for a value*

$a \in D_i$ w.r.t. a constraint C_{ij}, is the first support $b \in D_j$ w.r.t. the considered ordering of values in D_j.

This notion can be generalized to path-consistency:

Definition 5 *For path-consistency, a support for a pair of values $(a, b) \in R_{ij}$ is a value $c \in D_k$ such that the relations $R_{ik}(a, c)$ and $R_{jk}(b, c)$ hold. The value $c \in D_k$ is the minimal support for the pair of values $(a, b) \in R_{ij}$ if c is the first support of (a, b) in D_k w.r.t. the considered ordering of values in D_k.*

Algorithms which are based on (minimal) supports are build on the same scheme. After a first phase related to the initializations, that is the initialization of data structures related to supports, and the deletion of trivially inconsistent values (for arc-consistency) or pair of values (for path-consistency), the second phase processes the propagation. This propagation is driven by the list of deleted values (or pair of values) which have not yet been propagated; we call that list *List-D*. When all deleted objects have been propagated, the algorithm stops. The scheme of these algorithms is given below.

Algorithm CONSISTENCY;
begin
 Initialization;
 while *List-D* $\neq \emptyset$ **do begin** { propagation}
  Choose an object δ in *List-D*;
  Propagate(δ)
 end
end;

The purpose of the primitive *Propagate*(δ) is to find new values (or pair of values) to be deleted and to verify that other values (or pair of values) remain consistent. If δ represents the last support for a value (or pair of values), this value (or pair of values) is then deleted and inserted in the list of propagations *List-D*. So the property which is maintained during the propagation is that a value (or a pair of values) is consistent, or is inconsistent and has already been propagated, or has not yet been propagated but then appears in the list *List-D*.

Note that the number of supports for arc-consistency is $O(ed^2)$ while it is $O(n^3d^3)$ for path-consistency. This fact explains the worst-case space complexity of AC-4 and PC-4 since these algorithms need to represent all the supports. In [2], Bessière has proposed an optimization of AC-4, the algorithm AC-6. While AC-4 needs to represent all supports for all values, AC-6 considers only one support by value w.r.t. each constraint. AC-6 considers ordered domains and then it considers minimal support w.r.t. orderings in domains. So the number of supports to be recorded is then bounded by $O(ed)$.

This approach has been generalized to path consistency resulting the algorithm PC-{5|6} [6] [4] [7] [5]. As a consequence, space complexity for PC-{5|6} is limited to $O(n^3d^2)$, that is the number of minimal supports while its time complexity is $O(n^3d^3)$ as for PC-4. Nevertheless, to obtain a such complexity time, PC-{5|6} needs to use particular data structures that allow to be theoretically optimal (double-linked lists, crossed references). Unfortunately, the managment of these data structures reduce its practical efficiency (see section Experiments in the last part of this paper). As a consequence, PC-{5|6} seems practicable only for small CSPs. Consider for example $n = 128$ and $d = 8$; required space to run PC-{5|6} will be $2^{27} \approx 13 \times 10^6$ space units.

It seems that for arc-consistency, it is hard to optimize theoretical and practical efficiency. On the contrary, existing algorithms as PC-2 or PC-{5|6} are not efficient in practice. The motivation here is to present an algorithm that allows to run path-consistency in practice, not necessarily with an optimal theoretical time complexity. PC-8 realizes this goal by making a compromise between time and space complexity to establish an usable algorithm.

3. Arc-consistency in $O(n)$ space complexity

AC-8 is based on supports but without recording any of them. When a value $a \in D_i$ is removed from its domain, AC-8 records the reference of the variable x_i, that is the number i, in the list of propagation now denoted *List-AC*. Propagations will be realized w.r.t. variables in this list. Suppose that a variable x_i is removed from the list. Then, all neighbouring variables will be considered, i.e. for all $x_j \in X$ such that $C_{ji} \in C$, and for each value $b \in D_j$, AC-8 will ensure that there is a value $a \in D_i$ such that $R_{ij}(a, b)$ holds. Unlike AC-6, AC-8 has to start again the search from the first value of the domains. If no support a of b is found in D_i, then b must be deleted, and the number of the variable, namely j must be inserted in *List-AC*. To ensure that j is not duplicated in *List-AC*, we must maintain an array of booleans, denoted *Status-AC*, recording the status of variables. So, the data structures used for AC-8 are the list *List-AC* containing variables which have lost some values in their domains and not propagated yet, and the boolean table *Status-AC*[i] that always verifies $\{i \in$ *List-AC* $\Leftrightarrow$ *Status-AC*[i]$\}$. The initialization phase consists in checking if, for each $x_j \in X$ such that $C_{ij} \in C$, there exists at least one support per value $a \in D_i$. So, if for some variable x_j, a has no support in D_j, it must be deleted and i must be added to the list, updating *Status-AC*[i]. Three constant time functions are used to handle domains D_i:

- $First(D_i)$ returns the first value in the domain D_i,
- $Last(D_i)$ returns the last value in the domain D_i and
- $Next\_value(a, D_i)$ which returns the successor of a in D_i.

These functions are used in the *Withoutsupport-AC* function which checks if $b \in D_j$ has a support in D_i. Concerning the propagation phase, AC-8 restarts

looking for a new support from the first value of the domain. Finally, the scheme of AC-8 is a classical scheme of propagation as the algorithm CONSISTENCY (which is defined in section 2): the algorithm stops when the list of propagation becomes empty, each step corresponding to the propagation of a deletion:

```
Function Withoutsupport-AC( i, j: integer; b : values) : boolean;
{ If b ∈ D_j has a support a in D_i, then Withoutsupport-AC returns False,}
{ else Withoutsupport-AC returns True}
var a:values;
begin
   a ← First(D_i);
   while a < Last(D_i) and not R_ij(a, b) do a ← Next_value(a, D_i);
   Withoutsupport-AC ← not R_ij(a, b)
end; { Withoutsupport-AC}


Procedure Initialization-AC;
begin
 List-AC ← ∅;
 for i = 1 to n do Status-AC[i] ← False;
 for i = 1 to n do
  for j such that C_ij ∈ C do for b ∈ D_j do
   if Withoutsupport-AC(i, j, b) then begin
    D_j ← D_j\{b};
    if not Status-AC[j] then begin
     Append(List-AC,j);
     Status-AC[j] ← True
    end
   end
end; { Initialization-AC }


procedure Propagate-AC(in i:integer);
begin
 for j such that C_ij ∈ C do for b ∈ D_j do
     if Withoutsupport-AC(i, j, b) then begin
       D_j ← D_j\{b};
       if not Status-AC[j] then begin
             Append(List-AC,j);
             Status-AC[j] ← True
       end
     end
end; { Propagate-AC }
```

Algorithm AC-8;
begin
Initialization-AC;
while *List-AC* $\neq \emptyset$ **do begin** { propagation}
Choose i in *List-AC*;
List-AC $\leftarrow$ *List-AC* $\setminus\{i\}$;
Status-AC[i] $\leftarrow$ *False*;
Propagate-AC(i)
end
end;

The space complexity of AC-8 is bounded by the size of the table *Status-AC*, since the maximum size of the list *List-AC* is bounded by the number of true values in this table, that is at most n. So, the space complexity of AC-8 is $O(n)$. Note that $O(n)$ is the size of the additional space for AC-8 to run since the size of the data, that is the CSP is bounded by ed^2. It is clear that the time complexity of the initialization step is $O(ed^2)$. Concerning the propagation step, since we have at most nd values that can be deleted, there is at most $n \times d$ calls to the procedure *Propagate-AC* in which the number of iterations is bounded by $n_i \times d$ where n_i is the number of neighbouring variables of x_i in the constraint graph. The cost of the function *Withoutsupport-AC* is bounded by the domain's size d. Consequently, the time complexity of one call to the procedure *Propagate-AC* is bounded by $n_i \times d^2$. For all propagations, the total cost is then bounded by $\sum_{i=1}^{n} d \times n_i \times d^2 = ed^3$. So, the time complexity of AC-8 is $O(ed^3)$.

4. Path-consistency in $O(n^2d)$ space complexity

Path-consistency algorithm PC-8 is based on the same principles as AC-8 and appears to be an optimization of PC-7 with better space complexity[1]. So, like AC-8, PC-8 is based on supports without recording any of them. When a pair of values (a, c) is removed from a relation R_{ik}, two 3-tuples will be recorded in the list of propagations called *List-PC*: (i, a, k) and (k, c, i). So, propagations will be realized w.r.t. these 3-tuples. For example, if (i, a, k) is propagated, then for all values $b \in D_j$ such that $R_{ij}(a, b)$ holds, the propagation must verify that there exists $c' \in D_k$ which supports $(a, b) \in R_{ij}$. If no support c' of (a, b) is found in D_k, then (a, b) must be deleted, and two 3-tuples, namely (i, a, j) and (j, b, i) have to be inserted in *List-PC*. Figure Fig. 1 shows this principle. In this figure, the pair of values (a, c) has been deleted, so the 3-tuple (i, a, k) has been inserted in *List-PC* and the function Withoutsupport-PC tries to find the first value c' w.r.t. the ordering in D_k which supports $(a, b) \in R_{ij}$, that is such that $R_{ik}(a, c')$ and

[1] Algorithm PC-7 has been presented during the National Conference on Artificial Intelligence AAAI'96 [17]. The space complexity of PC-7 is $O(n^2d^2)$; the experimental results of PC-7 and PC-8 are similar.

$R_{jk}(b,c')$ hold. As AC-8, PC-8 has to start again the search from the first value of the domains.

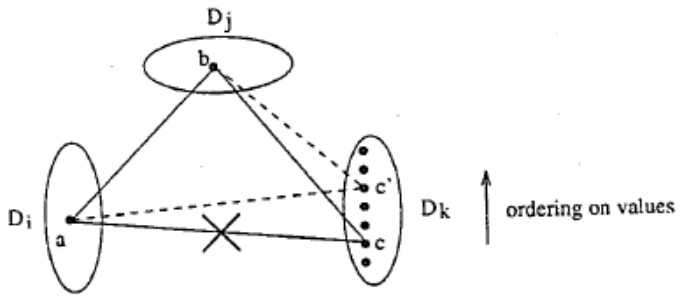

Fig. 1. After the deletion of (a,c), the new minimal support of (a,b) in D_k is c' since it is the minimal value in D_k which is consistent with a and b.

To ensure that a 3-tuple is not duplicated in *List-PC*, we must maintain an array of booleans, denoted *Status-PC*, recording the status of 3-tuples. So, the data structures used by PC-8 are the list *List-PC* of 3-tuples corresponding to deleted pairs of values and not propagated yet, and the boolean table *Status-PC* that always verifies $\{(i,a,j) \in List \Leftrightarrow Status\text{-}PC[i,a,j]\}$. The initialization phase consists in checking if there exists at least one support $c \in D_k, \forall x_k \in X$ per pair of values $(a,b) \in R_{ij}, i \neq k \neq j$. So, if it is not the case, (a,b) must be deleted and two 3-tuples, (i,a,j) and (j,b,i), must be added to the list, updating *Status-PC*. To handle domains, PC-8 used the three constant time functions used by AC-8, i.e. $First(D_i)$, $Last(D_i)$ and $Next\_value(a,D_i)$. These functions are used in the *Withoutsupport-PC* function which checks if $(a,b) \in R_{ij}$ has a support in D_k. Finally, the scheme of PC-8 is the classical scheme of propagation CONSISTENCY which is presented in section 2 since propagations stop when the *List-PC* becomes empty:

Function Withoutsupport-PC(i,j,k: **integer**; a,b : **values) : boolean;**
{ If there is a support c of $(a,b) \in R_{ij}$ in D_k,}
{ then *Withoutsupport-PC* returns *False*,}
{ else it returns *True*}
begin
  $c \leftarrow$ First(D_k);
  while $c < Last(D_k)$ **and not** $(R_{ik}(a,c)$ **and** $R_{jk}(b,c))$ **do**
    $c \leftarrow$ Next_value(c,D_k);
  Withoutsupport-PC $\leftarrow$ **not** $(R_{ik}(a,c)$ **and** $R_{jk}(b,c))$
end; { Withoutsupport-PC }

```
Procedure Initialization-PC;
begin
  List-PC ← ∅;
  for i,j = 1 to n : i ≠ j do for a ∈ D_i do Status-PC[i,a,j] ← False;
  for i,j,k = 1 to n : i < j, k ≠ i, k ≠ j do for (a,b) ∈ R_ij do
    if Withoutsupport-PC(i,j,k,a,b) then begin
      R_ij(a,b) ← False; R_ji(b,a) ← False;
      if not Status-PC[i,a,j] then begin
          Append(List-PC,(i,a,j));
          Status-PC[i,a,j] ← True
      end;
      if not Status-PC[j,b,i] then begin
          Append(List-PC,(j,b,i));
          Status-PC[j,b,i] ← True
      end
    end
end; { Initialization-PC }

Procedure Propagate-PC(in i,k:integer;in a: values);
begin
   for j = 1 to  n: j ≠ i, j ≠ k do
     for b ∈ D_j such that R_ij(a,b) = True do
       if Withoutsupport-PC(i,j,k,a,b) then begin
         R_ij(a,b) ← False; R_ji(b,a) ← False;
         if not Status-PC[i,a,j] then begin
               Append(List-PC,(i,a,j));
               Status-PC[i,a,j] ← True
         end;
         if not Status-PC[j,b,i] then begin
               Append(List-PC,(j,b,i));
               Status-PC[j,b,i] ← True
         end
       end
end; { Propagate-PC }

Algorithm PC-8;
begin
  Initialization-PC;
  while List-PC ≠ ∅ do begin { propagation}
    Choose (i,a,k) in List-PC; List-PC ← List-PC \{(i,a,k)};
    Status-PC[i,a,k] ← False;
    Propagate-PC(i,k,a)
  end
end;
```

The space complexity of PC-8 is exactly the size of the table *Status-PC*, i.e. n^2d since there is at most n^2d 3-tuples that can be recorded in the list *List-PC* because of the table. So, space complexity is $O(n^2d)$. Nevertheless, we must note that the size of the filtered CSP is bounded by n^2d^2. So, $O(n^2d)$ is clearly the space complexity for PC-8 to run, without taking into account the the size of the filtered problem. It is clear that the time complexity of the initialization step is $O(n^3d^3)$. Concerning the propagation step, since we have at most n^2d^2 pairs of values that can be deleted, there is at most $n^2 \times d^2$ calls to the procedure *Propagate-PC* in which the number of iterations is bounded by nd. Finally, the cost of the function *Withoutsupport-PC* is bounded by the domain's size d. Consequently, the time complexity of PC-8 is bounded by $O(n^2d^2 \times nd \times d) = O(n^3d^4)$.

Note that if the size of the domains d is a constant of the problem (this is possible for some applications), the time complexity of PC-8 becomes $O(n^3)$ i.e the same as for PC-4 and PC-{5|6}; so PC-8 complexity is then optimal in space and time.

5. Experiments

In order to validate our results, we have compared experimentally our algorithms with existing ones. Experiments were performed over randomly generated CSPs using the random model proposed in [18]. The generator considers four parameters: the number of variables n, the domain size d, the tightness of the constraints t, and the constraint graph density cd. The *constraint tightness* t is the fraction of the possible pairs that are not allowed by the constraints between two variables: $t = 1 - \frac{|R_{ij}|}{d^2}$. The *constraint graph density* is a value cd varying between 0 and 1 indicating the fraction of the possible constraints beyond the minimum $n - 1$ (for a connected acyclic graph). Note that if $cd = 1$, the number of constraints is $(n^2 - n)/2$, which corresponds to a complete constraint graph.

We have chosen two measures of comparison, on the one hand the number of consistency checks performed by each algorithm, on the other hand the CPU execution time. Figures provide results in terms of the number of consistency checks as well as the CPU time which are represented by the *y-axis*. In all figures, the *x-axis* represents the constraints tightness; it varies from 0.1 to 0.9 with a step of 0.05 for arc-consistency and 0.1 for path-consistency algorithms. For each 4-tuples (n, d, t, cd) i.e. one point over the curves, 20 randomly CSPs were generated. Results reported so far represent the average over the 20 problems for each of the algorithms.

Our AC-8 was compared with different algorithms of arc-consistency, which are AC-3, AC-4 and AC-6. While we realized experiments on a large number of classes of CSPs, for lack of place, the experiments presented here were performed on CSPs with $n = 128$, $d = 8$ and $cd = 0.4$ and with $n = 128$, $d = 16$ and $cd = 0.5$. Note that for all classes, the results are similar, and so, the results reported here present a summary of all the results we have observed.

Figures Fig.2 through Fig.5 report results obtained by the different arc-consistency algorithms.

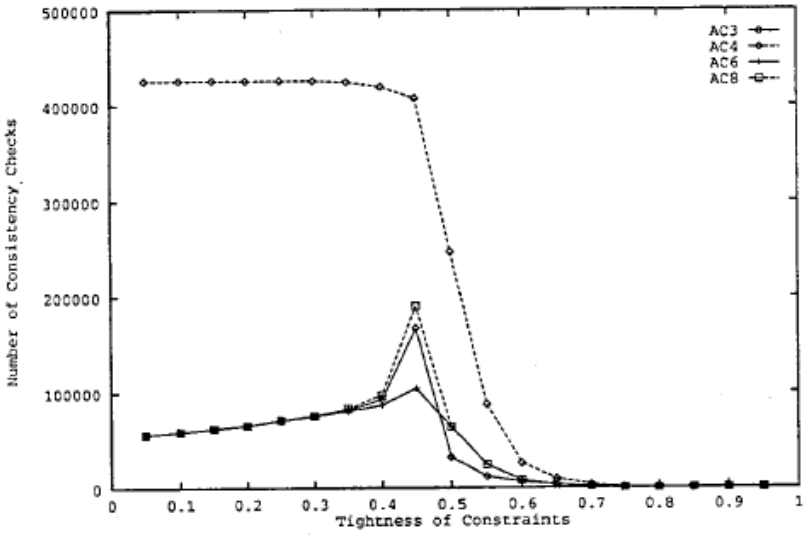

Fig. 2. Consistency checks ($n = 128, d = 8$ and $cd = 0.4$)

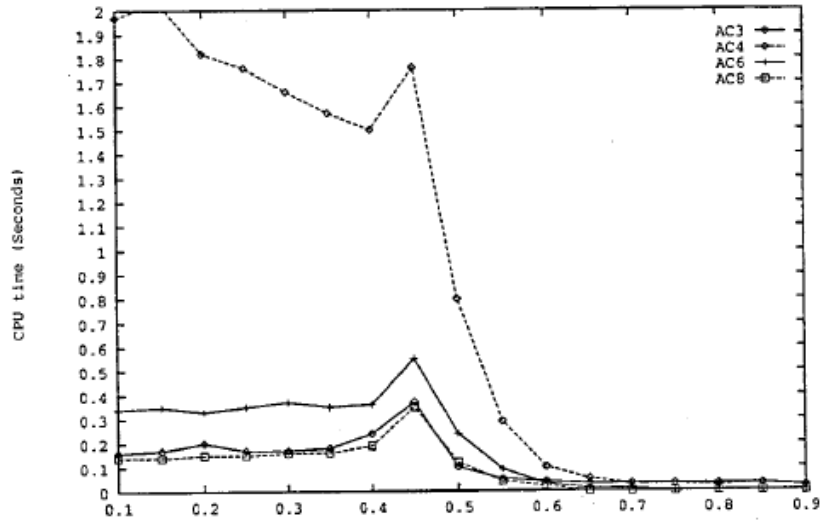

Fig. 3. CPU time ($n = 128, d = 8$ and $cd = 0.4$)

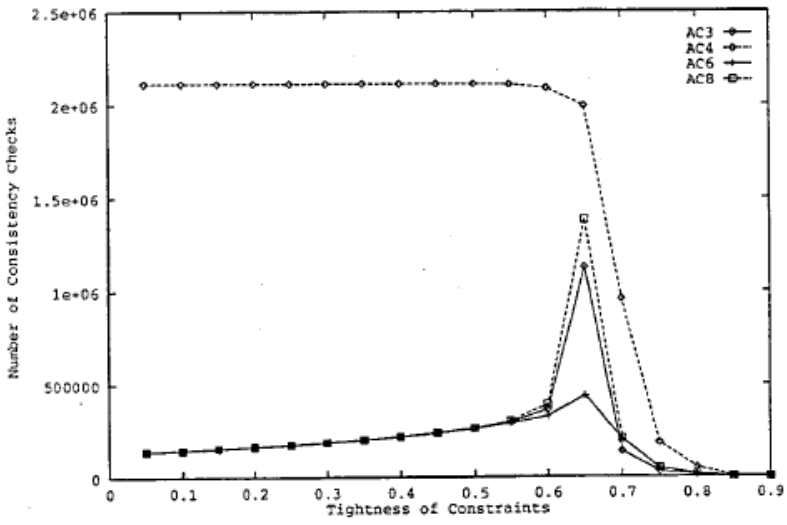

Fig. 4. Consistency Checks ($n = 128, d = 16$ and $cd = 0.5$)

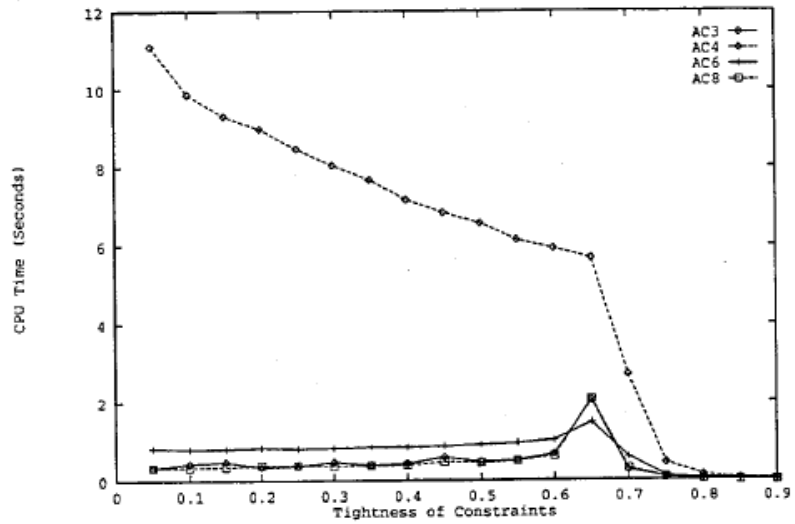

Fig. 5. CPU time ($n = 128, d = 16$ and $cd = 0.5$)

Looking at the curves which represent these results, we remark that for the number of consistency checks as measure of efficiency, AC-6 is often the best one. It is due to the fact that while AC-6 looks for a new support from the next value with respect to the previous support, both AC-8 and AC-3 restart from the first value in the domain. By contrast, for the CPU time measure, the behaviour of AC-3 and

AC-8 are similar and both outperform AC-6 except for the point (128,16,0.65,0.5) which corresponds to the hardest classe in the figure Fig.5 for which AC-6 has a better performance. We can explain this by the fact that AC-6 has to memorize supports and by consequence to handle a complicated data structures while both AC-8 and AC-3 work with a simple data structures very easy to be treated.

For path-consistency algorithms, PC-8 was compared with PC-2 and PC-{5|6}. We have chosen these algorithms on account of their time and space complexity. The choice of PC-2 is motivated by the fact that it has a better space complexity and it is preferable in practice. Moreover, PC-2's time complexity is also $\Omega(n^3d^3)$. We have conserved PC-{5|6} to show that its efficiency in practice is lower than that of PC-8. Here we do not consider PC-4 because it is really inefficient in practice [17].

Figures Fig.6 through Fig.9 present comparisons between PC-2, PC-{5|6} and PC-8 for a CSPs with $n = 32$ and $d = 8$. Here, results taken into account concern CSPs the graph density of which $cd \in \{0.2, 0.5\}$.

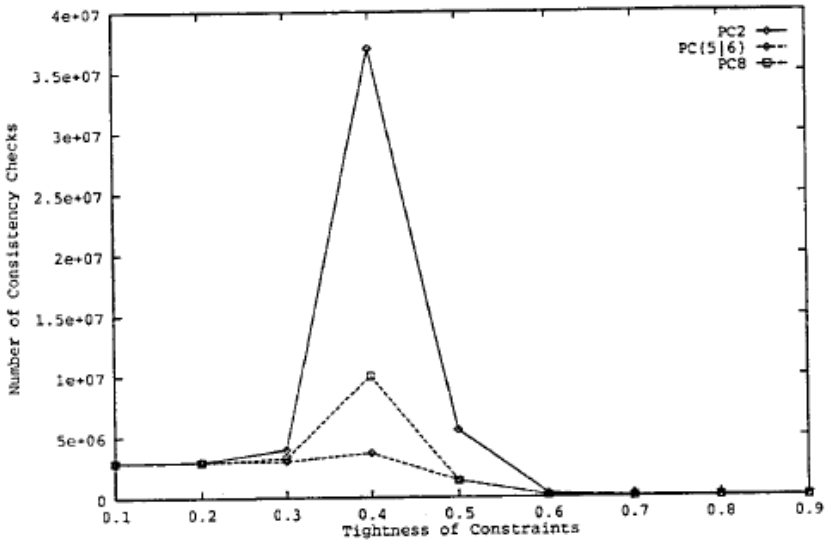

Fig. 6. Consistency checks for $n = 32, d = 8$ and $cd = 0.2$

As for experiments on arc-consistency, we did not reported here all the results, but the results reported here summarize all the results we have observed.

It is clear that PC-{5|6} realizes the smallest number of consistency checks. By contrast, PC-8 seems to be the best algorithm for CPU time as a measure of performance. Figures concerning CPU time show that PC-8 always outperforms

both PC-2 and PC-{5|6} except for $t = 0.1$. Also, PC-8 outperform PC-2 for the number of consistency checks as a measure of efficiency. For the number of consistency checks, we remark that PC-8 performs as much as PC-{5|6}, this is due to the fact that PC-8 always restarts from the first value of domains (during the search for a support).

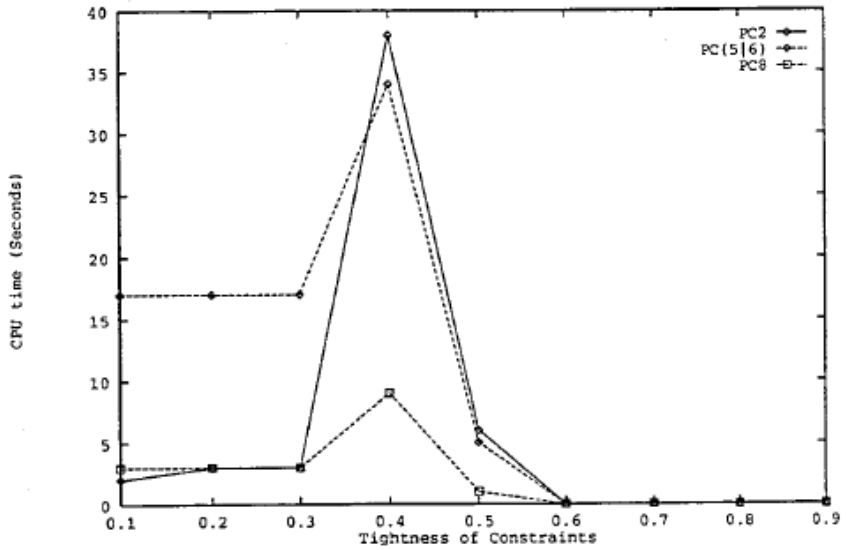

Fig. 7. CPU time for $n = 32, d = 8$ and $cd = 0.2$

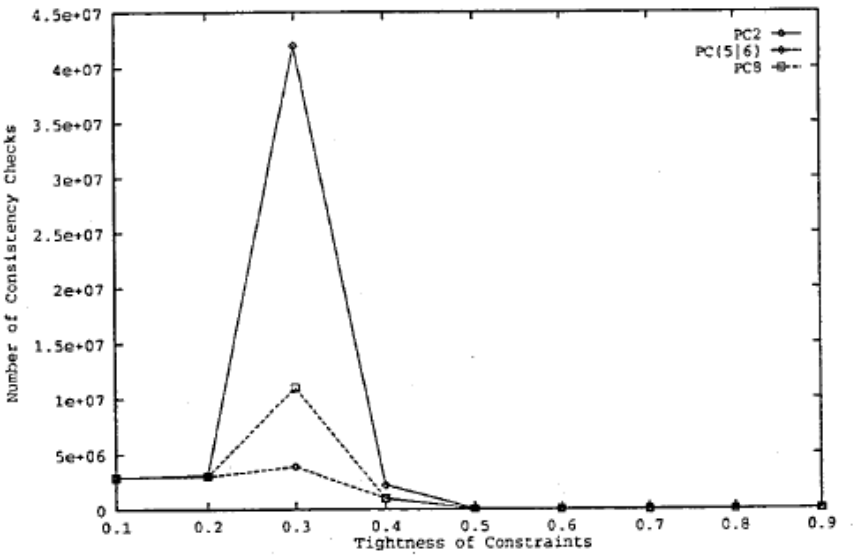

Fig. 8. Consistency checks $n = 32, d = 8$ and $cd = 0.5$

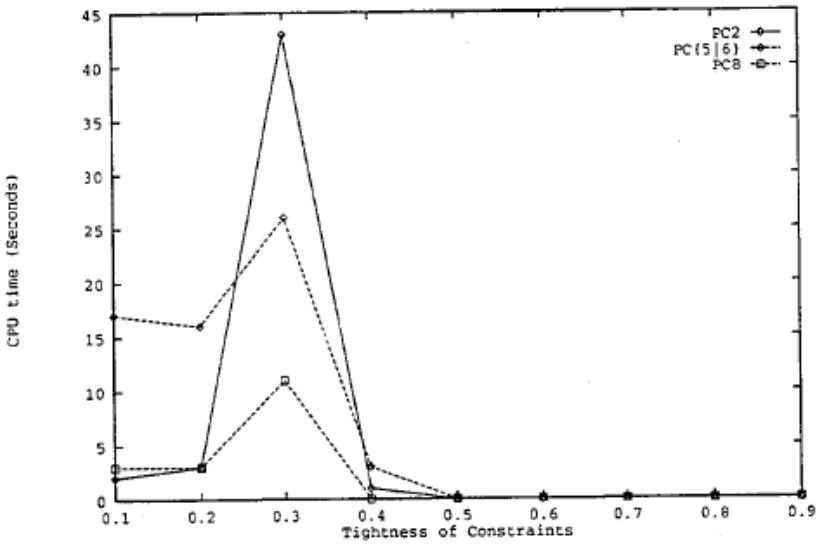

Fig. 9. CPU time for $n = 32, d = 8$ and $cd = 0.5$

In order to better evaluate the efficiency of these algorithms we have performed experiments on CSPs for which we varied the size of the domains. Two classes of problems have been tested : the first is with $n = 16$ and $d = 16$ while the second is with $n = 10$ and $d = 25$. For these problems we concentrate on the efficiecny of PC-8 and PC-{5|6}. Moreover, we consider only the execution time as a measure of performance since for the number of consistency checks, obtained results for these classes confirm those presented above which is compatible with theoretical evaluation of the time complexity. We chose to present the problems for which the behaviour of these algorithms are close. It is a matter of the hardest classes.

Now if we get a look at figure Fig. 10 which presents the execution time of PC-8 and PC-{5|6} for CSPs with graph density equals to 30%. In this figure, curves $PC\{5|6\}cpu$ and $PC8cpu$ present the CPU time of PC-{5|6} and PC-8 respecitvely while curves $PC\{5|6\}$ and $PC8$ show the total time consumed by the system and each of the algorithms. We can remark that for PC-8 these curves are identical which is not the case for PC-{5|6}.

We recall that the essential difference between PC-{5|6} and PC-8 is in memorizing supports. Figure Fig. 11 shows the CPU time and the total execution time of PC-{5|6} and PC-8 for the initialization phase. We remark that PC-{5|6} spends a long time to realize this step. It is due to the fact that PC-{5|6} treats a lists of supports and spended time during this phase is essentially in handling these lists. We mention that for CSPs which verify path-consistency, recording supports is time consuming. Moreover, it will be the case for CSPs for which the path-inconsistency

can be proven during the initialization step.

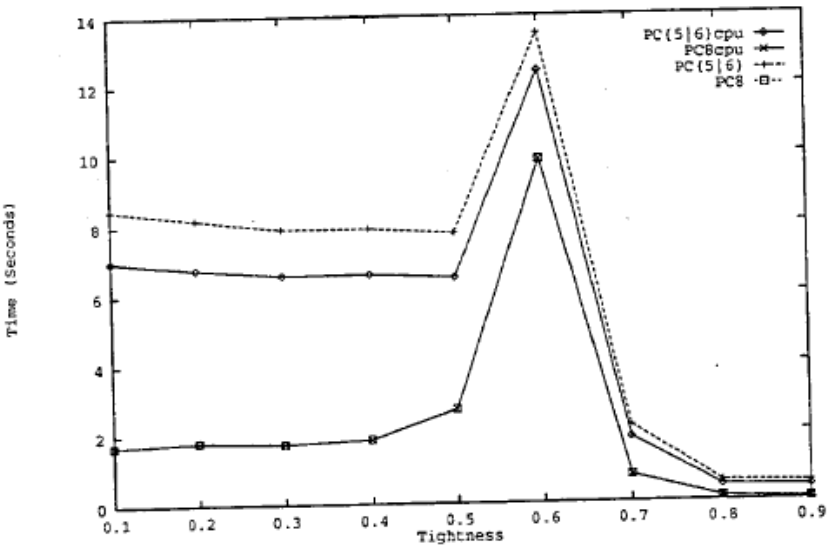

Fig. 10. CPU time and total time for $n = 16$, $d = 16$ and $cd = 0.3$

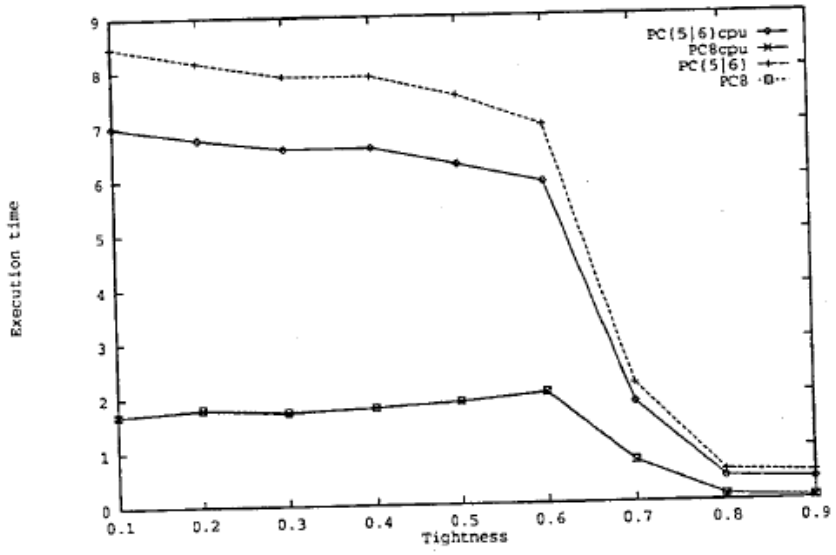

Fig. 11. Initialization phase : $n = 16$, $d = 16$ and $cd = 0.3$

Other experiments were also performed on CSPs with $n = 10$ and $d = 25$ which allow us to study the behaviour of these algorithms for CSPs for which the size of domains is greater than the number of variables.

For these classes of problems the behaviour of PC-2 with respect to PC-{5|6} or PC-8 is similar to its behaviour on the classes studied above. Concernig the initialization phase for PC-{5|6} and PC-8 we have remarked the same behaviour as for CSPs with $n = d = 16$, i.e. PC-{5|6} spends much time in realizing this step. Figures Fig. 12 through Fig. 15 show some results obtained for these classes.

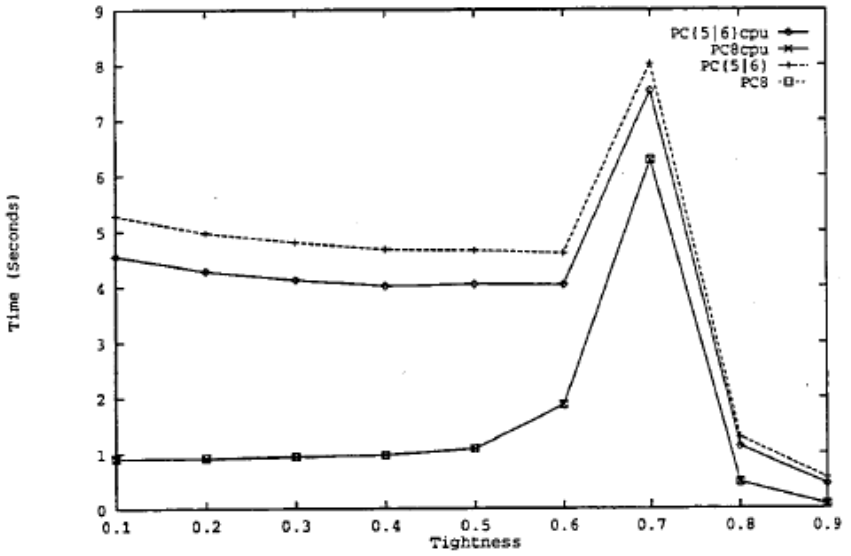

Fig. 12. CPU time and total time for $n = 10$, $d = 25$ and $cd = 0.4$

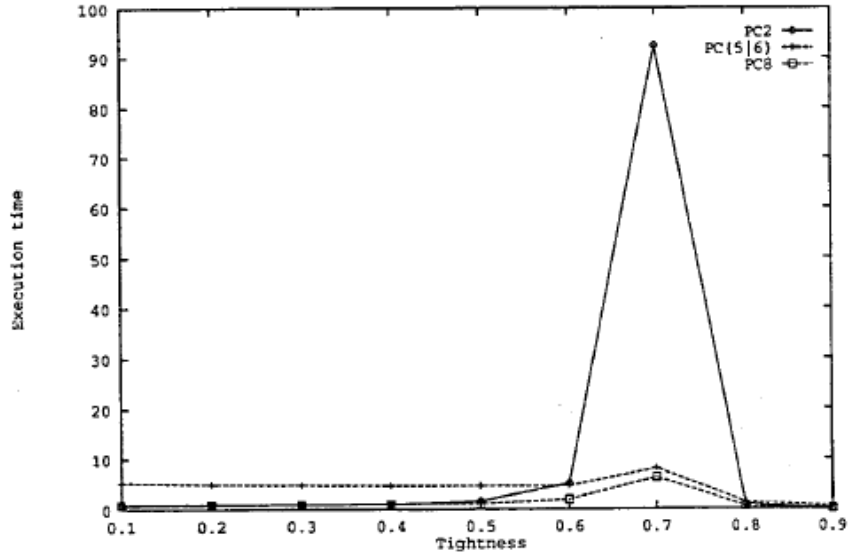

Fig. 13. CPU time and total time for $n = 10$, $d = 25$ and $cd = 0.4$

We remark that PC-{5|6} outperforms PC-8 in one point in the figure Fig. 14 for CSPs with graph density equals to 70% when the tightness equals to 60%. For this point (n=10, d=25, t=0.6 and cd=0.7), over the 20 problems three are path-inconsistent and their path-inconsistencies were been proven during the propagation phase. In fact for these three CSPs PC-{5|6} was faster than PC-8 in proving path-inconsistency. Also, PC-{5|6} was better than PC-8 for another point, namely (n=10,d=25, t=0.6 and cd=0.8), but this time with 16 path-inconsistency problems, the corresponding curve is not presented here. Otherwise, PC-8 often outperforms PC-{5|6}.

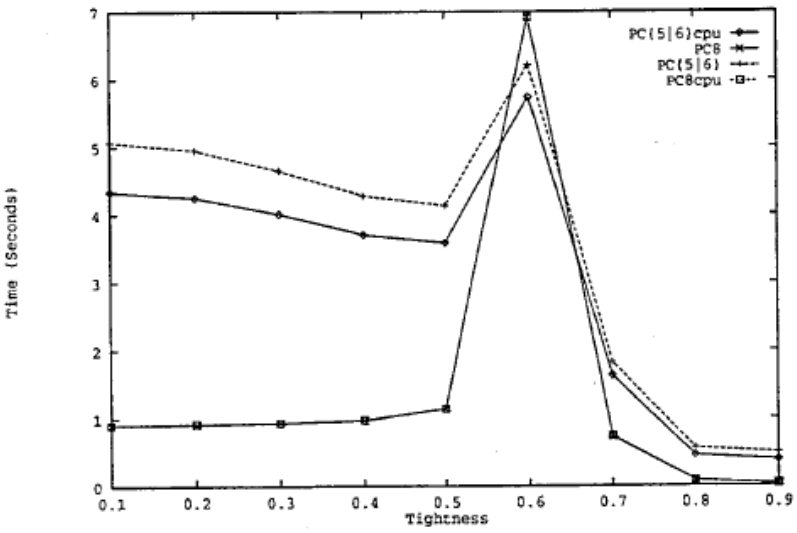

Fig. 14. CPU time and total time for $n = 10$, $d = 25$ and $cd = 0.7$

Finally, note that PC-8 was also tested for CSPs with $n = 64$ and $d = 8$ for which the CPU execution time was between 0 and 90 seconds, then for CSPs with $n = 128$ and $d = 8$ for which CPU time was about 800 seconds for some classes of problems (the hardest ones). For these classes, it was generally impossible to run PC-2 or PC-{5|6} because of their time consuming.

Observing these experiments, we can conclude that PC-8 seems to be the best path-consistency filtering algorithm, specially it allows to handle CSPs of large size while other algorithms failed. Contrary to the theoretical evaluation of time complexity, the surprise comes from the efficiency of PC-8 versus PC-{5|6} which could be explained by the time lost by PC-{5|6} in treating the used data structures which require handling doubly linked lists with crossed references. If a such data structures lead to an optimal theoretical time complexity, they increase the CPU time because of the required number of operations for each propagation step, which

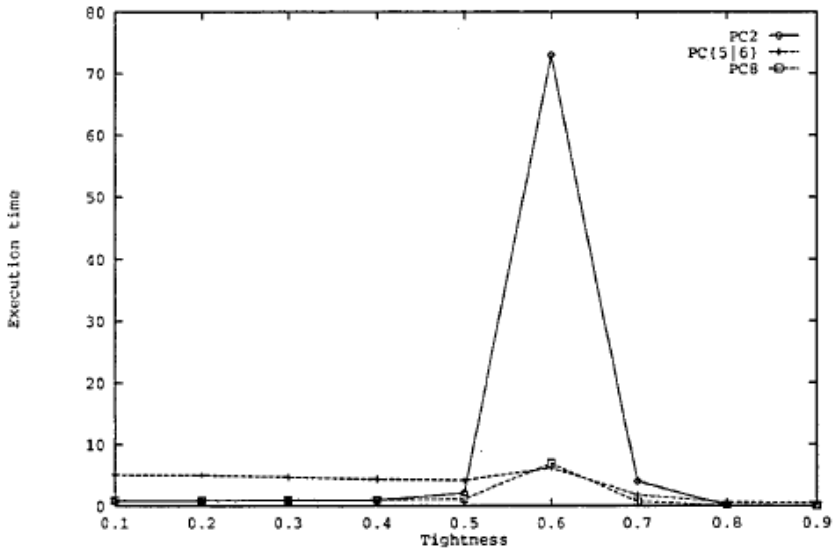

Fig. 15 CPU time and total time for $n = 10$, $d = 25$ and $cd = 0.7$

is greater than the one of PC-8. The multiplicative hidden constant of PC-{5|6} seems to be widely greater than PC-8 one. *A contrario*, PC-8 depends only on the lists of modified arcs with some value in their domains, i.e. a 3-tuples of the form (i, j, a), and another point is that the implementation of PC-8 is very simple. These different points enable PC-8 to have a remarkable efficiency in time and a weak memory usage. Finally, this work can be situated in the same spirit as the work of Wallace which explains the better performance of AC-3 with respect to AC-4 [19].

6. Conclusion

In this paper, we have presented two new algorithms to achieve partial consistency in constraint networks. The first one, called AC-8 achieves arc-consistency while the second called PC-8 achieves path-consistency. These algorithms are based on the same principle. Unlike AC-6 or PC-{5|6} which manage minimal supports in recording them, AC-8 and PC-8 use minimal supports without any recording.

As a consequence, the additive space complexity of AC-8 is $O(n)$ while its time complexity is $O(ed^3)$ as that of AC-3. While the advantage of AC-8 w.r.t. AC-6 is not clear in practice (it performs more consistency checks than AC-6) and the CPU time required to run is close to the one of AC-3, the advantage of the principle used in AC-8, and PC-8, is more clear for path-consistency. Indeed, space complexity of PC-8 is $O(n^2d)$, that is currently the best space complexity for algorithms to achieving path-consistency. Nevertheless, time complexity of PC-8 is $O(n^3d^4)$, that is slightly more than the one of PC-6 but the simplicity of the algorithm and of

its data structures allow PC-8 to be really efficient in practice, i.e. the CPU time required to run is less than that for PC-{5|6}. Moreover, for cases such that the size of domains is a constant parameter of problems (this fact frequently appears in real life applications), PC-8 becomes theoretically optimal for time complexity, that is $O(n^3)$ like PC-{5|6}. Finally, experiments being performed on randomly generated CSPs, it may be interesting to study the behaviour of these algorithms on a real world problems.

References

[1] A. Chmeiss and Ph. Jégou, *Two New Constraint Propagation Algorithms Requiring Small Space Complexity*, Proc. of the International Conference on Tools with Artificial Intelligence, IEEE-ICTAI'96 (1996) 286–289.

[2] C. Bessière, *Arc-Consistency and Arc-Consistency Again*, J. Artif. Intell. **65** (1994) 179–190.

[3] C. Bessière, E. Freuder and J.C. Régin *Using Inference to Reduce Arc Consistency Computation*, Proc. of IJCAI'95 (1995) 592–598.

[4] M. Singh, *Path Consistency Revisited*, Proc. of the 5th International Conference on Tools for Artificial Intelligence (1995) 318–325.

[5] M. Singh, *Path Consistency Revisited*, J. IJAIT **5** (1996) 127–141.

[6] A. Chmeiss and Ph. Jégou, *Partial and global path Consistency revisited*, Technical report No. 1995.120, L.I.M., Marseille (1995).

[7] A. Chmeiss, *Sur la consistance de chemin et ses formes partielles*, In Actes du Congrès AFCET-RFIA 96, Rennes, France (1996) 212–219.

[8] R. Mohr and T.C. Henderson, *Arc and Path Consistency Revisited*, J. Artif. Intell. **28** (1986) 225–233.

[9] C.C. Han and C.H. Lee, *Comment on Mohr and Henderson's Path Consistency Algorithm*, J. Artif. Intell. **36** (1988) 125–130.

[10] P. Codognet and G. Nardiello, *Path Consistency in clp(FD)*, Proc. 1st International Conference on Constraints in Computational Logics, Munchen, Germany, Lecture Notes in Computer Science, vol. **845** (1994) 201–216.

[11] H. Bennaceur, *Partial Consistency for Constraint Satisfaction Problems*, Proc. ECAI'94, Amsterdam, The Netherelands (1994) 120–124.

[12] P. Berlandier, *Filtrage de problèmes par consistance de chemin restreinte*, Revue d'Intelligence Artificielle **9 (1)** (1995) 225–238.

[13] D.L. Waltz, *Generating semantic descriptions from drawings of scenes with shadows*, MAC AI-TR-271, MIT (1972)

[14] A.K. Mackworth, *Consistency in networks of relations*, J. Artif. Intell. **8** (1977) 99–118.

[15] P. Van Hentenryck, Y. Deville and C.M. Teng, *A generic arc-consistency algorithm and its specializations*, J. Artif. Intell. **57-2:3** (1992) 291–322.

[16] U. Montanari, *Network of Constraints: fundamental properties and applications to picture processing*, Inform. Sci. **7** (1974) 95–132.

[17] A. Chmeiss and Ph. Jégou, *Path-Consistency: When Space Misses Time*, Proc. of the National Conference on Artificial Intelligence, AAAI'96 (1996) 196–201.

[18] P. Hubbe and E. Freuder, *An Efficient Cross-Product Representation of the Constraint Satisfaction Problem Search Space*, Proc. of AAAI'92 (1992) 421–427.

[19] R. Wallace, *Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs*, Proc. IJCAI'93 (1993)