Path Consistency by Dual Consistency

Christophe Lecoutre, Stéphane Cardon, and Julien Vion

CRIL – CNRS FRE 2499,
rue de l'université, SP 16
62307 Lens cedex, France
$\{lecoutre, cardon, vion\}$@cril.univ-artois.fr

Abstract. Dual Consistency (DC) is a property of Constraint Networks (CNs) which is equivalent, in its unrestricted form, to Path Consistency (PC). The principle is to perform successive singleton checks (i.e. enforcing arc consistency after the assignment of a value to a variable) in order to identify inconsistent pairs of values, until a fixpoint is reached. In this paper, we propose two new algorithms, denoted by sDC2 and sDC3, to enforce (strong) PC following the DC approach. These algorithms can be seen as refinements of Mac Gregor's algorithm as they partially and totally exploit the incrementality of the underlying Arc Consistency algorithm. While sDC3 admits the same interesting worst-case complexities as PC8, sDC2 appears to be the most robust algorithm in practice. Indeed, compared to PC8 and the optimal PC2001, sDC2 is usually around one order of magnitude faster on large instances.

1 Introduction

Constraint Networks (CNs) can naturally represent many interesting problems raised by real-world applications. To make easier the task of solving a CN (i.e. the task of finding a solution or proving that none exists), one usually tries to simplify the problem by reducing the search space. This is called inference [7].

Consistencies are properties of CNs that can be exploited (enforced) in order to make inferences. *Domain filtering consistencies* [6] allow to identify inconsistent values while *relation filtering consistencies* allow to identify inconsistent tuples (pairs when relations are binary) of values. For binary networks, Arc Consistency (AC) and Path Consistency (PC) are respectively the most studied domain and relation filtering consistencies. Many algorithms have been proposed to enforce PC on a given CN, e.g. PC3 [18], PC4 [9], PC5 [21], PC8 [5] and PC2001 [2].

Recently, a new relation filtering consistency, called Dual Consistency (DC) has been introduced [12]. The principle is to record inconsistent pairs of values identified after any variable assignment followed by an AC enforcement. Just like SAC (Singleton Arc Consistency), a domain filtering consistency, DC is built on top of AC. Interestingly, when applied on all constraints of a binary instance (including the implicit universal ones), DC is equivalent to PC, but when it is applied conservatively (i.e. only on explicit constraints of the binary network), Conservative DC (CDC) is stronger than Conservative PC (CPC). In [12], CDC is investigated: in particular, its relationship with other consistencies is studied and a cost-effective algorithm, called sCDC1, is proposed.

In this paper, we focus on DC and propose two new algorithms to enforce (strong) PC by following the principle underlying DC. It means that we establish (strong) PC by performing successive singleton checks (enforcing AC after a variable assignment) as initially proposed by Mac Gregor [16]. These two algorithms, denoted by sDC2 and sDC3, correspond to refined versions of the algorithm sCDC1, as they partially and totally exploit the incrementality of the underlying AC algorithm, respectively.

In terms of complexity, sDC2 admits a worst-case time complexity in $O(n^5d^5)$ and a worst-case space complexity in $O(n^2d^2)$ where n is the number of variables and d the greatest domain size. On the other hand, by its full exploitation of incrementality, sDC3 admits an improved worst-case time complexity in $O(n^3d^4)$ while keeping a worst-case space complexity in $O(n^2d^2)$. It makes sDC3 having the same (worst-case) complexities as PC8, the algorithm shown to be the fastest to enforce PC so far [5].

The paper is organized as follows. First, we introduce constraint networks and consistencies. Then, we describe two new algorithms to enforce strong path consistency, following the dual consistency approach, and establish their worst-case complexities. Finally, before concluding, we present the results of an experimentation we have conducted.

2 Constraint Networks and Consistencies

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom^P(X)$, which represents the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$, called scope and denoted $scp(C)$, and has an associated relation denoted $rel^P(C)$, which represents the set of tuples allowed for the variables of its scope. When possible, we will write $dom(X)$ and $rel(C)$ instead of $dom^P(X)$ and $rel^P(C)$. X_a denotes a pair (X, a) with $X \in \mathscr{X}$ and $a \in dom(X)$ and we will say that X_a is a value of P. d will denote the size of the greatest domain, and λ the number of allowed tuples over all constraints of P, i.e. $\lambda = \sum_{C \in \mathscr{C}} |rel(C)|$. If P and Q are two CNs defined on the same sets of variables $\mathscr{X}$ and constraints $\mathscr{C}$, then we will write $P \leq Q$ iff $\forall X \in \mathscr{X}$, $dom^P(X) \subseteq dom^Q(X)$ and $\forall C \in \mathscr{C}$ $rel^P(C) \subseteq rel^Q(C)$. $P < Q$ iff $P \leq Q$ and $\exists X \in \mathscr{X} \mid dom^P(X) \subset dom^Q(X)$ or $\exists C \in \mathscr{C} \mid rel^P(C) \subset rel^Q(C)$. A binary constraint is a constraint which only involves two variables. In the following, we will restrict our attention to binary networks, i.e., networks that only involve binary constraints. Further, without any loss of generality, we will consider that the same scope cannot be shared by two distinct constraints. The density D of a binary CN is then defined as the ratio $2e/(n^2 - n)$.

A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable. Constraint networks can be characterized by properties called consistencies. The usual way to exploit them is to enforce them on a CN, while preserving the set of solutions. It then consists in identifying and removing some inconsistent values (e.g. with arc consistency), inconsistent pairs of values (e.g. with path consistency), etc. Here, "inconsistent" means that the identified values, pairs of values, etc. correspond to *nogoods*, i.e. cannot participate to any solution. We start introducing the consistencies we are interested in at the level of pairs of values. From now on, we will consider a binary constraint network $P = (\mathscr{X}, \mathscr{C})$.

Definition 1. *A pair (X_a, Y_b) of values of P such that $X \neq Y$ is:*

- *arc-consistent (AC) iff either $\nexists C \in \mathscr{C} \mid scp(C) = \{X, Y\}$ or $(X_a, Y_b) \in rel(C)$;*
- *path-consistent (PC) iff (X_a, Y_b) is AC and $\forall Z \in \mathscr{X} \mid Z \neq X \wedge Z \neq Y$, $\exists c \in dom(Z)$ such that (X_a, Z_c) is AC and (Y_b, Z_c) is AC.*

We can now introduce Arc Consistency (AC) and Path Consistency (PC) with respect to a CN.

Definition 2. *A value X_a of P is AC iff $\forall Y (\neq X) \in \mathscr{X}$, $\exists b \in dom(Y) \mid (X_a, Y_b)$ is AC. P is AC iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, X_a is AC.*

Definition 3. *A pair (X, Y) of distinct variables of $\mathscr{X}$ is PC iff $\forall a \in dom(X)$, $\forall b \in dom(Y)$, (X_a, Y_b) is PC. P is PC iff any pair of distinct variables of $\mathscr{X}$ is PC.*

AC admits the following property: for any network P, there exists a greatest subnetwork of P which is arc-consistent, denoted by $AC(P)$. Remark that if any variable in $AC(P)$ has an empty domain, P is unsatisfiable. We will denote this by $AC(P) = \bot$. For any value X_a, we will write $X_a \in AC(P)$ iff $a \in dom^{AC(P)}(X)$ (we will consider that $X_a \notin \bot$). Finally, $P|_{X=a}$ represents the network obtained from P by restricting the domain of X to the singleton $\{a\}$. The singleton check of a pair (X, a) corresponds to determine whether or not $AC(P|_{X=a}) = \bot$. When the check is positive, we say that (X, a) is singleton arc inconsistent. We can now introduce Dual Consistency (DC) [12].

Definition 4. *A pair (X_a, Y_b) of values of P such that $X \neq Y$ is dual-consistent (DC) iff $Y_b \in AC(P_{|X=a})$ and $X_a \in AC(P_{|Y=b})$. A pair (X, Y) of distinct variables of $\mathscr{X}$, is DC iff $\forall a \in dom(X)$, $\forall b \in dom(Y)$, (X_a, Y_b) is DC. P is DC iff any pair of distinct variables of $\mathscr{X}$ is DC.*

Surprisingly, DC appears to be equivalent to PC although it was predictable since McGregor had already proposed an AC-based algorithm to establish sPC [16]. A proof can be found in [12].

Proposition 1. *DC = PC*

Finally, from any relation filtering consistency, it is possible to obtain a new consistency by additionally considering Arc Consistency. Classically, a network is strong path-consistent, denoted sPC, iff it is both arc-consistent and path-consistent. We can remark that enforcing AC (only once) on a PC network is sufficient to obtain an sPC network.

Algorithm 1: AC (P = $(\mathscr{X}, \mathscr{C})$: Constraint Network, Q: Set of Variables)

```
1 while $Q \neq \emptyset$ do
2     pick and delete $X$ from $Q$
3     foreach $C \in \mathscr{C} \mid X \in scp(C)$ do
4         let $Y$ be the second variable involved in $C$
5         if $revise(C, Y)$ then
6             $Q \leftarrow Q \cup \{Y\}$
```

Algorithm 2: FC (P = $(\mathscr{X}, \mathscr{C})$: Constraint Network, X: Variable)

```
1 foreach $C \in \mathscr{C} \mid X \in scp(C)$ do
2     let $Y$ be the second variable involved in $C$
3     $revise(C, Y)$
```

3 New Algorithms to Enforce Strong Path Consistency

The two algorithms that we propose to establish sPC are called sDC2 and sDC3. Before describing them, we need to quickly introduce a basic AC algorithm, FC and a direct adaptation of sCDC1. Remember that the general principle is to perform successive singleton checks until a fix-point is reached.

3.1 AC and FC

The description of the sDC algorithms (which enforce sPC since DC = PC) is given in the context of using an underlying coarse-grained AC algorithm, such as AC3 [15], AC2001/3.1 [2] or AC3^{rm} [13], with a variable-oriented propagation scheme. If $P = (\mathscr{X}, \mathscr{C})$ is a CN, then $AC(P, Q)$ with $Q \subseteq \mathscr{X}$ means enforcing arc consistency on P from the given propagation set Q. Q contains all variables that can be used to detect arc-inconsistent values in the domain of other variables.

The description is given by Algorithm 1. As long as there is a variable in Q, one is selected and the revision of any variable connected to it (via a constraint) is performed. A revision is performed by a call to the function $revise$ (e.g. see [2]) specific to the chosen coarse-grained arc consistency algorithm, and entails removing values in the domain of the given variable that have become inconsistent with respect to the given constraint. When a revision is effective (at least one value has been removed), the set Q is updated. We will not discuss here about potential optimizations.

In the sequel, we will also refer to Forward Checking (FC) [10] which is an algorithm that maintains a partial form of arc consistency. More precisely, whenever a variable is assigned during search, only unassigned variables connected to it are revised. This is described by Algorithm 2. Remark that the worst-case time complexity of a call to FC is $O(nd)$ since there are at most $n - 1$ revisions and the revision of a variable against an assigned variable is $O(d)$.

Algorithm 3: sDC1($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1  $P \leftarrow AC(P, \mathscr{X})$
2  $X \leftarrow first(\mathscr{X})$
3  $marker \leftarrow X$
4  repeat
5      if $|dom(X)| > 1$ then
6          if $checkVar1(P, X)$ then
7              $P \leftarrow AC(P, \{X\})$
8              $marker \leftarrow X$
9      $X \leftarrow next\text{-}modulo(\mathscr{X}, X)$
10 until $X = marker$
```

3.2 Algorithm sDC1

Here, our purpose is to establish sPC (strong PC). As we know that CDC (i.e. DC only considered between variables connected by a constraint) is equivalent to PC when the constraint graph is complete, we can slightly adapt the algorithm sCDC1 introduced in [12]. We quickly describe this algorithm (in order to make this paper self-contained as much as possible), called sDC1, before proposing some improvements to it.

In Algorithm 3, AC is enforced first (line 1), and then, at each iteration of the main loop, a different variable is considered (let us call it the current variable). Assuming here that $\mathscr{X}$ is ordered, $first(\mathscr{X})$ returns the first variable of $\mathscr{X}$, and $next\text{-}modulo(\mathscr{X}, X)$ returns the variable that follows X in $\mathscr{X}$, if it exists, or $first(\mathscr{X})$ otherwise. Calling $checkVar1$ at line 6 enables us to make all possible inferences from X (this is depicted below). If any inference is performed, $true$ is returned and arc consistency is re-established (line 7). Remark that, when the domain of the current variable is singleton, no inference can be made anymore since the network is always maintained arc-consistent. This is the reason of the test at line 5 (not present in sCDC1).

Performing all inferences with respect to a variable X is achieved by calling the function $checkVar1$ (Algorithm 4). For each value a in the domain of X, AC is enforced on $P|_{X=a}$. If a is singleton arc-inconsistent, then a is removed from the domain of X (line 5). Otherwise (lines 8 to 12), for any value Y_b present in P and absent in P', the tuple (X_a, Y_b) is removed from $rel(C)$.

In [12], it is proved that sCDC1 always terminates, enforces sCDC and admits a worst-case time complexity of $O(\lambda end^3)$. A direct consequence is that sDC1 enforces sPC and admits a worst-case time complexity of $O(\lambda n^3 d^3)$.

3.3 Algorithm sDC2

The algorithm sDC2 can be seen as a refinement of sDC1. The idea is to limit the cost of enforcing AC each time we have to perform a singleton check. In sDC1, we apply $AC(P|_{X=a}, \{X\})$. This is strictly equivalent to $AC(FC(P|_{X=a}, X), Q)$ where Q denotes the set of variables of P whose domain has been reduced by $FC(P|_{X=a}, X)$. Indeed, when applying $AC(P|_{X=a}, \{X\})$, we first start by revising each variable $Y \neq X$

Algorithm 4: checkVar1($P = (\mathscr{X}, \mathscr{C})$: CN, X: Variable): Boolean

```
1  $modified \leftarrow false$
2  foreach $a \in dom^P(X)$ do
3      $P' \leftarrow AC(P|_{X=a}, \{X\})$
4      if $P' = \bot$ then
5          remove $a$ from $dom^P(X)$
6          $modified \leftarrow true$
7      else
8          foreach $Y \in \mathscr{X} \mid Y \neq X$ do
9              let $C \in \mathscr{C} \mid scp(C) = \{X, Y\}$
10             foreach $b \in dom^P(Y) \mid b \notin dom^{P'}(Y)$ do
11                 remove $(X_a, Y_b)$ from $rel^P(C)$
12                 $modified \leftarrow true$
13 return $modified$
```

against X, and put this variable in the propagation queue if some value(s) of its domain has been removed. The first pass of AC enforcement is then equivalent to forward checking. Except for the first singleton check of (X, a), in sDC2, we will apply $AC(FC(P|_{X=a}, X), Q')$ where Q' is a set of variables built from some information recorded during propagation. The point is that necessarily $Q' \subseteq Q$, which means that sDC2 is less expensive than sDC1 (since some useless revisions may be avoided, and, as we will see, the cost of managing the information about propagation is negligible). Roughly speaking, we partially exploit the incrementality of the underlying arc consistency algorithm in sDC2. An arc consistency algorithm is said incremental if its worst-case time complexity is the same when it is applied one time on a given network P and when it is applied up to nd times on P where, between two consecutive executions, at least one value has been deleted. All current arc consistency algorithms are incremental.

To enforce sPC on a given network P, one can then call the second algorithm we propose, sDC2 (see Algorithm 5). This algorithm differs from sDC1 by the introduction of a counter and a data structure, denoted $lastModif$, which is an array of integers. The counter is used to count the number of turns of the main loop (see lines 4 and 6). The use of $lastModif$ is defined as follows: for each variable X, $lastModif[X]$ indicates the number of the last turn where one inference concerning X has been performed. Such an inference can be the removal of a value in $dom(X)$ or the removal of a tuple in the relation associated with a constraint involving X. When the function $checkVar2$ returns $true$, it means that at least one inference concerning X has been performed. This is why $lastModif[X]$ is updated (line 10). Then, AC is maintained (line 11), and if at least one value has been removed since X is the current variable ($nbValues(P)$ indicates the cumulated number of values in P), we consider that each variable has been "touched" at the current turn. Of course, a more subtle update of the array $lastModif$ can be conceived (looking for variables really concerned by the inferences performed when maintaining AC). From experience, it just bloats the algorithm without any noticeable benefit.

Algorithm 5: sDC2($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1  $P \leftarrow AC(P, \mathscr{X})$
2  $X \leftarrow first(\mathscr{X})$
3  $marker \leftarrow X$
4  $cnt \leftarrow 0$
5  repeat
6      $cnt \leftarrow cnt + 1$
7      if $|dom(X)| > 1$ then
8          $nbValuesBefore \leftarrow nbValues(P)$
9          if $checkVar2(P, X, cnt)$ then
10             $lastModif[X] \leftarrow cnt$
11             $P \leftarrow AC(P, \{X\})$
12             if $nbValues(P) \neq nbValuesBefore$ then
13                 $lastModif[Y] \leftarrow cnt, \forall Y \in \mathscr{X}$
14             $marker \leftarrow X$
15     $X \leftarrow next\text{-}modulo(\mathscr{X}, X)$
16 until $X = marker$
```

All inferences, if any, concerning a given variable X, are achieved by calling the function $checkVar2$ (Algorithm 6). For each value a of $dom(X)$, if this is the first call to $checkVar2$ for X (line 3), then we proceed as usually. Otherwise, incrementality is partially exploited by removing first at least all values that were removed by the last AC enforcement of (X, a). This is done by calling FC. Then, we apply AC from a propagation queue composed of all variables that were concerned by at least one inference during the last $|\mathscr{X}| - 1$ calls to $checkVar2$. The remaining of the function is identical to $checkVar1$, except for the update of $lastModif$ (line 13) whenever a tuple is removed ($lastModif[X]$ is not updated since done at line 10 of Algorithm 5).

Proposition 2. *The algorithm sDC2 enforces sPC.*

Proof. First, it is immediate that any inference performed by sDC2 is correct. Completeness is guaranteed by the following invariant: when $P' \leftarrow AC(FC(P|_{X=a}, X), Q)$ with $Q = \{Y \mid cnt - lastModif[Y] < |\mathscr{X}|\})$ is performed at line 4 of Algorithm 6, we have $P' = AC(P|_{X=a}, \mathscr{X})$. It simply means that P' is either arc-consistent or equal to $\bot$. The reason is that the network P is maintained arc-consistent whenever a modification is performed (line 11 of Algorithm 5) and that any inference performed with respect to a value X_a has no impact on $P|_{X=b}$, where b is any other value in the domain of the variable X. The invariant holds since, whenever an inference is performed, it is recorded in $lastModif$. □

Proposition 3. *The worst-case time complexity of sDC2 is $O(\lambda n^3 d^3)$ and its worst-case space complexity is $O(n^2 d^2)$.*

Proof. Remember that the worst-case time and space complexities of sDC1 are respectively $O(\lambda n^3 d^3)$ and $O(n^2 d^2)$. Hence, to obtain the same result for sDC2, it suffices

Algorithm 6: checkVar2($P = (\mathscr{X}, \mathscr{C})$: CN, X: Variable, cnt: integer): Boolean

```
$modified \leftarrow false$
foreach $a \in dom^P(X)$ do
    if $cnt \leq |\mathscr{X}|$ then $P' \leftarrow AC(P|_{X=a}, \{X\})$
    else $P' \leftarrow AC(FC(P|_{X=a}, X), \{Y \mid cnt - lastModif[Y] < |\mathscr{X}|\})$
    if $P' = \bot$ then
        remove $a$ from $dom^P(X)$
        $modified \leftarrow true$
    else
        foreach $Y \in \mathscr{X} \mid Y \neq X$ do
            let $C \in \mathscr{C} \mid scp(C) = \{X, Y\}$
            foreach $b \in dom^P(Y) \mid b \notin dom^{P'}(Y)$ do
                remove $(X_a, Y_b)$ from $rel^P(C)$
                $lastModif[Y] \leftarrow cnt$
                $modified \leftarrow true$
return $modified$
```

to remark that the cumulated worst-case time complexity of line 13 of Algorithm 5 is $O(n^2d)$, and that the space complexity of the new structure $lastModif$ is $\theta(n)$. □

3.4 Algorithm sDC3

To fully exploit the incrementality of an AC algorithm such as AC3[1] when enforcing strong path consistency, we simply need to introduce the specific data structure of AC3 with respect to each value (it is then related to the approach used in [1] for the algorithm SAC-OPT). The set Q_{X_a}, which denotes this structure, represents the propagation queue dedicated to the problem $P|_{X=a}$. The principle used in sDC3 is the following: If P_1 corresponds to $AC(P|_{X=a}, \mathscr{X})$, and P_i with $i > 1$ denotes the result of the i^{th} AC enforcement wrt X_a, then P_{i+1} corresponds to $AC(P'_i, Q_{X_a})$ where P'_i is a network such that $P'_i < P_i$ and $Q_{X_a} = \{X \in \mathscr{X} \mid dom^{P'_i}(X) \subset dom^{P_i}(X)\}$. The cumulated worst-case time complexity of making these successive AC enforcements wrt X_a is clearly $O(ed^3) = O(n^2d^3)$ since AC3 is incremental and from one call to another, at least one value has been removed from one domain.

To enforce sPC on a given network P, sDC3 (see Algorithm 7) can then be called. As mentioned above, a propagation queue Q_{X_a} is associated with any value X_a. We also introduce a queue Q to contain the set of values for which a singleton check must be performed. Note that $X_a \in Q$ iff $Q_{X_a} \neq \emptyset$. Initially, AC is enforced on P, all dedicated queues are initialized with a special value denoted $\top$ (which is equivalent to $\mathscr{X}$, although considered as being different) and Q is filled up. Then, as long as Q is not empty, one value is selected from Q, a singleton check is performed with respect to this value and potentially some tuples are removed.

[1] To establish our complexity results for sDC3, we do not need to use an optimal AC algorithm.

Algorithm 7: sDC3(P = $(\mathscr{X}, \mathscr{C})$: CN)

```
$P \leftarrow AC(P, \mathscr{X})$
$Q_{X_a} \leftarrow \top, \forall X \in \mathscr{X}, \forall a \in dom(X)$
$Q \leftarrow \{X_a \mid X \in \mathscr{X} \wedge a \in dom^P(X)\}$
while $Q \neq \emptyset$ do
    pick and remove an element $X_a$ from $Q$
    $R \leftarrow checkValue(P, X_a)$
    $removeTuples(P, R)$
```

Algorithm 8: checkValue($P = (\mathscr{X}, \mathscr{C})$: CN, X_a: Value): Set of Tuples

```
$R \leftarrow \emptyset$
if $Q_{X_a} = \top$ then $P' \leftarrow AC(P|_{X=a}, \{X\})$
else $P' \leftarrow AC(FC(P|_{X=a}, X), Q_{X_a})$
$Q_{X_a} \leftarrow \emptyset$
if $P' = \bot$ then
    remove $a$ from $dom^P(X)$
    foreach $Y \in \mathscr{X} \mid Y \neq X$ do
        foreach $b \in dom^P(Y) \mid (X_a, Y_b)$ is AC do
            add $Y_b$ to $Q$ ; add $X$ to $Q_{Y_b}$
else
    foreach $Y \in \mathscr{X} \mid Y \neq X$ do
        foreach $b \in dom^P(Y) \mid b \notin dom^{P'}(Y)$ do
            add $(X_a, Y_b)$ to $R$
return $R$
```

Algorithm 9: removeTuples($P = (\mathscr{X}, \mathscr{C})$: CN, R: Set of Tuples)

```
$initsize \leftarrow |R|$
$cnt \leftarrow 0$
while $R \neq \emptyset$ do
    $cnt \leftarrow cnt + 1$
    pick and delete the first element $(X_a, Y_b)$ of $R$
    remove $(X_a, Y_b)$ from $rel^P(C)$ where $C \in \mathscr{C} \mid scp(C) = \{X, Y\}$
    if $cnt > initsize$ then
        add $X_a$ to $Q$ ; add $Y$ to $Q_{X_a}$
    add $Y_b$ to $Q$ ; add $X$ to $Q_{Y,b}$
    foreach $Z \in \mathscr{X} \mid Z \neq X \wedge Z \neq Y$ do
        foreach $c \in dom^P(Z) \mid (Z_c, X_a)$ is AC $\wedge (Z_c, Y_b)$ is AC do
            if $\nexists d \in dom^P(X) \mid (Z_c, X_d)$ is AC $\wedge (X_d, Y_b)$ is AC then
                add $(Z_c, X_a)$ to the end of $R$
            if $\nexists d \in dom^P(Y) \mid (Z_c, Y_d)$ is AC $\wedge (X_a, Y_d)$ is AC then
                add $(Z_c, Y_b)$ to the end of $R$
```

When $checkValue$ (see Algorithm 8) is called, it is possible to determine whether or not this is the first call for the given value X_a. Indeed, if $Q_{X_a} = \top$, this is the case, and then we just have to make a classical AC enforcement on $P|_{X=a}$. Otherwise, we enforce AC by using FC and the dedicated propagation queue Q_{X_a}. If a is detected as singleton arc inconsistent, it is removed and we look for any value Y_b being compatible with X_a. For each such value, we add X to Q_{Y_b} (and Y_b to Q) since the next time we will make an AC enforcement wrt Y_b, the value X_a which was present so far will have disappeared. We then need to take this into account, thanks to $X \in Q_{Y_b}$, in order to guarantee that AC is really enforced. If a is not detected as singleton arc inconsistent, we look for tuples that can be removed. Here, they are simply put in a set R.

When $removeTuples$ (see Algorithm 9) is called, each tuple (X_a, Y_b) is considered in turn in order to remove it (line 6). Then, we add Y to Q_{X_a} (and Y_b to Q): the next time we will make an AC enforcement wrt Y_b, the value X_a which was present so far (otherwise, the tuple will have already been removed) will have disappeared. Adding X to Q_{Y_b} is not necessary if the tuple (X_a, Y_b) has been put in R during the execution of $checkValue$ (by recording the initial size of the set R and using a counter, we can simply determine that). Indeed, if this is the case, it means that Y_b was removed during the last AC enforcement wrt X_a. Finally, we have to look for any value Z_c which is both compatible with X_a and Y_b. If there is no value X_d compatible with both Z_c and Y_b, it means that the tuple Z_c, X_a can be removed. Similarly, if there is no value Y_d compatible with both Z_c and X_a, it means that the tuple Z_c, Y_b can be removed.

Proposition 4. *The algorithm sDC3 enforces sPC.*

Sketch of proof. The following invariant holds: when $P' \leftarrow AC(FC(P|_{X=a}, X), Q_{X_a})$ is performed at line 3 of Algorithm 8, we have $P' = AC(P|_{X=a}, \mathscr{X})$. □

Proposition 5. *The worst-case time complexity of sDC3 is $O(n^3d^4)$ and its worst-case space complexity is $O(n^2d^2)$.*

Proof. The worst-case time complexity of sDC3 can be computed from the cumulated worst-case time complexity of $checkValue$ and the cumulated worst-case time complexity of $removeTuples$. First, it is easy to see that, in the worst-case, the number of calls to $checkValue$, for a given value X_a, is $O(nd)$. Indeed, between two calls, at least one tuple t of a relation, associated with a constraint involving X and another variable, such that $t[X] = a$ is removed. The cumulated worst-case time complexity of making the network FC, wrt X_a, is then $O(nd \times nd) = O(n^2d^2)$ whereas the cumulated worst-case time complexity of enforcing AC, wrt X_a, is $O(ed^3) = O(n^2d^3)$ if we use AC3 due to its incrementality. Lines 6 to 9 can only be executed once for X_a (it is only $O(nd)$) and the cumulated worst-case time complexity of executing lines 11 to 13, wrt X_a, is $O(n^2d^2)$. As a result, we obtain a cumulated worst-case time complexity of $checkValue$ with respect to any value in $O(n^2d^3)$ and then a cumulated worst-case time complexity of $checkValue$ in $O(n^3d^4)$ since there are $O(nd)$ values.

On the other hand, the cumulated number of turns of the main loop of $removeTuples$ is $O(n^2d^2)$ since the number of tuples in the constraint network is $O(n^2d^2)$ and since, at each turn, one tuple is removed (see line 6 of Algorithm 9). As the worst-case time complexity of one turn of the main loop of $removeTuples$ is $O(nd^2)$, we can deduce that

the cumulated worst-case time complexity of $removeTuples$ is then $O(n^3d^4)$. From those results, we deduce that the worst-case time complexity of $sDC3$ is $O(n^3d^4)$.
In terms of space, remember that representing the instance is $O(n^2d^2)$. The data structures introduced in sDC3 are the sets Q_{X_a} which are $O(n^2d)$, the set Q which is $O(nd)$ and the set R which is $O(n^2d^2)$. We then obtain $O(n^2d^2)$. $\square$

Remark that if we use an optimal AC algorithm such as AC2001, the worst-case time complexity of sDC3 remains $O(n^3d^4)$ but the worst-case space complexity becomes $O(n^3d^2)$ since the $last$ structure, in $O(n^2d)$ of AC2001 must be managed independently for each value. It can also be shared but, contrary to [1], the interest is limited here. On the other hand, one can easily adopt AC3^{rm} since the specific structure of AC3^{rm}, in $O(n^2d)$, can be naturally shared between all values.

3.5 Complexity Issues

As λ is bounded by $O(n^2d^2)$, sDC1 and sDC2 may be up to $O(n^5d^5)$. This seems to be rather high (this is the complexity of PC1), but our opinion is that, similarly to sCDC1, both algorithms quickly reach a fix-point (i.e. the number of times the function $checkVar1$ or $checkVar2$ is called for a given variable is small in practice) because inferences about inconsistent values and, especially pairs of values, can be immediately taken into account. Besides, sDC2 partially benefits from incrementality, and so is close to sDC3 which admits a nice worst-case time complexity in $O(n^3d^4)$.

On the other hand, the following proposition indicates that the time wasted to apply one of the three introduced algorithms on a network which is already sPC is quite reasonable. The three algorithms have essentially the same behaviour.

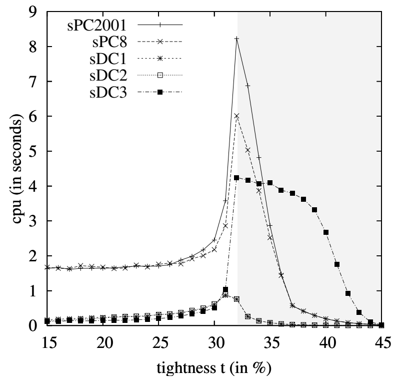
Proposition 6. *Applied to a constraint network which is sPC, the worst-case time complexity of sDC1, sDC2 and sDC3 is $O(n^3d^3)$.*

Proof. If the network is sPC, the number of singleton checks will be $O(nd)$. As a singleton check is $O(ed^2) = O(n^2d^2)$ if we use an optimal AC algorithm, we obtain $O(n^3d^3)$. $\square$

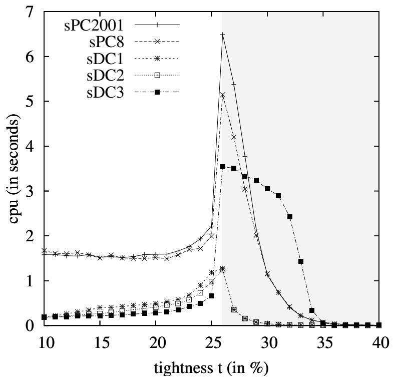
Proposition 7. *The best-case time complexity of sDC1, sDC2 and sDC3 is $O(n^2d^2)$.*

Proof. The best case is when all constraints are universal (i.e. when all tuples are allowed). Indeed, in this case, enforcing AC corresponds to calling FC since we just need to check that any value of any variable is compatible with the current assignment. A singleton check is then $O(nd)$, and the overall complexity is $O(n^2d^2)$. $\square$

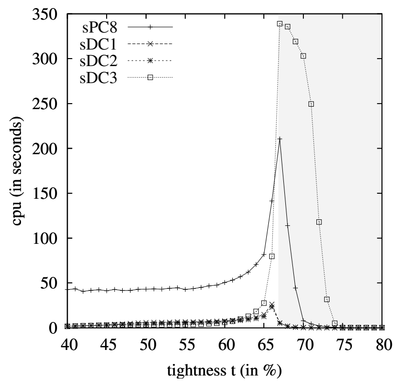
There is another interesting case to be considered. This is when after the first pass of AC (actually, FC), many revisions can be avoided by exploiting Proposition 1 of [3]. Considering a network that is sPC and assuming that all revisions can be avoided by using this *revision condition* [17], the worst-case time complexity becomes $O(nd.(nd+n^2)) = O(n^2d.max(n,d))$ as for each singleton check the number of revisions which comes after FC is $O(n^2)$, each one being $O(1)$ since the revision effort is avoided. This has to be compared with the cost of the initialization phase of PC8 and PC2001 which is $O(n^3d^2)$ in the same context. It means that one can expect here an improvement by a factor $O(min(n,d))$.



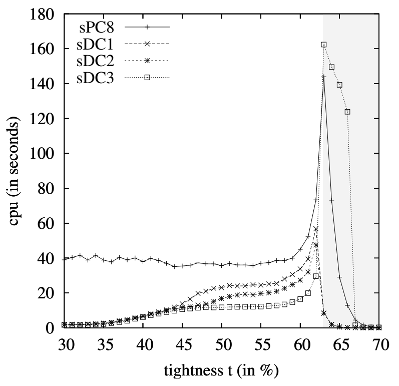
(a) $d = 10, e = 612$



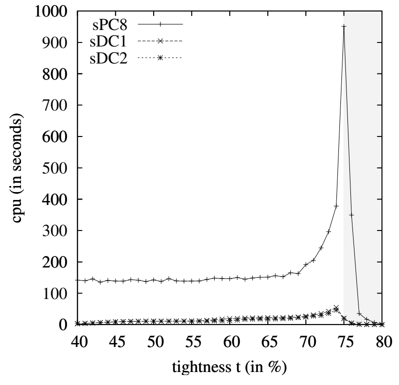
(b) $d = 10, e = 1225$



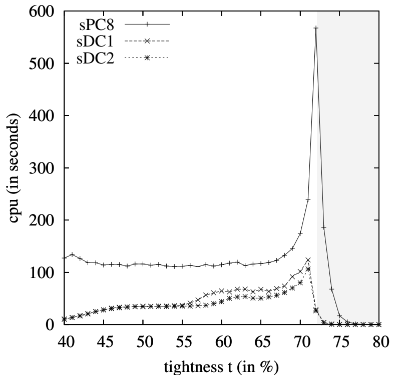
(c) $d = 50, e = 612$



(d) $d = 50, e = 1225$



(e) $d = 90, e = 612$



(f) $d = 90, e = 1225$

Fig. 1. Average results obtained for 100 random binary instances of classes $\langle 50, d, e, t \rangle$.

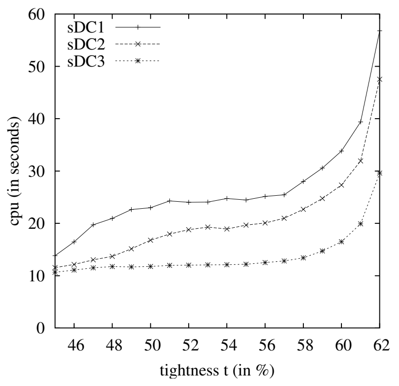


Fig. 2. Zoom on the average behaviour of sDC1, sDC2 and sDC3 below the threshold for 100 random binary instances of classes $\langle 50, 50, 1225, t \rangle$.

4 Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an experimentation on a i686 2.4GHz processor equipped with 1024 MiB RAM. We have compared the CPU time required to enforce sPC on (or show the inconsistency of) a given network with algorithms sDC1, sDC2, sDC3, sPC8 and sPC2001. The underlying Arc Consistency algorithm used for sDC algorithms was (an optimized version for binary constraints of) AC3^{rm} [13] equipped with the revision condition mechanism [3, 17].

We have first tested the different algorithms against random instances. We have collected results for classes of the form $\langle 50, d, 1225, t \rangle$ with $d \in \{10, 50, 90\}$ and t ranging from 0.01 to 0.99. We also tried classes of the form $\langle 50, d, 612, t \rangle$, that is to say random instances involving 50% of universal explicit constraints and 50% of constraints of tightness t. Figure 1 shows the average cpu time required to enforce sPC on these different classes. The shaded area on each sub-figure indicates tightnesses for which more than 50% of generated instances were proved to be inconsistent. First, we can remark that when the domain size d is set to 50 (resp. 90), sPC2001 (resp. sDC3) runs out of memory. This is the reason why they do not appear on all sub-figures. Second, when we focus our attention on the three algorithms introduced in this paper, we can make the following observations. For small tightnesses, sDC1, sDC2 and sDC3 have a similar behaviour, which can be explained by the fact that no propagation occurs. For tightnesses below the threshold (see Figure 2) , sDC3 has a better behaviour than sDC1 and sDC2 since it benefits from a full exploitation of incrementality. At and above the threshold, sDC3 is highly penalized by its fine grain, which prevents it from quickly proving inconsistency. This is particularly visible in Figure 1(d). On the other hand, the main result of this first experimentation is that sDC2, while being slightly more efficient than sDC1, is far more efficient than sPC8 (and sPC2001). For small tightnesses, there is a significant gap (up to two orders of magnitude for $d = 90$) existing between sDC2 and sPC8, which is partly due to the fact many revisions can be avoided as discussed in Sec-

$Instances$		$sPC8$	$sPC2001$	$sDC1$	$sDC2$	$sDC3$
queens-30	cpu	5.06	5.37	2.22	2.28	2.60
	mem	17	76	17	17	37
queens-50	cpu	50.9	–	4.6	4.5	5.3
	mem	30		22	22	149
queens-80	cpu	557.9	–	26.8	24.7	–
	mem	97		44	44	
queens-100	cpu	1549	–	62	58	–
	mem	197		73	73	
langford-3-16	cpu	45.45	66.66	4.91	4.44	57.8
	mem	27	612	21	21	129
langford-3-17	cpu	63.48	–	6.06	6.07	76.79
	mem	34		22	22	157
langford-3-20	cpu	140	–	11	9.7	198
	mem	43		26	26	250
langford-3-30	cpu	1247	–	60	50	–
	mem	138		56	56	

Table 1. Results obtained on academic queens and langford instances ; cpu in seconds and mem(ory) in MiB.

tion 3.5, while for tightnesses around the threshold, it is still very important (about one order of magnitude for $d = 90$). We can even observe that the gap increases when the density decreases, which is not surprising since the number of allowed tuples increases with the number of universal constraints and the fact that classical PC algorithms deal with allowed tuples.

Table 1, built from two series of academic instances, confirms the results obtained for random instances. Indeed, on such structured instances, sDC2 is about 20 times more efficient than sPC8 for large ones, whatever inference occurs or not. The $queens$ instances are already sPC, which is not the case of the $langford$ instances.

5 Conclusion

In this paper, we have introduced two algorithms to enforce strong path consistency. The algorithm sDC2 has been shown to be a good compromise between the basic sDC1 and the more refined sDC3. Even if the worst-case time complexity of sDC2 seems rather high, its close relationship with sDC3, which admits a complexity close to the optimal, suggests its practical efficiency. In practice, on random instances, sDC2 is slightly slower than sDC3 when the number of inferences is limited, but far faster at the phase transition of path consistency. Compared to sPC8 and the optimal sPC2001, sDC2 is usually around one order of magnitude faster on large instances.

Maybe, one can wonder about the interest of PC algorithms when the constraint graph is not complete. Indeed, when a pair of values is identified as not being PC, it has to be removed from the network. When no constraint binding the two involved variables exists in the CN, a new one has to be inserted (consequently, changing the constraint graph). To avoid this drawback, it is possible to enforce relation filtering consistencies in a conservative way, i.e. without adding new constraints, This gives rise to consistencies such as CDC and CPC. Of course, some pairs of values identified as inconsistent must be ignored, and consequently, some information is lost. However, there exists an alternative to inserting new constraints: recording nogoods, especially as this approach [8, 20] has been recently re-investigated by the CSP community [11, 4, 19, 14]. As a perspective of this work, we project to enforce strong path consistency by combining nogood recording with an adaptation of sDC2. Interestingly, it could be applied to any constraint network (even one involving non binary constraints).

References

1. C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
2. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004.
4. K. Boutaleb, P. Jégou, and C. Terrioux. (no)good recording and robdds for solving structured (v)csps. In *Proceedings of ICTAI'06*, pages 297–304, 2006.
5. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.
6. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
7. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
8. D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.
9. C.C. Han and C.H. Lee. Comments on Mohr and Henderson's path consistency. *Artificial Intelligence*, 36:125–130, 1988.
10. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
11. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.
12. C. Lecoutre, S. Cardon, and J. Vion. Conservative dual consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.
13. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.
14. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Nogood recording from restarts. In *Proceedings of IJCAI'07*, pages 131–136, 2007.
15. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
16. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
17. D. Mehta and M.R.C. van Dongen. Reducing checks and revisions in coarse-grained MAC algorithms. In *Proceedings of IJCAI'05*, pages 236–241, 2005.
18. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
19. G. Richaud, H. Cambazard, B. O'Sullivan, and N. Jussien. Automata for nogood recording in constraint satisfaction problems. In *Proceedings of SAT/CP workshop held with CP'06*, 2006.
20. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
21. M. Singh. Path consistency revisited. *International Journal on Artificial Intelligence Tools*, 5:127–141, 1996.