Dual Consistency for Non-binary Constraint Networks

EMN Research Report nb. 09/04/INFO

Julien Vion
julien.vion@emn.fr

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.

Abstract. Dual Consistency (DC) was introduced by Lecoutre, Cardon and Vion in [10, 11]. DC is a novel way of handling Path Consistency (PC), with a simpler definition, and new efficient algorithms and approximations. Interestingly, the new definition may be extended to non-binary constraint networks (CNs). DC is thus a way to generalize PC to any CN, while keeping the initial non-binary constraints of the CN, and their associated propagators, untouched. DC can also be seen as a simple and efficient way to generate automatically implicit binary constraints. This article presents the implications of this generalization in terms of complexity. Preliminary experimental results shows the potential effectiveness of dynamic implicit constraints generation, as well as identifying its weaknesses. Prospective ideas of approximations, whose purpose is to handle these weaknesses in practice, are then proposed.

Consistencies are properties of Constraint Networks (CN) that can be used to identify and prune, usually in polynomial time and space, values or instantiations that cannot be part of the solutions of the Constraint Satisfaction Problem (CSP) associated with the CN. They are essential in the process of solving a CSP and one main reason of the success of Constraint Programming [8, 2].

The most useful consistency is (Generalized) Arc Consistency (GAC), which is used to detect values that are inconsistent according to a given constraint. Path Consistency (PC) is one of the oldest available consistencies. It is used to identify inconsistent *pairs of values*. PC is defined on binary CNs, and usually applied to complete binary constraint graphs [14]. Dual Consistency (DC) is a simpler, alternative (equivalent) definition of PC. It was used in [10] to define a new approximation of PC, called Conservative DC, and in [11] to introduce new efficient algorithms to establish PC on complete binary CNs.

In this paper, we propose to extend the definition of (C)DC to non-binary CNs. Interestingly, DC can be seen as an easy way to automatically deduce implied binary constraints from the original domains and constraints of the CN. Implied constraints are constraints that can be added to the CN without changing the set of solutions, but that can be used to improve the propagation capability.

After a few definitions and notations, we describe properties of coarse-grained GAC algorithms, and we present Dual Consistency. Then we discuss about applying Dual Consistency to non-binary networks featuring semantics-based propagators. We propose sDC^{clone}, an algorithm to enforce Dual Consistency on general CNs by generating implied constraints "on the fly", that fully exploits the incrementality of underlying GAC algorithms to obtain a worst-case time complexity in $O(ne_id^3 + nd\psi)$ ($O(n(e+e_i)d^3)$ in the case of binary CNs), which is better than previously stated algorithms.[1] sDC-2.1, an optimized and generalized variant of sDC-2 is then proposed, and its worst-case time complexity is thoroughly studied. Finally, we identify experimentally some drawbacks of DC that can prevent its use on real-world problems. As a perspective of this work, we propose possible approximations of DC to handle its identified weaknesses.

1 Background

1.1 CN and CSP

A *constraint network* (CN) P consists of a set $\mathscr{X}$ of n variables and a set $\mathscr{C}$ of e constraints. A domain, associated to each variable X and denoted $\mathrm{dom}^P(X)$, is a finite set of at most d values the variable can take in the CN P. When possible without ambiguity, $\mathrm{dom}^P(X)$ will be simply denoted $\mathrm{dom}(X)$. The constraints specify the allowed combinations of values for given subsets of variables. An instantiation I is a set of variable/value couples, (X, v), denoted X_v, where v is a value from a given universe U ($\forall X \in \mathscr{X}, \mathrm{dom}(X) \subseteq U$). I is *valid* w.r.t. a CN P iff for any variable X involved in I, $v \in \mathrm{dom}^P(X)$.

A *relation* R of arity less or equal to k is a set of instantiations.[2] A *constraint* C of arity r is a pair $(\mathrm{scp}(C), \mathrm{rel}(C))$, where $\mathrm{scp}(C)$ is a set of r variables and $\mathrm{rel}(C)$ is a relation of arity r. Given a constraint C, an instantiation I of $\mathrm{scp}(C)$ (or of a superset of $\mathrm{scp}(C)$, considering only the variables in $\mathrm{scp}(C)$), *satisfies* C iff $I \in \mathrm{rel}(C)$. We say that I is *allowed* by C. Controlling whether an instantiation satisfies a constraint is called a *constraint check* (or *check* for short). An instantiation I is *locally consistent* iff it is valid, and allowed by all constraints of the CN (it does not falsify any constraint of P). A *solution* of a CN $P(\mathscr{X}, \mathscr{C})$ is a locally consistent instantiation of all variables in $\mathscr{X}$.

A CSP is the problem of deciding whether a solution to a given CN exists. An instantiation is *globally consistent* iff it is a subset of at least one solution. Instantiations that are not globally consistent are also called *no-goods*. Determining whether a locally consistent instantiation is a no-good is NP-complete in the general case, but consistencies are used to identify some no-goods using polynomial algorithms. Given a consistency Φ, it can be *enforced* on P using a "Φ-consistency algorithm" whose purpose is to detect and remove all Φ-inconsistent instantiations from P until a fix-point is reached (a *closure* of P by Φ is obtained).

[1] ψ is the amortized complexity to enforce GAC on the CN incrementally, and e_i is the number of generated implied binary constraints.

[2] k is the maximum arity of the constraints in a given CN.

Algorithm 1: GAC($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{A}$)

```
   P: the CN to filter
   A: an initial set of arcs to revise
1  Q ← A
2  while Q ≠ ∅ do
3  |  pick (C, X) from Q
4  |  if revise(C, X) then
5  |  |  if dom(X) = ∅ then return ⊥
6  |  |  Q ← Q ∪ mod({X}, C)\(C, X)
7  return P
```

Most often, the fix-point is unique. The CN obtained from P by enforcing the consistency Φ will be denoted $\Phi(P)$.

1.2 Generalized Arc Consistency

Generalized Arc Consistency (GAC) is the most common and useful consistency. It is a *domain consistency* [2], i.e. it identifies no-goods of size 1 (globally inconsistent values). Consistencies that identify larger no-goods are usually called *relational* consistencies, since they are used to remove allowed instantiations from the relations of the constraints.

Definition 1 (Generalized Arc Consistency). *Given a CN $P = (\mathscr{X}, \mathscr{C})$:*

1. *$X_a \mid X \in \mathscr{X}$ and $a \in \mathrm{dom}(X)$ is GAC w.r.t. a constraint $C \in \mathscr{C}$ iff $\exists I \mid X_a \in I \wedge I$ is valid and allowed by C. I is then called a support of a in C.*
2. *X_a is GAC iff $\forall C \in \mathscr{C} \mid X \in \mathrm{scp}(C)$, X_a is GAC w.r.t. C.*
3. *P is GAC iff $\forall X \in \mathscr{X}$, $\forall a \in \mathrm{dom}(X)$, X_a is GAC.*

Notation 1 *The set $\{(C, Y) \mid \{X, Y\} \subseteq \mathrm{scp}(C) \wedge X \in \mathscr{X} \wedge C \in \mathscr{C}\}$ of arcs to be revised after the modification of the set of variables $\mathscr{X}$ or of the set of constraints $\mathscr{C}$ will be denoted $\mathrm{mod}(\mathscr{X}, \mathscr{C})$.*

Algorithm 1 presents the main loop of coarse-grained GAC algorithms, i.e. variants of GAC-3. The variants reside in the nature of the `revise` function called on Line 4 of the algorithm. This version is arc-oriented: the propagation queue contains all arcs that need to be revised. An arc is a pair (C, X) with $X \in \mathrm{scp}(C)$. In any CN, $O(ek)$ arcs can be devised. An arc (C, X) must be revised if there is a possibility for X not to be AC w.r.t. C. If we have no clue on the state of the CN, all the possible arcs must be inserted in the initial queue and thus revised at least once. However, if we know that a single variable X has been modified in a GAC CN, only arcs $\mathrm{mod}(\{X\}, \mathscr{C})$ need to be inserted. Algorithm 1 can be initialized from a set of arcs $\mathscr{A}$ (Line 1). Using Notation 1, GAC can be enforced on a given CN $P = (\mathscr{X}, \mathscr{C})$ by calling $P \leftarrow \mathrm{GAC}(P, \mathrm{mod}(\mathscr{X}, \mathscr{C}))$.

The worst-case complexities of these algorithms rely on the fact that a given arc (C, X) is inserted in the propagation queue Q only when one of the $O(k)$ variables of $\text{scp}(C)$ is modified, and a variable can be modified only d times. Thus, $O(ek^2d)$ calls to `revise` are performed in the worst case. The basic algorithm for `revise` iterates over the Cartesian product of the domains of the variables in $\text{scp}(C)$ until it finds one support for each value of the domains, hence a complexity in $O(kd^k)$ for `revise` (assuming a constraint check is in $O(k)$) and $O(ek^3d^{k+1})$ for establishing GAC. The idea beneath GAC-2001 [5] is to make the `revise` function incremental, so that the amortized complexity of all calls to `revise` for a given constraint is in $O(k^2d^k)$.[3] Hence the complexity of GAC-2001 is in $O(ek^2d^k)$. GAC-schema [4] reaches the $O(ekd^k)$ optimal complexity by making use of *fine-grained* propagation queues and exploiting the multi-directionality of the constraints. However, this requires many additional data structures, and the obtained algorithms are not so much more efficient in practice. Other works have stated that another suboptimal variants of GAC-3, such as GAC-3^{rm} [12] or GAC-watched [9] are most efficient in practice when maintained during search, due to the presence of backtrack-stable data structures. Note that in the binary case ($k = 2$), (G)AC-2001 has an optimal time complexity in $O(ed^2)$. Finally, in the binary case, we can use highly optimized algorithms such as AC-3^{bit} [13] to make the computation of AC as fast as possible: the propagation of many extensional binary constraints is thus reasonable.

All the GAC algorithms variants are incremental: once the fix-point is reached for a given CN, the amortized worst-case complexity for multiple calls to the algorithm, with at least one value removed between each call, is the same as the complexity of a single call. In order to apply the GAC-3 algorithm incrementally, one has to make sure that an arc will be revised if and only if a value has been removed from the variable.

1.3 Propagators

The idea of *propagators* comes from the AC-5 propagation algorithm [19]. The generic GAC algorithms described in the previous section are exponential in the arity of the constraints, and are impractical when k grows. However, in practice, GAC for non-binary constraints can often be computed in polynomial time by exploiting the semantic properties of the constraints [1, 16]. AC-5 enables this by *abstracting* the `revise` function, which can be specialized for each type of constraint. These specialized `revise` functions are usually called *propagators*. In this context, general GAC algorithms are most often used for constraints defined in *extension*, i.e. an exhaustive list of forbidden instantiations,[4] or for constraints defined in *intention* and involving scalar operations.

Most works on generic solving of CSP have focused on the use of homogeneous constraints, most often binary constraints in extension. In practice, real-world

[3] The additional k factor is due to the fact that the multi-directionality of the relations cannot be exploited and each possible instantiation may be checked up to k times.

[4] In the case of a list of *authorized* instantiations, other algorithms such as Simple Tabular Reduction [17] can apply.

problems involve heterogeneous constraints, associated with semantics-based, efficient propagators. In this paper, we make no hypothesis with respect to the initial network and algorithms used to enforce GAC. Any constraint with a specific propagator is kept, and the semantic of the constraint is exploited.

In the following, we will denote by $O(\phi)$ the worst-case time complexity to establish GAC on a given constraint network, $O(\psi)$ the amortized worst-case time complexity to establish GAC *incrementally* on the same CN, removing at least one value between each propagation, and $O(\rho)$ the global space complexity of all propagators involved in the CN. On a general CN using GAC-schema as a propagation scheme, $O(\phi) = O(\psi) = O(ekd^k)$ and $O(\rho) = O(nd + ekd)$.

1.4 Dual Consistency

Dual Consistency (DC), introduced in [10], is an alternative (equivalent) definition of Path Consistency [14]:

Definition 2 (Dual Consistency). *Given a CN $P = (\mathscr{X}, \mathscr{C})$, a binary instantiation $I = \{X_a, Y_b\}$ is DC iff $X_a \in \mathrm{AC}(P|_{Y=b}) \wedge Y_b \in \mathrm{AC}(P|_{X=a})$.*
P is DC iff all locally consistent binary instantiations of P are DC.
P is strongly DC (sDC) iff it is both AC and DC.

The most efficient known algorithm to enforce Strong Dual Consistency is sDC-2, described in [11]. Lecoutre *et al.* devised that establishing sDC with this algorithm has a worst-case time complexity in $O(n^5d^5)$, well above the best complexity of PC algorithms, which is $O(n^3d^3)$. However, the worst case of sDC-2 is very unlikely to appear and the algorithm is in practice faster than state-of-the-art PC algorithms on most instances of CSP. Moreover, sDC-2 only uses a very lightweight additional data structure (in $O(n)$).

2 Dual Consistency and non-binary networks: Algorithms and Complexity Issues

Existing algorithms for PC require that the CN is a complete binary graph of constraints supporting composition. However, the definition of DC leads to the following remarks:

1. Simply by replacing AC with GAC in Definition 2, DC may be applied on CNs of any arity.
2. DC algorithms do not have any assumption on the way (G)AC is established. The algorithms thus can use all state-of-the-art propagators (for constraints in extension or semantics-based) featured by the solver.
3. In order to establish sDC, one "only" needs to store binary no-goods (even if the original CN contains non-binary constraints). This does not necessarily involve the completion of the binary constraint graph (only in the worst case).[5]

[5] Note that this is also true for PC algorithms, but has not been experimented in previous works as far as we know.

Algorithm 2: sDC-1($P = (\mathscr{X}, \mathscr{C})$)

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  mark ← X ← first(𝒳)
3  repeat
4  |  if |dom(X)| > 1 ∧ checkVar-1(P, X) then
5  |  |  P ← GAC(P, mod({X}, 𝒞))
6  |  |  if P = ⊥ then return ⊥
7  |  |  mark ← X
8  |  X ← next(𝒳, X)
9  until X = mark
10 return P
```

The latest proposition rises from a new point of view on DC (and PC): the additional constraints that are introduced in order to enforce DC are constraints that are *implied* by the original constraints of the problem. sDC allows us to deduce these constraints purely semantically by executing the propagators of the original constraints.

Three different algorithms, sDC-1, sDC-2 and sDC-3, dedicated to enforce Dual Consistency on binary networks are proposed in [11]. We propose to extend sDC-1 and sDC-2 in order to handle non-binary constraint networks (sDC-3 is targeted towards binary CNs and has been experimentally showed not to be the most efficient algorithm in practice). We try to benefit from the incrementality of propagation algorithms to obtain a better high bound of time complexity. We also get inspiration in the SAC-OPT algorithm [3]. sDC (and SAC) algorithms rely on the idea of performing *singleton tests*: for each value X_v of a CN P, v is assigned to X ($\mathrm{dom}(X)$ is reduced to the *singleton* $\{v\}$), and (G)AC is enforced on the resulting CN, denoted $P|_{X=v}$. SAC removes the value from the domain of the variable iff an inconsistency is detected. DC goes further by translating all information that can be deduced from the singleton test into binary no-goods that are added to the CN by modifying and/or adding implied constraints. The subset of added implied constraints will be denoted $\mathscr{C}_i$ ($\mathscr{C}_i \subseteq \mathscr{C}$). All binary extensional constraints from the original CN (or that can be safely converted to extensional constraints) can be immediately put into $\mathscr{C}_i$. We will denote $e_i = |\mathscr{C}_i|$. We have $e_i \leq \binom{n}{2} \in O(n^2)$. According to the proof in [3] on the globality of supports for SAC, if any modification is performed during a singleton test, all singletons (involving a different variable) must be tested again.

2.1 sDC-1: the naive approach.

sDC-1, already described in [11], is the most "naive" algorithm for enforcing sDC. Algorithms 2 and 3 depict sDC-1. The algorithm iterates over all values in the domains of the variables. The *mark* and `first`/`next` functions in Algorithm 2 are used to iterate over all variables of the CN in turn, until all variables have been checked by `checkVar-1` without any change. `next`($\mathscr{X}, X$) considers $\mathscr{X}$ to

Algorithm 3: checkVar-1($P = (\mathscr{X}, \mathscr{C})$, X)

```
1  modif ← false
2  foreach a ∈ dom(X) do
3      P' ← GAC(P|_{X=a}, mod({X}, C))
4      if P' = ⊥ then
5          remove a from dom^P(X)
6          modif ← true
7      else
8          foreach Y ∈ X \ X do
9              let C be s.t. C ∈ C_i ∧ scp(C) = {X, Y}
10             foreach b ∈ dom^P(Y) | b ∉ dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) do
11                 remove {X_a, Y_b} from rel(C)
12                 modif ← true
13 return modif
```

be ordered, and returns either the first variable right after X in $\mathscr{X}$, or the first variable of $\mathscr{X}$ (`first`($\mathscr{X}$)) iff X is the last element of $\mathscr{X}$. Singleton tests are performed in the main **foreach do** loop of `checkVar-1` (Algorithm 3). The second loop on Lines 8-12 stores the no-goods that can be deduced from the enforcement of GAC on Line 3. On Line 9, the constraint is created "on the fly" iff it does not exist. Created constraints are initially universal constraints that allow all instantiations of the variables.

Proposition 1. *Applied on a general CN, sDC-1 has a worst-case time complexity in* $O(ne_i^2d^5 + ne_id^3\phi)$ *and a space complexity in* $O(e_id^2 + \rho)$.

Proof (Sketch). $O(nd)$ singletons must be tested, and if any singleton test leads to a modification, all singletons must be checked again [3]. The smallest change that can occur is the removal of a binary no-good from an implied constraint. There can be $O(e_id^2)$ such changes. One singleton test consists in:

1. Establishing GAC in $O(e_id^2 + \phi)$ on the CN (Line 3 of Algorithm 3). The $O(e_id^2)$ term corresponds to the propagation of the e_i additional implied binary constraints using some optimal AC algorithm.
2. Finding changes in the CN in $O(nd)$ and removing the corresponding no-goods from the implied constraints (Lines 8-12 of Algorithm 3). This point is incremental if the deltas of the domains after a propagation can be obtained (as suggested by the AC-5 propagation scheme) and is amortized to a total in $O(e_id^2)$, that can be safely discarded.

The only data structures sDC-1 uses are for storing the no-goods in extensional binary constraints, which is in $O(e_id^2)$. The data structures of the optimal AC algorithms that are used to propagate these constraints are in $O(e_id)$ and can be discarded. □

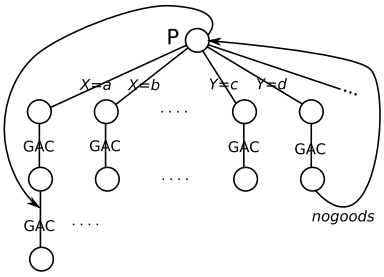


Fig. 1. Illustrating the incrementality of GAC over singleton tests

Experimental results in [11] state that despite its simplicity and high theoretical worst-case complexity, this algorithm has a very good practical behavior, often better than state-of-the-art PC algorithms.

2.2 sDC^{clone}: enforcing sDC in $O(ne_id^3 + nd\psi)$.

This complexity for enforcing DC is obtained by exploiting the full strength of the incrementality of (G)AC algorithms. As illustrated by Figure 1, if some no-goods are deducted for the singleton test $Y = d$, the previous singleton test $X = a$ should not be recomputed from scratch, but restarted incrementally from the latest fix-point obtained for $X = a$. The idea for sDC^{clone}, borrowed to the SAC-OPT algorithm, is to achieve this by creating a physical copy of the CN in memory for each one of the $O(nd)$ possible singleton tests. In order to benefit from the incrementality of propagation algorithms, all the data structures of these algorithms must be duplicated as well.

Finally, one should note that the implied constraints must be propagated *every time a constraint is modified* (and not only when a value is removed from the domain of a variable). Looking at Algorithm 1, it means that each arc implying an implicit constraint can be added $O(d^2)$ times, hence $O(e_id^2)$ calls to `revise` may be made. Even though the `revise` procedure for AC-2001 is incremental, it requires at least $O(d)$ operations to validate the current supports. Thus, AC-2001's complexity falls to $O(e_id^3)$ and cannot be used to obtain the optimal incremental complexity for propagating the implied constraints, and one should rely on a *fine-grained* algorithm such as AC-4, AC-6 or AC-7.

Proposition 2. *Applied on a general CN, sDC^{clone} has a worst-case time complexity in $O(e_ind^3 + nd\psi)$ and a space complexity in $O(e_ind^2 + nd\rho)$.*

Proof (Sketch). The worst-case time complexity is obtained by considering the full incrementality of GAC algorithms for each of the $O(nd)$ singleton tests, assuming an $O(\psi)$ amortized complexity for enforcing GAC incrementally on the CN, and $O(e_id^2)$ for enforcing AC on the implied constraints, using an optimal, fine-grained AC algorithm.

The CN must be cloned $O(nd)$ times. However, the list of no-goods (i.e. the relations of the implied constraints) can be shared between all clones. Only the data structures of the optimal AC algorithm (in $O(e_id)$) are cloned. □

Algorithm 4: sDC-2.1($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{C}_i$)

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  mark ← X ← first(𝒳)
3  repeat
4      if |dom(X)| > 1 ∧ checkVar-2.1(P, 𝒞_i, X) then
5          P' ← GAC(P, {arcQueue[i] | firstArcMain ≤ i < cnt})
6          if P' = ⊥ then return ⊥
7          foreach (Y, b) | Y ∈ 𝒳 ∧ b ∈ dom^P(Y) ∧ b ∉ dom^P'(Y) do
8              remove b from dom^P(Y)
9              foreach (C, Z) ∈ mod({Y}, 𝒞) do arcQueue[cnt++] ← (C, Z)
10         firstArcMain ← cnt
11         mark ← X
12     X ← next(𝒳, X)
13 until X = mark
14 return P
```

For example, on a CN consisting only in all-different constraints ($O(\psi) = O(ek^2d^2)$ and $O(\rho) = O(ekd)$ [16]), sDC can be enforced using this algorithm in $O(e_i nd^3 + enk^2d^3)$ with an $O(e_i nd^2 + enkd^2)$ space complexity. In the binary case, sDC^{clone} reaches a worst-case time complexity in $O(n(e + e_i)d^3)$, which is less than previously stated PC algorithms.Still, the space requirements are excessive for practical use. Moreover, Debruyne & Bessière state in [3] that the results of the experimentations conducted with the SAC-OPT algorithm, which rely on the same principle, were quite disappointing. This algorithm is thus not further described nor experimented in this article.

2.3 sDC-2.1: a compromise.

"Restoring the state of the CN", which is essential in catching the enhanced complexities of sDC^{clone}/SAC-OPT, can be done in a very natural way, without any additional data structure, when establishing DC. The information recorded in constraints during the previous test of the same singleton can be exploited to restore the domains of the variables to the state they were at the end of the previous test. This can be performed by calling the `revise` function of implied constraints that involve the current singleton X_a: **foreach** $(C, Y) \in \text{mod}(\{X\}, \mathscr{C}_i)$ **do** `revise`(C, Y). This process is related to Forward Checking and has a worst-case time complexity in $O(nd)$, as there may be at most n binary constraints involving a given variable, and the call to `revise` on a binary constraint whose one variable is a singleton is in $O(d)$.

This observation was already partly used to design the sDC-2 algorithm. We present here sDC-2.1, an optimized and extended version of the sDC-2 algorithm that handles non-binary constraints specifically. It is described on Algorithms 4 and 5. The *lastModified* data structure of sDC-2 is replaced by a new *arcQueue* data structure and $O(nd)$ pointers (one for each singleton X_a)

Algorithm 5: checkVar-2.1($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{C}_i$, X)

```
1  modif ← false
2  foreach a ∈ dom(X) do
3      if cnt ≤ |𝒳| then
           /* First turn                                                    */
4          P' ← GAC(P|_{X=a}, mod({X}, 𝒞))
5      else
           /* Consequent turns: forward checking                           */
6          P' ← P|_{X=a}
7          foreach (C', Y') ∈ mod({X'}, 𝒞_i') do revise(C', Y')
           /* Propagation with pre-initialized propagation queue           */
8          P' ← GAC(P', {arcQueue[i] | firstArc[X_a] ≤ i < cnt}))
9      if P' = ⊥ then
10         remove a from dom^P(X)
11         foreach (C, X) ∈ mod({X}, 𝒞) do arcQueue[cnt++] ← (C, X)
12         modif ← true
13     else
14         foreach Y | {X, Y} ⊆ 𝒳 do
15             let C be s.t. C ∈ 𝒞_i ∧ scp(C) = {X, Y}
16             foreach b ∈ dom^P(Y) | b ∉ dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) do
17                 remove {X_a, Y_b} from rel(C)
18                 arcQueue[cnt++] ← (C, X) ; arcQueue[cnt++] ← (C, Y)
19                 modif ← true
20 firstArc[X_a] ← cnt
21 return modif
```

denoted $firstArc[X_a]$. This data structure works as follows: *arcQueue* holds all arcs that must be revised in all singletons tests. An arc is inserted at the end of the queue whenever a neighbor variable or constraint is modified. Every time an arc is revised for the singleton test X_a, $firstArc[X_a]$ is moved to the next element in *arcQueue*. When GAC is called, the revision queue is initialized with all arcs of *arcQueue* starting from $firstArc[X_a]$. This mechanism ensures that adding an arc to the propagation queue of all singletons is done in constant time, and that the initializations of the revision queues are incremental.

These data structures are used on Line 5 of Algorithm 4 and Line 8 of Algorithm 5 to initialize the propagation queue of the underlying propagation algorithm, so as to make sure that only the propagators that are likely to make some inference will be called (i.e. a variable involved by the constraint or the constraint itself has been modified since the last occurrence of the singleton test). X', Y', C' and C_i' are the variables, constraint or set of constraints in P' that respectively correspond to X, Y, C and C_i in P.

Proposition 3. *Applied on a general CN, sDC-2.1 has a worst-case time complexity in* $O(e_i^2 d^4 + e_i n d^5 + nkd^2\phi)$ *and a space complexity in* $O(e_i d^2 + ek^2 d + \rho)$.

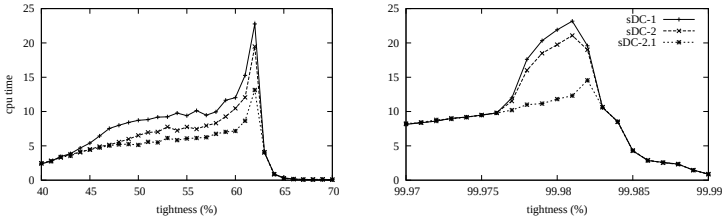
Proof (Sketch). In this variant of the algorithm, a singleton test consists in:

1. The restoration of the state of the CN at the end of the previous singleton test of the same (Variable, Value) pair, in $O(nd)$ by using Forward Checking. The Forward Checking over all singletons is amortized to $O(e_i d^2)$, and this is performed $O(e_i d^2)$ times in the worst case (as this is the number of possible modifications), hence a first term in $O(e_i^2 d^4)$.
2. Initializing the propagation queue of GAC. In the worst case, each one of the $O(e_i + ek)$ arcs is handled once for each possible modification in each singleton test ($O(d^2)$ for the arcs implying the implied constraints and $O(kd)$ for the others), hence an amortized complexity in $O(nd.(e_i d^2 + ek^2 d))$.
3. The propagation of the singleton assignment or of the updates. Since the data structures are not cloned, the incrementality of the propagators is lost and falls back to $O(e_i d^2 + \phi)$. We know that a given propagator can be called only $O(kd)$ times on a given singleton (resp. $O(d^2)$ for implied constraints), when a value (resp. a no-good) is removed. Thus, for each of the $O(nd)$ singletons, the amortized complexity for all propagations of a singleton test is in $O(e_i d^4 + kd\phi)$.
4. If an inconsistency is detected, removing the value and updating the *arcQueue* data structure (Line 11 of Algorithm 5). Detecting an inconsistency on Line 9 can only occur $O(nd)$ times and requires $O(n)$ operations to update *arcQueue*, which are amortized to a total of $O(e_i d + ed)$. This part can be safely discarded in the final complexity, assuming $e \leq \phi$.
5. Else, updating the implied constraints and *arcQueue*. As for sDC-1, this is incremental if deltas are exploited and can be discarded.

For the space complexity: the relations of the implied constraints are in $O(e_i d^2)$. The $O(e_i)$ arcs corresponding to the implied constraint can be added $O(d^2)$ times in the revision queue, and the $O(ek)$ arcs corresponding to the initial constraints can be added $O(kd)$ times. The *arcQueue* data structure has thus a space complexity in $O(e_i d^2 + ek^2 d)$ and the $firstArc$ pointers in $O(nd)$. This last term can be discarded as we can safely assume that $\rho > nd$. □

Note that applying this algorithm on a complete binary network ($O(\phi) = O(n^2 d^2)$ and $O(e_i) = O(n^2)$) results in a worst-case complexity in $O(n^4 d^4 + n^3 d^5)$, which is better than sDC-2 (in $O(n^5 d^5)$). If we have for example an initial CN composed exclusively of all-different constraints (e.g. for modelling Latin Square problems), which can be propagated in $O(k^{1.5} d^2)$ [16], sDC can be computed in $O(e_i^2 d^4 + e_i n d^5 + enk^{2.5} d^4)$.

Implementation notes. In practice, the *arcQueue* data structure can be safely replaced by two integer tables $lastModVar$ and $lastModCons$, closer to the original sDC-2 algorithm. Their management is not incremental and add a theoretical $O(n^4 d^3 + e_i n^3 d^3)$ term to the worst-case complexity of the algorithm, due to the initialization process of the propagation queues. However, the space complexity is smaller ($O(n+e_i)$ instead of $O(nd+e_i d^2+ekd)$), and their management is faster in practice (as it naturally regroups revisions on the same arcs).



$(50, 50, 1225, 2)$ $(20, 6, 50, 8)$

Fig. 2. Preprocessing on random instances of various tightnesses. CPU times are in seconds. Characteristics of the problems are given as a quadruple (n, d, e, k).

The arc-oriented GAC algorithm presented on Algorithm 1 is emulated by an enhanced variable-oriented version, using revision ordering heuristics and special counters to avoid useless revisions, as described in [6, 7].

3 Experiments

Experiments are performed on benchmarks from the Third International Solver Competition [18], featuring either binary or non-binary CN involving either constraints in intention, extension or with semantics-based propagators. We used the Concrete solver, built on the CSP4J library [20].

Figure 2 gives an idea of the enhanced performance of sDC-2.1 w.r.t. sDC-1 and sDC-2, even on binary problems. sDC is enforced on binary (leftmost graph) and non-binary (rightmost graph) random problem instances with each algorithm. The difference between the algorithms only appears near the threshold, where singletons must be processed multiple times to reach the fix-point.

Two strategies are compared: problems are solved using standard MGAC-*dom/wdeg* after a phase of preprocessing where GAC or DC are respectively established. The data structures and propagation algorithms used to propagate the binary extensional constraints are those of AC-3^{bit} or AC-3^{bit+rm} [13].

Table 1 shows representative results where applying sDC has a positive impact on search. sDC is mostly efficient on very hard instances, where the reduced number of explored nodes can compensate the time taken during the preprocessing and propagating the additional constraints during search (this is estimated by the *nodes per second* metric). The selected problems are all structured problems, with various constraints. *taillard* instances are scheduling instances, modelled with *disjunctive inequality constraints* with a specific incremental linear-time propagator. *tsp* is a Travelling Salesman Problem and *series* is an All-Interval-Series problem with ternary positive table constraints propagated with STR algorithm [17]. *scen11-f1* is the hardest available RLFAP instance, and uses only binary constraints propagated with AC-3^{bit}. These binary instances demonstrate that on real-world problems, completing the constraint graph is not mandatory to enforce PC (*scen11-f1* uses 680 variables and completing the graph would require more than 230,000 constraints).

		GAC	sDC	GAC	sDC
		tsp-25-715		*scen11-f1*	
		(76, 1 001, 350, 3)		(680, 42, 4 103, 2)	
prepro	cpu	0	353	0	41
	add cstr	0	2,303	0	757
search	cpu	497	**23**	9,646	**9,408**
	nodes	397k	**2k**	6,749 k	**4,335 k**
	nodes/s	**798**	96	**700**	461
total	cpu	497	**376**	9,646	**9,448**
		series-15		*os-taillard-7-100-0*	
		(29, 15, 210, 3)		(49, 434, 294, 2)	
prepro	cpu	0	1	0	27
	add cstr	0	182	0	294
search	cpu	475	**238**	> 600	**104**
	nodes	6,920 k	**1,890 k**	$>$ 2,790 k	**148 k**
	nodes/s	**15,142**	7,908	**4,650**	1,423
total	cpu	475	**239**	> 600	**131**

Table 1. Comparing search performance with and without sDC preprocessing. Positive representative instances.

		GAC	sDC	GAC	sDC	GAC	sDC
		os-taillard-7-95-7		*os-gp-10-10-1092*		*graceful-K5-P2*	
		(49, 403, 294, 2)		(101, 1 090, 1 000, 2)		(35, 26, 370, 3)	
prepro	cpu	0	25	0	> 275	0	5
	add cstr	0	303	0	$> 1{,}020$	0	200
search	cpu	**675**	1,654	$> 2{,}400$	–	**157**	246
	assgn	2 749 k	**1,698 k**	$>$ 864 k	–	**985 k**	993 k
	nodes/s	**4 073**	1,026	360	–	**6,280**	4,040
total	cpu	**675**	1,679	$> 2\,400$	–	**157**	251

Table 2. Negative representative instances.

There are many cases where the new constraints interfere with the variable ordering heuristics: indeed, the best generic variable ordering heuristics such as *dom/wdeg* rely on the structure of the CN to select the variables to assign. The *tsp-25-715* instance is spectacularly influenced by this phenomenon.

Table 2 shows examples of instances for which applying sDC is counter-productive. Three main drawbacks are identified, and a representative instance was chosen in each case: in the case of *os-taillard-7-95-7*, the reduced number of nodes does not compensate the time lost propagating the new constraints. In the case of *os-gp-10-10*, a very large number of implied constraints is generated (up to 5000), and the size of the instance just too high for these constraints to fit in memory (with a 600 MiB memory limit). Finally, in the case of *graceful* instances, *dom/wdeg* is actually misguided by the new constraints. In our experience, degraded cases are rare, and search strategies specific to the problems would of course not be affected.

	GAC	sDC/RC
	os-gp-10-10-1092	
prepro	cpu	799
	add cstr	851
search	cpu	599
	nodes	321k
	nodes/s	536
total	cpu	1,398

Table 3. Solving a problem by generating implied row-convex constraints

4 Perspectives

The main usefulness of Dual Consistency is to automatically improve the robustness of the solver in front of poorly modelled problems. Indeed, some of the implied constraints introduced by DC could have been manually inserted during the modelling phase, with an appropriate propagation algorithm and lighter data structures.

However, since the constraints added by Dual Consistency are implied constraints, they can be naturally processed and relaxed without adding solutions to the CSP. The Conservative DC proposed in [10] is already a successful example of such an approximation, where no-goods that cannot be stored in one of the original constraints of the problem are discarded. As a perspective, we can propose to extend our work in these directions:

- Relax implied constraints such that they confer to some property, e.g. row convexity. Table 3 gives an example of a hard problem solved by using an approximated form DC to generate row-convex constraints (no-goods that violate the row-convexity property are discarded). Full DC could not be applied on the same problem (see Table 2).
- Propagate the implied constraint only when they prove to be useful. In particular, we can apply works such as [15] without having to detect the redundant constraints.

5 Conclusion

In this paper, we proposed an extension of the definition of Dual Consistency, and thus Path Consistency, to non-binary constraint networks. We showed how DC can be used a way to automatically deduce implied binary constraints from the original domains and constraints of the constraint network. A new variant of the Strong Dual Consistency algorithm, called sDC^{clone}, with an enhanced worst-case time complexity, and specific handling of non-binary constraints was described. An efficient compromise, called sDC-2.1, was proposed. Experiments on the new algorithm with "on the fly" generation of implied binary constraints were described, and showed that this approach is promising.

Finally, the main drawbacks of Dual Consistency, that can prevent its use on real-world problems, were identified. New open ideas were proposed to bring solutions to these drawbacks.

References

1. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
2. C. Bessière. *Handbook of Constraint Programming*, chapter 3: Constraint Propagation, pages 29–84. Elsevier, 2006.
3. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1):29–41, 2008.
4. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 1997.
5. C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
6. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
7. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004.
8. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
9. I.P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In *Proceedings of CP'06*, pages 182–197, 2006.
10. C. Lecoutre, S. Cardon, and J. Vion. Conservative Dual Consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.
11. C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proceedings of CP'2007*, 2007.
12. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.
13. C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2:21–35, 2008.
14. U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
15. C. Piette. Let the solver deal with redundancy. In *Proceedings of ICTAI'08*, volume 1, pages 67–73, 2008.
16. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.
17. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
18. M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. http://www.cril.univ-artois.fr/CPAI08, 2008.
19. P. van Hentenryck, Y. Deville, and CM. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
20. J. Vion. Constraint Satisfaction Problem for Java. http://cspfj.sourceforge.net/, 2006.