Dynamic Virtual Arc Consistency [*]

Hiep Nguyen
Unité de Biométrie et Intelligence Artificielle
INRA, Toulouse, France
hnguyen@toulouse.inra.fr

Thomas Schiex
Unité de Biométrie et Intelligence Artificielle
INRA, Toulouse, France
tschiex@toulouse.inra.fr

Christian Bessiere
University of Montpellier
Montpellier, France
bessiere@lirmm.fr

ABSTRACT

Virtual Arc Consistency (VAC) is a recent local consistency for processing Cost Function Networks (or Weighted Constraint Networks) that exploits a simple but powerful connection with classical Constraint Networks. It has allowed to close hard frequency assignment benchmarks and is capable of directly solving networks of submodular functions. The algorithm enforcing VAC is an iterative algorithm that solves a sequence of classical Constraint Networks. In this work, we show that Dynamic Arc Consistency algorithms can be suitably injected in the virtual arc consistency iterative algorithm, providing noticeable speedups.

Keywords

WCSP, Weighted CSP, Cost Function Networks, arc consistency, dynamic arc consistency, virtual arc consistency

1. INTRODUCTION

Graphical model processing is a central problem in AI. The optimization of the combined cost of local cost functions, central in the valued CSP framework [12], captures problems such as weighted MaxSAT, Weighted CSP or Maximum Probability Explanation in probabilistic networks. It has applications in *resource allocation*, *combinatorial auctions*, *bioinformatics*...

Dynamic programming approaches such as bucket or cluster tree elimination can be used to tackle such problems but are inherently limited by their guaranteed exponential time and space behavior on graphical models with high treewidth. Instead, Depth First Branch and Bound allows to keep a reasonable space complexity but requires good (strong and cheap) lower bounds on the minimum cost to be efficient.

In the last years, increasingly better lower bounds have been designed by enforcing local consistencies on Cost Function Networks (CFNs). Enforcing is done by the iterated application of so-called *Equivalence Preserving Transformations* (EPTs, [7]) which extend the usual local consistency operations used in pure CSP. EPTs move costs between cost functions of different arities while keeping the problem equivalent. By ultimately moving cost to a constant function with empty scope, they are able to provide a lower bound on the optimum cost which can be incrementally maintained during branch and bound search.

Traditional local consistencies such as AC*, DAC*, FDAC* or EDAC* [10] apply available EPTs in any order. Instead, Virtual Arc Consistency (VAC [5, 6]) planifies the sequence of EPTs to apply from the result of enforcing classical AC on a classical constraint network which forbids combinations of values with non zero costs. VAC is not only stronger than those local consistencies: it is also able to solve submodular cost functions, it has a low order polynomial enforcing algorithm and has allowed to close hard frequency assignment benchmarks [5]. But it is still too expensive for general use.

In this paper, we significantly increase the efficiency of VAC by exploiting its iterative behavior. Indeed, each iteration of VAC requires to enforce classical AC on the hardened version of the current network. But this network is just the result of the incremental modifications done by EPTs applied in the previous iterations. This situation, where AC is iteratively enforced on incrementally modified versions of a constraint network, has been previously considered in Dynamic Arc Consistency algorithms [1, 3] for Dynamic CSPs [8].

After suitable adaptation, we observe that the introduction of Dynamic arc consistency inside VAC provides significant speedups on a variety of problems. This may be, as far as we know, one of the first successful application of Dynamic AC algorithms.

2. BACKGROUND

2.1 Cost function networks

A Cost Function Network (CFN), or weighted CSP (WCSP) is a tuple (X, D, W, m) where X is a set of n variables. Each variable $i \in X$ has a domain $D_i \in D$. For a set of variables S, we denote by $\ell(S)$ the set of tuples over S. W is a set of e cost functions. Each cost function $w_S \in W$ assigns costs to assignments of variables in S i.e. $w_S : \ell(S) \to [0..m]$ where $m \in \{1, ..., +\infty\}$. The addition and subtraction of costs are bounded operations, defined as $a \oplus b = \min(a + b, m)$, $a \ominus b = a - b$ if $a < m$ and m otherwise. The cost of a complete tuple t is the sum of costs $Val_P(t) = \bigoplus_{w_S \in W} w_S(t[S])$

[*]This work has been partly funded by the "Agence nationale de la Recherche", reference ANR-10-BLA-0214.

where $t[S]$ is the projection of t on S. In this paper, we restrict ourselves to binary CFNs. We use notations w_i and w_{ij} for the unary and binary cost functions on variable i and on variables i, j respectively. We assume the existence of a unary cost function w_i for every variable, and a nullary cost function, noted $w_\varnothing$. This constant positive cost defines a lower bound on the cost of every solution.

Enforcing a given local consistency on a CFN P transforms it in an equivalent problem P' ($Val_P(t) = Val_{P'}(t)\ \forall t$) with a possibly increased lower bound $w_\varnothing$ on the optimal cost. Enforcing is done by using equivalence-preserving transformations (EPTs) which shift costs between cost functions. Algorithm 1 introduces three basic EPTs. $\mathsf{Project}(w_{ij}, i, a, \alpha)$ moves an amount of cost α from a binary cost function to a unary one. Conversely, $\mathsf{Extend}(i, a, w_{ij}, \alpha)$ sends an amount of cost α from a unary cost function to a binary one. Finally, $\mathsf{UnaryProject}(i, \alpha)$ projects an amount of cost α from a unary cost function to the nullary cost function $w_\varnothing$.

Algorithm 1: Three elementary EPTs

```
1 Procedure Project($w_{ij}, i, a, \alpha$)
2     $w_i(a) \longleftarrow w_i(a) \oplus \alpha$ ;
3     foreach $b \in D_j$ do $w_{ij}(a,b) \longleftarrow w_{ij}(a,b) \ominus \alpha$;
4 Procedure Extend($i, a, w_{ij}, \alpha$)
5     foreach $b \in D_j$ do $w_{ij}(a,b) \longleftarrow w_{ij}(a,b) \oplus \alpha$;
6     $w_i(a) \longleftarrow w_i(a) \ominus \alpha$ ;
7 Procedure UnaryProject($i, \alpha$)
8     foreach $a \in D_i$ do $w_i(a) \longleftarrow w_i(a) \ominus \alpha$;
9     $w_\varnothing \longleftarrow w_\varnothing \oplus \alpha$ ;
```

Notice that a classical binary CSP can be represented as a CFN with $m = 1$ (cost 1 being associated to forbidden tuples). As usual, a value (i, a) is said to be AC w.r.t. a constraint w_{ij} iff there is a pair (a, b) that satisfies w_{ij} (is a support) and such that $b \in D_j$ (is valid). A CSP is AC if all its values are AC w.r.t. to all constraints. Enforcing AC on a CSP produces its AC closure, which is equivalent to P and is AC.

2.2 Virtual Arc Consistency

DEFINITION 1. *Given a CFN $P = (X, D, W, m)$, the CSP Bool$(P) = (X, D, \overline{W}, 1)$ is such that $\exists \overline{w}_S \in \overline{W}$ iff $\exists w_S \in W$, $S \neq \varnothing$ and $\overline{w}_S(t) = 1 \Leftrightarrow w_S(t) \neq 0$. A CFN P is virtual arc consistent (VAC) iff the arc consistent closure of the CSP Bool(P) is non-empty [6].*

Bool(P) is therefore a CSP whose solutions are exactly all complete tuples having cost $w_\varnothing$ in P. If P is not VAC, enforcing AC on Bool(P) will lead to a domain wipe-out. In this case, it has been shown in [6] that there exists a sequence of EPTs which leads to an increase of $w_\varnothing$ when applied on P. To exploit this property, VAC enforcing uses an iterative three-phases process.

Phase 1 is an instrumented AC enforcing on the CSP Bool(P) that records every deletion in a dedicated data-structure denoted as killer. When a value (i, a) lacks a valid support on $\overline{w}_{ij}$, we set $\mathsf{killer((i,a))} = j$ and we delete the value. If no domain wipe-out occurs, P is VAC and we stop.

Then, Phase 2 identifies the subset of value deletions that are necessary to produce the wipe-out and stores them in a queue R. This is achieved by tracing back the propagation history defined by killer, in reverse order, from the wiped-out variable up to non-zero costs. Phase 2 then computes the maximum possible increase achievable in $w_\varnothing$, denoted λ, and the set of EPTs to apply to P in order to achieve this increase. As shown in [6], all the amounts of cost that the EPTs will move are stored in two arrays of integers, $k(j, b)$ and $k_{ij}(j, b)$, that store the number of λ that needs to be respectively projected on (j, b) and extended from (j, b) to w_{ij}. These cost moves follow a simple law of conservation. For any value (j, b) in non wiped-out variable j which is not a source of cost ($w_j(b) = 0$), the amount of cost that arrives in (j, b) by $\mathsf{Project}$ is exactly the amount of cost that leaves (j, b) by Extend (See [6], page 465).

$$\forall (j, b)\ s.t.\ w_j(b) = 0, k(j, b) = \sum_{w_{ij} \in W} k_{ij}(j, b) \qquad (1)$$

Phase 3 of VAC, as described in detail in Algorithm 2, modifies the original CFN by applying the EPTs defined by the data-structures k and k_{ij} on all the deleted values that have been stored in R. A value (j, b) deleted by $\overline{w}_{ij}$ will receive a cost of $k(j, b) \times \lambda$ by $\mathsf{Project}$ from w_{ij} (line 7). This requires to first extend a cost $k_{ij}(i, a) \times \lambda$ from the invalid supports (i, a) to w_{ij} (line 5). The result of this phase is a new problem P', equivalent to P but with an increased lower-bound $w_\varnothing$ (line 8).

It is important to realize that at the end of Phase 1, when a wipe-out is detected in variable i_0, there may still be undeleted values which have no valid support because the corresponding domain has not been revised. These "pending for revision" domains are represented in the propagation queue Q_{AC}. The remaining values either have a valid support or have been deleted and have a non empty associated killer. Such a problem will be called a *justified partial* AC closure of Bool(P). It cannot have larger domains than Bool(P) and all deletions are properly justified by their killer.

Algorithm 2: VAC - Phase 3: Applying EPTs

```
1 while $R \neq \varnothing$ do
2     $(j, b) \longleftarrow R.pop()$;
3     $i \longleftarrow$ killer $(j, b)$;
4     foreach $a \in D_i$ s.t. $k_{ij}(i, a) \neq 0$ do
5         Extend($i, a, w_{ij}, \lambda \times k_{ij}(i, a)$);
6         $k_{ij}(i, a) \longleftarrow 0$;
7     Project($w_{ij}, j, b, \lambda \times k(j, b)$);
8 UnaryProject($i_0, \lambda$);
```

This VAC algorithm takes a O(ed) space complexity. One iteration of the algorithm has a time complexity in O(ed^2) as long as an optimal AC algorithm is used in Phase 1.

Ultimately, VAC iterations enforce AC on a sequence of slightly modified CSPs: Bool(P), Bool(P'), ... This motivates the use of dedicated Dynamic AC algorithms to enforce VAC.

2.3 Dynamic Arc Consistency

DEFINITION 2. *A dynamic CSP problem is a sequence $P_0, P_1, ..., P_n$ of CPSs where each P_i is a CSP resulting from addition or retraction of a constraint in P_{i-1} [1].*

Dynamic arc consistency algorithms (DnAC) aim at maintaining arc consistency in the sequence of problems P_i. AC enforcing is naturally incremental for restriction (constraint addition): it suffices to start enforcing AC (Phase 1) with an initial queue Q_{AC} that contains just the variables of the new constraint. But AC is not incremental for relaxation (constraint removal). This last case needs therefore to be handled specifically: values that have been deleted directly or indirectly because of the removed constraint need to be restored, if there is no other reason to delete them.

Several algorithms have been proposed for DnAC. In this paper, we will use AC/DC2 [1] which improved AC/DC [2] by introducing persistent data-structures. The most important one is a *justification*(i, a) array (as in DnAC4 [3]) that remembers which constraint has been responsible for the deletion of (i, a). This is exactly equivalent to the killer data-structure of VAC.[1]

Upon retraction of the constraint w_{ij}, AC/DC2 goes through three stages: 1) initialization: only values in the domains of i and j which have been deleted because of w_{ij} are candidate for restoration and marked as "Propagable". This can be tested in the *justification* array. 2) propagation: each of the propagable values is propagated to neighboring variables to check if they offer a new valid support for deleted values (which also will be marked as "Propagable"). When a value has been propagated to all neighbor variables, it is marked as "Restorable" and will be restored. Note that a variable i having propagable values can check neighboring values (j, b) for restorability only if they have been removed due to the lost of support on the constraint w_{ji} (known through $justification$). 3) filtering: all restored values need to be checked again for arc consistency. This can be done using plain AC enforcing provided the queue Q_{AC} is initialized to enforce the revision of domains of variables with restored values.

The space complexity of AC3 based algorithm AC/DC is $O(nd + e)$. For AC/DC2, its time complexity is defined by the algorithm used for filtering in the last stage: $O(ed^3)$ for AC-3 and $O(ed^2)$ for AC-2001.

3. DYNAMIC VAC ALGORITHM

When VAC is enforced, the CFN P is incrementally modified in Phase 3 of every VAC iteration. Hence, we propose an improved version of VAC, called dynamic VAC (DynVAC), which uses dynamic AC to maintain AC on the successive Bool(P) instead of refiltering from scratch at each iteration as done in the standard VAC algorithm. We use AC/DC2 [1] based on AC2001 instead of AC3 as it is apparently the most efficient available algorithm. This also has the advantage that the *justification* data-structure of AC/DC2 is provided for free by the killer array in VAC.

In traditional Dynamic CSPs, DnAC algorithms are applied after each constraint removal or addition. In the case of VAC, the situation is more complex because a series of modifications of Bool(P) occurs during Phase 3 through applications of different EPTs. A simple call to Project(w_{ij}, i, a, α) can be decomposed in 1) an increase of cost of the unary cost function $w_i(a)$ and 2) a decrease of costs in the binary cost function w_{ij}. If a previously zero cost $w_i(a)$ becomes non-zero, the associated value (i, a) is removed from Bool(P) and this corresponds to a restriction. Conversely, if the non-zero cost of a pair (a, b) reaches zero, this previously forbidden pair in $\overline{w}_{ij}$ becomes authorized and this corresponds to a relaxation. Instead of applying a DnAC algorithm inside each Project, Extend and UnaryProject operation, we observe that a better approach consists in applying DnAC principles only after Phase 3 to avoid useless restorations/deletions of values by DnAC.

Each iteration of VAC transforms the current CFN P into a modified problem P' with cost functions w'_i and w'_{ij}. After Phase 2, it is already possible to compute the values of w'_i and w'_{ij} because they are defined by a known sequence of applications of Project, Extend and UnaryProject on w_i and w_{ij}. For example, if i is not the wiped-out variable, we have for any value a:

$$w'_i(a) = w_i(a) \oplus (k(i, a).\lambda) \underset{w_{ij} \in W}{\ominus} (k_{ij}(i, a).\lambda)$$

Similar computations can be done for the wiped-out variable and w'_{ij}. We now show that the global effect of all EPTs on Bool(P) in Phase 3 is a set of relaxations only, at the unary and binary levels.

PROPERTY 1. *Following Phase 2, we know that: a)* $\forall(i, a)$*:* $w'_i(a) \leq w_i(a)$*. b)* $\forall(i, a)$ *and* (j, b)*: if* $w'_{ij}(a, b) \neq w_{ij}(a, b)$ *then* (i, a) *or* (j, b) *is deleted in the current justified partial* AC *closure of Bool(P).*

PROOF. (a) In VAC, the only operation that may increase unary costs is the Project operation. However, according to equation 1, any value (i, a) that receives cost by Project will later Extend the same amount of cost (to other binary cost functions or to $w_\varnothing$). Hence, unary costs cannot increase.

(b) The only way for a binary cost $w_{ij}(a, b)$ to change is by a Project from w_{ij} or an Extend onto it. However, Phase 3 of VAC applies Project and Extend to values extracted from the queue R of deleted values (built by Phase 2). Therefore when the cost of a pair (a, b) changes, either (i, a) or (j, b) must have been deleted. □

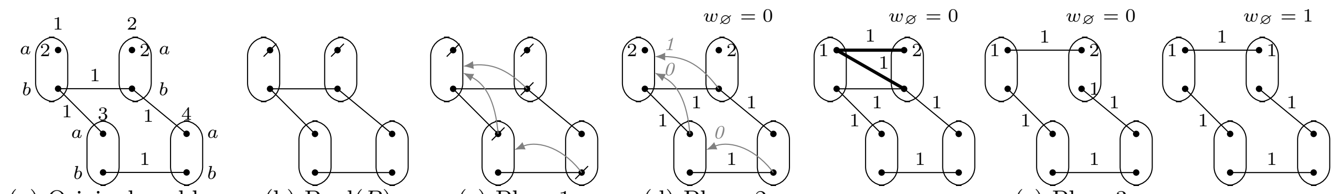
COROLLARY 1. *The EPTs applied in Phase 3 of VAC, transforming Bool*(P) *into Bool*(P')*, generate only the following types of relaxations: 1) values* (i, a) *that become authorized* $(w_i(a) > w'_i(a) = 0)$*. 2) pairs* $((i, a), (j, b)$ *that become authorized* $(w_{ij}(a, b) > w'_{ij}(a, b) = 0)$*.*

PROOF. From Property 1(a), we know that unary costs may only decrease. Some may therefore go from a non-zero cost to a zero cost. Therefore the corresponding value re-appears in Bool(P'). This can be considered as the retraction of a unary constraint.

From Property 1(b), the costs of pairs may either increase or decrease. If a binary cost (a, b) increases from zero to non-zero, this cannot destroy a valid support because either of the 2 values is deleted in the current partial closure. The support cannot be valid. If the cost of (a, b) decreases however, it may create a new support for a or b. □

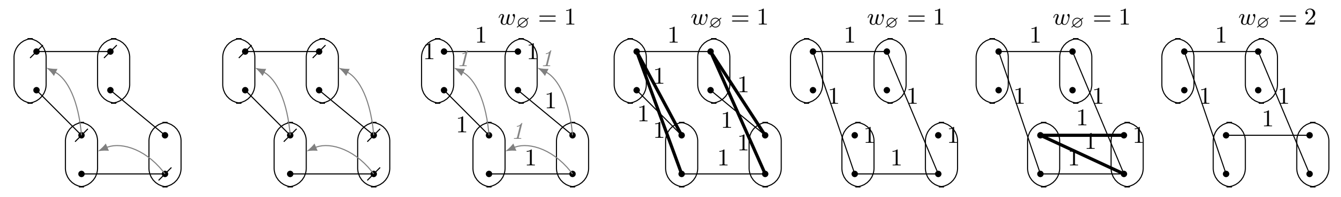
Therefore, the DnAC algorithm used can be specialized for relaxations (Algorithm 3). The restoration protocol consists of 3 stages, as in AC/DC2. Note that Bool(P) is maintained after Phase 3 of each iteration, so $\overline{D}_i$ mentioned in the following represents the domain of variable i in the final justified partial AC closure obtained after Phase 1.

[1] AC/DC2 also introduces a *time-stamp*(i, a) structure that remembers the order of deletions. The authors of AC/DC2 have acknowledged that this data-structure is actually subsumed by the *justification*(i, a) data-structure (Private communication).



(a) Original problem (b) Bool(P) (c) Phase1 (d) Phase 2 (e) Phase3

Figure 1: Iteration 1



(a) Updated Bool(P) (b) Phase 1 (c) Phase 2 (d) Phase3

Figure 2: Iteration 2

The **initialization stage** scans all the values in the queue R to identify which values should be restored (line 2).The wiped-out variable i_0 is processed separately (line 7). As Corollary 1 shows, there are 2 possible cases: (1) when a value (i, a) becomes authorized $w_i(a) > w'_i(a) = 0$, it will be restored (line 5), (2) when a new valid support appears for a value (j, b) by satisfying $(w_{ij}(a, b) \oplus w_i(a)) > (w'_{ij}(a, b) \oplus w'_i(a)) = 0$ and killer $(j, b) = i$, (j, b) will be restored (line 6). When a value (i, a) is restored, it is stored in an array *restored*$[i]$ and variable i is kept in a list RL for future propagation.

Algorithm 3: Update Bool(P)

```
1  Procedure Initialization
2    foreach $(j, b) \in R$ do
3      $i \longleftarrow$ killer $[j, b]$;
4      foreach $a \in D_i - \overline{D}_i$ do
5        if $(w_i(a) > 0) \wedge (w'_i(a) = 0)$ then Restore$(i, a)$;
6        if $b \notin \overline{D}_j \ \wedge\ w'_i(a) = 0 \ \wedge\ w'_{ij}(a, b) = 0$ then Restore$(j, b)$;
7    foreach $a \in D_{i_0}$ s.t. $w_{i_0}(a) > 0 \ \wedge\ w'_{i_0}(a) = 0$ do Restore$(i_0, a)$;
8  Procedure Restore$(i, a)$
9    add $a$ into $\overline{D}_i$ and restored$[i]$;
10   add $i$ into $RL$;
11   killer $[i, a] \leftarrow$ nil;
12 Procedure Propagation
13   while $RL \neq \varnothing$ do
14     $i \leftarrow RL.pop()$;
15     foreach $w_{ij} \in W$ do
16       foreach $b \in D_j - \overline{D_j}$ s.t. killer $[j, b]= i$ do
17         if $\exists a \in$ restored$[i]$ s.t. $w'_{ij}(a, b) = 0$ then Restore$(j, b)$;
18     restored$[i] \leftarrow \varnothing$;  $Q_{AC} \leftarrow Q_{AC} \cup \{j \mid w_{ij} \in W\}$
```

The **propagation stage** (line 12) propagates value restorations to direct neighbours of the variables whose domain has been extended, as in AC/DC2. Each such variable i can restore a value (j, b) in a neighbour variable j if it was deleted due to w_{ij} (line 16) and is now supported by a restored value in i (line 17). After propagating all restored values, the *restored* list is emptied (line 18) to avoid repropagating values which have been already propagated.

The **filtering stage** must eliminate the restored values (i, a) which are not arc consistent on some constraint $\overline{w}_{ij}$ and properly set the associated killer (i, a) to j. This is precisely what is achieved by the Phase 1 of VAC. Hence, we integrated this stage into Phase 1 by adding the neighbour variables of variables having restored values into the revision propagation queue Q_{AC} (line 18).

The DynVAC algorithm can be proved to be correct by showing that the result it provides to the next iteration of VAC is a justified partial AC closure of Bool(P'). We do not give the proof here because of limited space (it will be made available in the final version using additional pages).

3.1 Example

The essential gain of DynVAC compared to VAC lies in the fact that the list of variables to revise during Phase 1 is not reset to the full set of variables X at each iteration but is instead maintained along all iterations, avoiding useless repeated filtering. We illustrate this on a small example.

Consider the binary CFN in Figure 1(a). Each variable has two values a and b represented as vertices. Non-zero unary costs are displayed besides values. An edge between two vertices indicates that the corresponding pair has a non-zero binary cost. Zero costs are not represented. In Bool(P) (Figure 1(b)), forbidden values are shown as crossed-out and edges represent *forbidden* pairs.

Suppose that the revision order in Phase 1 is (w_{13}, w_{34}, w_{12}, w_{24}). After revising w_{13}, w_{34}, w_{12}, $(3, a)$, $(4, b)$ and $(2, b)$ have been deleted from Bool(P) respectively. Phase 1 stops because variable 2 has wiped-out (Figure 1(c)). The gray arrows represent the state of the killer data-structure, pointing to the variable that offered no valid support. In

Phase 2 (Figure 1(d)), the deletion of $(2, b)$ alone is sufficient for the wipe-out. It uses the non-zero costs $w_{12}(b, b)$ and $w_1(a)$ to provide $w_\varnothing$ with a maximal amount of cost $\lambda = 1$. The numbers in italic associated with gray arrows precisely indicate the corresponding value of $k(i, a)$. Applying identified EPTs, Phase 3 (Figure 1(e)) transforms P into an equivalent problem P' with $w'_\varnothing = 1$. Extended costs are shown in bold.

VAC enforcing continues because P' is still not VAC. To update Bool(P) in Fig. 1(c), we consider $((1, a), (2, a), (2, b))$ for restoration because only w_{12} has been modified by EPTs in Phase 3. Only $(2, b)$ is restored because it has a zero cost and a support (b, b) on w_{12}. This restoration does not lead to further restorations. The constraints of the updated Bool(P) are directly defined by P'. In fact, the updated result in Figure 2(a) is already a justified partial AC closure of Bool(P') with two extra deleted values $(3, a)$ and $(4, b)$ and the associated killer. The next Phase 1 starts from this updated Bool(P). $(4, a)$ is removed after revising w_{24} and variable 4 wipes out (Figure 2(b)). Phase 2 and Phase 3, perform as in the previous iteration. The final problem (Figure 2(d)) with $w''_\varnothing = 2$ is VAC.

3.2 Complexity

DynVAC has the same space complexity as VAC, in O(ed).

The update initialization stage has a O(nd) time complexity because there are at most $n \times d$ values having positive costs (line 5, Algorithm 3).

The worse-case time complexity of the propagation stage is O(ed^2) because each pair of values involved in a constraint is tested at most once (line 16, Algorithm 3).

So, the time complexity to update Bool(P) is O(ed^2), which does not modify the complexity of Phase 3 of VAC.

An iteration of DynVAC has therefore a O(ed^2) time complexity, similarly to VAC, as long as an optimal AC algorithm is used in Phase 1. Although DynVAC does not have an improved asymptotic complexity compared to VAC, experimental tests presented in the next section will show important speedups in practice.

4. EXPERIMENTS

In this section, we compare the efficiency of DynVAC and VAC used as pre-processing algorithms on a large set of benchmarks from the Cost Function Library[2]. For each problem, as in [5,6], we enforce a limited version of VAC that stops iterating as soon as the increase in the lower bound $w_\varnothing$ becomes less than $\varepsilon = 0.05$. The experiments are implemented in the `toulbar2` solver and run on a 2.66GHz Intel(R) Core(TM)2 with 4GB of RAM.

Table 1 presents results including the run-time and the lower bound $w_\varnothing$(lb) on a set of specific instances which have previously been used for VAC experimentations in [5, 6]. The problems are Radio Link Frequency Assignment problems (*celar*, [4]) uncapacited warehouse location problems (*warehouse*, [9]) and bioinformatics Tag-SNP identification problems (*tagsnp*, [11]). These problems are interesting because they are non trivial in terms of size and not already VAC. The final lower bound produced by DynVAC and VAC should not necessarily the same because they may use different orders of constraint revision in Phase 1 after the first iteration.

[2] https://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/?root=costfunctionlib

Table 1: Results of DynVAC compared to VAC on a subset of usual significant instances

		VAC$_\varepsilon$		DynVAC$_\varepsilon$	
		lb	time	lb	time
celar	scen07$_r$	28,809	17.29	29,639	11.39
	scen08$_r$	8	1.34	8	1.12
	scen11$_r$	2,953	0.76	2,953	0.45
	scen13$_r$	9,741	10.16	9,683	4.3
warehouse lb ($\times 10^6$)	capa	17.16	2,362.25	17.16	325.35
	capb	12.80	2,111.44	12.96	397.92
	capc	11.45	2,389.82	11.46	439.32
	mq1	3.25	1,990.23	3.30	442.91
	mq2	3.29	2,041.63	3.15	407.68
	mq3	3.24	2,244.76	3.12	455.56
	mq4	3.36	2,172.19	3.48	418.26
	mq5	3.21	1,864.61	3.24	330.03
tagsnp lb ($\times 10^6$)	10442	20.69	19.29	20.69	3.98
	11739	5.62	2,748.7	5.62	142.07
	13306	7.19	10.46	7.19	3.43
	14226	25.67	33.5	25.67	8.99
	16421	3.15	43.71	3.15	5.69
	17034	38.31	113.34	38.31	25.51
	6858	20.15	123.19	20.15	40.49
	8956	6.66	21.9	6.66	2.97
	9150	40.51	125.44	40.51	8.51
	6389	13.53	551.18	13.28	11.47
	27498	4.47	33.55	4.47	12.48
	30461	20.29	91.48	20.29	29.19
	39997	1.52	147.55	1.52	28.62

As expected, DynVAC outperforms VAC on these instances. This is already visible in the *celar* benchmarks, where speedups by a factor of 2 or more are observed. Even more impressive speedups, up to a factor of 7, can be observed on the *warehouse* or *tagsnp* instances. Two explanations can be identified for these large speedups. First, these problems contain very large costs, leading to a large number of VAC iterations. This gives more opportunities for DynVAC to reuse work done on previous iterations. Second, these problems have variables with large domains, making AC enforcing costly, thus boosting the effects of DynVAC savings. These lower bounds of DynVAC can be either smaller or larger than the bounds provided by the static VAC but over all instances, the difference never exceeds 3%.

To collect more extensive evidence of the interest of DynVAC, we applied DynVAC on a larger set of instances extracted from the CFLib repository. Table 2 shows the mean value of the run-time (in seconds), the lower bound (lb) and the number of iterations (iter) for enforcing VAC using either the usual static VAC algorithm or the new DynVAC variant. Each line corresponds to a different problem class covering *Size* instances. The cases where our DynVAC algorithm performs better in time or lower bound compared to the static VAC are indicated in bold. These experiments show that DynVAC is respectively 1.6, 3 and 5 times faster than VAC for the classes *celar, tagsnp, warehouse* while providing similar lower bounds on average. Notice that the mean number of iterations of DynVAC may increase compared to VAC, but the total average run-time is nevertheless reduced as expected: there is less work to do at each iteration in DynVAC because deletions are inherited from previous iterations.

Table 2: Value of the lower bound, cpu-time and number of iterations needed to process different real and crafted problem from the "Cost Function Library". The "Size" column indicates the number of instances in the class.

class	Size	VAC$_\varepsilon$			DynVAC$_\varepsilon$			DynVAC$_\varepsilon$ with heuristic		
		lb	time	iter	lb	time	iter	lb	time	iter
celar	32	6,180	3.14	382	**6,204**	**1.92**	418	5,892	1.12	319
protein_maxclique	10	1,016	**51**	1,022	1,016	364	1,022	1,016	56.95	1,022
tagsnp_r0.5	25	1.43×10^6	364.31	8,798	1.43×10^6	**116.57**	4,653	1.43×10^6	81.46	5,810
tagsnp_r0.8	82	1.11×10^6	4.64	155	1.11×10^6	**1.53**	120	1.11×10^6	2.54	150
dimacs_maxclique	65	266	**0.78**	284	266	3.65	284	266	0.96	284
planning	68	1,074	0.25	46	1,074	**0.19**	50	1,072	0.23	76
warehouse	57	7.23×10^6	341	946	**7.24**$\times10^6$	**66**	719	7.25×10^6	114.17	790

However, DynVAC is significantly slower than VAC (respectively 7 and 4 times) for all the maximum clique problems categories we tested: *protein_maxclique* and *dimacs_maxclique*. These problems have a specific structure, with only Boolean domains, unary cost functions with only unit costs and binary differences. On these problems, we noticed that each iteration leads to the useless restoration of many values in cascade which will again be uselessly deleted in the next iteration. These values are therefore iteratively restored and deleted at each iteration, increasingly slowing down the algorithm.

In order to improve the efficiency of DynVAC, we have used the variable-based revision heuristic proposed in [13] during AC enforcing in Phase 1 to improve the efficiency of the Phase 1. This heuristic selects first in Q_{AC} the variable having the smallest current domain size in Bool(P). Then, constraints are processed in ascending order of the domain sizes of the opposite variables. The results, presented in the last column of Table 2, show that the heuristic allows to drastically improve the performance of DynVAC on maximum clique problems, leading to performances which are comparable to the static VAC in this highly unfavorable case (a 20% overhead in time, which is probably largely explained by the computation of the ordering heuristics).

5. CONCLUSION

This paper presents an incremental approach for enforcing VAC in CFNs. It combines the idea of dynamic arc consistency algorithms with the iterative VAC algorithm in order to efficiently maintain arc consistency in the CSP Bool(P) during VAC enforcing.

The new algorithm provides a lower bound of the same quality level as the static VAC algorithm but rapidly outperforms static VAC on many problems, as expected. This is especially true for problems involving large costs and large domains. However, DynVAC may become slow on some specific problems like the *maximum clique* problems. Using a revision heuristic on domain sizes inside the AC instrumented algorithm allows to avoid this pathological behavior while still providing good speedups on other problems.

In the future, we intent to use DynVAC to maintain VAC in a branch and bound search tree to directly solve CFNs and also to extend the algorithm to non-binary cost functions. Another interesting area would be to identify other revision ordering heuristics that could improve the performance of DynVAC but also improve the produced lower bound.

6. REFERENCES

[1] R. Barták and P. Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. In *Proc. of the* 18th *International FLAIRS Conference*, pages 161–166, Menlo Park, CA, USA, 2005. AAAI Press.

[2] P. Berlandier and B. Neveu. Maintaining Arc Consistency through Constraint Retraction. In *Proc. of the 6th IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

[3] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. of AAAI'91*, pages 221–226, Anaheim, CA, 1991.

[4] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints Journal*, 4:79–89, 1999.

[5] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proc. of AAAI'2008*, Chicago, USA.

[6] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174:449–478, 2010.

[7] M. C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.

[8] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI'88*, pages 37–42, St. Paul, MN, 1988.

[9] J. Kratica, D. Tosic, V. Filipovic, and I. Ljubic. Solving the Simple Plant Location Problems by Genetic Algorithm. *RAIRO Operations Research*, 35:127–142, 2001.

[10] J. Larrosa, S. de Givry, F. Heras, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proc. of the 19th IJCAI*, pages 84–89, Edinburgh, Scotland, Aug. 2005.

[11] M. Sanchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *Proc. IJCAI'09*, pages 603–608, San Diego (CA), USA, 2009.

[12] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proc. of the 14th IJCAI*, pages 631–637, Montréal, Canada, Aug. 1995.

[13] R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Biennial Conference-Canadian Society for Computational*

Studies of Intelligence, pages 163–163, 1992.