Maintaining Virtual Arc Consistency Dynamically during Search

Simon de Givry, Thomas Schiex, Christian Bessiere, Thi Hông Hiêp Nguyên

Maintaining Virtual Arc Consistency Dynamically During Search

Hiep Nguyen, Simon de Givry, Thomas Schiex
Unité de Mathématiques et Informatique Appliquées de Toulouse
Toulouse, France
thhnguen/sdegivry/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 (aka weighted constraint networks) that exploits a simple but powerful connection with standard 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 standard constraint networks. This algorithm has been improved by exploiting the idea of dynamic arc consistency between each iteration, leading to the dynamic VAC algorithm. When VAC is maintained during search, the difference between two adjacent nodes in the search tree is also limited. In this paper, we show that the incrementality of Dynamic VAC can also be useful when maintaining VAC during search and we present results showing that maintaining dynamic VAC during search can effectively accelerate search.**

***Index Terms*—Weighted CSP, Cost Function Networks, arc consistency, dynamic arc consistency, virtual arc consistency**

I. Introduction

Graphical model processing is an important problem in Artificial Intelligence. The optimization of the combined cost of local cost functions, central in the valued CSP framework [1], captures a variety of problems such as Weighted CSP, Maximum Probability Explanation in probabilistic networks or weighted MaxSAT. It has applications in *resource allocation* [2], *combinatorial auctions*, *bioinformatics* [3],...

A variety of techniques can be used to solve this problem. When the problem has a graph of bounded tree-width, it can be solved using dynamic programming approaches such as bucket or cluster tree elimination. In the general case, Depth First Branch and Bound (DFBB) has the advantage of a reasonable space complexity. However, DFBB requires strong and computationally inexpensive incremental lower bounds on the minimum cost of a node to be efficient. These bounds can also be useful in the context of AND/OR and Tree decomposition based tree search algorithms [4], [5], or more space intensive search algorithms such as A*.

In the last decade, increasingly better lower bounds have been proposed for optimization in graphical models. They can be local consistency based bounds for solving Cost Function Networks [6] (CFNs), MaxSAT resolution based for solving the Partial Weighted MaxSAT problem [7], [8], or Linear programming dual based bounds for computing a maximum probability assignment in Markov Random Fields [9]: they all reformulate the current problem in a new equivalent problem using local cost transfer operations, also known as *Equivalence Preserving Transformations* (EPTs, [10]). EPTs extend the traditional local consistency operations by moving 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* [6], [11], apply available EPTs in any order. Instead, Virtual Arc Consistency (VAC [6], [12]) planifies the sequence of EPTs to apply from the result of enforcing AC on a standard constraint network which forbids tuples with non zero cost. VAC is not only stronger than those local consistencies: it is also able to directly solve networks of submodular cost functions, it has a low order polynomial time enforcing algorithm and has allowed to close hard frequency assignment benchmarks [12]. However, it is still quite expensive for general use and needs to be accelerated.

In this paper, we show that the efficiency of maintaining VAC during search, just like its iterative behavior, can be accelerated by exploiting the incrementality of the changes due to branching operations. Indeed, whether an iteration of VAC has just been executed or a decision has been taken by Branch and Bound search, maintaining VAC requires to repeatedly enforce standard AC on the hardened version of an incrementally modified version of the problem. The exploitation of incremental changes while maintaining Arc Consistency is the traditional target of Dynamic Arc Consistency algorithms [13], [14] for Dynamic CSPs [15]. We show that, just like the incremental changes of EPTs during successive iterations of VAC [16], the incremental changes between the problems considered at adjacent nodes in the search tree can be suitably exploited by Dynamic Virtual Arc Consistency, leading to a global improvement of the efficiency on a variety of problems extracted from the Cost Function Library[1] and other related resources.

[1]See https://mulcyber.toulouse.inra.fr/projects/costfunctionlib.

II. Background

A Cost Function Network (CFN), aka weighted CSP (WCSP) is a tuple $P = (X, D, W, m)$ 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. $w_S : \ell(S) \rightarrow [0..m]$ where $m \in \{1, ..., +\infty\}$. The addition and subtraction of costs are bounded operations, defined as $a +_m b = \min(a + b, m)$, $a -_m b = a - b$ if $a < m$ and m otherwise. The cost of a complete tuple t is the sum of costs $Val_P(t) = \sum_{w_S \in W} w_S(t[S])$ where $t[S]$ is the projection of t on S. We assume w.l.o.g. the existence of a unary cost function w_i for every variable, and a nullary cost function, noted $w_\varnothing$. All costs being non negative in a CFN, this constant cost defines a lower bound on the cost of every solution. In this paper, for simplicity, we restrict ourselves to binary CFNs.

Enforcing a given local consistency on a CFN P transforms it in an equivalent problem P' (such that $Val_{P'}(t) = Val_P(t)\ \forall t$) with a possibly increased lower bound $w_\varnothing$. The equivalence-preserving transformations (EPTs) which shift costs between cost functions are the elementary operations of local consistency enforcing. Algorithm 1 describes the main EPTs used for enforcing arc consistency, which shifts a cost $\alpha > 0$ between a binary cost function w_{ij} and a value (i, a). This operation corresponds to either a projection of costs from w_{ij} to (i, a) when $\alpha > 0$ or an extension of costs from (i, a) to w_{ij} when $\alpha < 0$. In node consistency, similar cost transfers can occur from unary cost functions to $w_\varnothing$, thus strengthening the lower bound.

Algorithme 1 : Equivalence Preserving Transformation

```
1 Procedure Shift(i, j, a, α)
      // precondition: w_i(a) + α ≥ 0 and
         w_ij(a, b) ≥ α for ∀b ∈ D_j
2     w_i(a) ⟵ w_i(a) + α ;
3     foreach b ∈ D_j do  w_ij(a, b) ⟵ w_ij(a, b) − α ;
```

Notice that a standard constraint network (CN) P can be represented as a CFN with $m = 1$ (a cost of 1 is associated with forbidden tuples). In a binary CN, represented as a CFN with $m = 1$, a value (i, a) is Arc Consistent (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 CN is AC if all its values are AC w.r.t. to all constraints. Enforcing AC on a CN produces its AC closure, which is equivalent to P and is AC.

III. Dynamic VAC

A. Virtual arc consistency

Definition 1. *Given a CFN $P = (X, D, W, m)$, the CN 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 CN Bool(P) is non-empty [6].*

Bool(P) is therefore a CN whose solutions are exactly all the complete assignment 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, when applied on P, lead to an increase of $w_\varnothing$. To exploit this property, VAC enforcing uses an iterative process, each iteration being decomposed in three phases.

- Phase 1 is an instrumented AC enforcing on the CN Bool(P) that records the cause of every deletion in a dedicated data-structure denoted as killer. killer$(i, a) = j$ means that (i, a) has been deleted because of $\overline{w}_{ij}$. killer$(i, a) = i$ means that (i, a) is a value deleted because of its own positive cost. Otherwise, killer$(i, a) = nil$ is used when (i, a) is not deleted in Bool(P). When a value (i, a) lacks a valid support on $\overline{w}_{ij}$, we set killer$(i, a) = j$ and we delete the value. If no domain wipe-out occurs, P is VAC and we stop.
- Otherwise, Phase 2 is performed to identify a minimal subset of value deletions that are necessary to produce the wipe-out by tracing back the propagation history defined by killer, in reverse order, from the wiped-out variable up to non-zero costs. Phase 2 also computes the maximum possible increase achievable in $w_\varnothing$ and the set of EPTs to apply to P in order to achieve this increase.
- Phase 3 of VAC, modifies the original CFN by applying the EPTs defined in Phase 2.

These 3 phases can be iterated until the problem is VAC or until the lower bound does not increase more than a threshold ε. In this last case, we say that we enforce VAC$_\varepsilon$. In all cases, each iteration leaves a locally reformulated problem which is used as the starting point of the next iteration.

B. Dynamic Virtual Arc Consistency

Ultimately, VAC iterations therefore enforce AC on a sequence of slightly modified CNs: Bool(P), Bool(P'), ... The first proposed VAC enforcing algorithm [6] enforces AC from scratch on each of these problems. The incremental changes on P performed at Phase 3 of each iteration induce incremental changes in Bool(P), and this motivated the use of dedicated Dynamic AC (DnAC) algorithms to enforce VAC [16].

The aim of DnAC algorithms is to maintain AC in CN problems after constraint additions or retractions. Among existing algorithms proposed for DnAC, the AC/DC2 algorithm has been used because of its simplicity and good empirical results [13]. AC/DC2 uses a data structure *justification*(i, a) to remember the cause of deletion for (i, a). This is exactly equivalent to the killer data-structure of VAC and therefore *justification* is available for free. The DynVAC algorithm proposed in [16] uses a version of AC/DC2 [13] based on AC2001 instead of AC3.

AC being naturally incremental for restriction, the addition of a constraint can be directly handled by any coarse grained AC algorithm that relies on a variable based propagation queue Q_{AC} by pushing the variables of the new constraint in Q_{AC}. Relaxation (constraint removal) is not so easy. In this case,

AC/DC2 goes through three stages. In the initialization stage, only values which have been deleted because of the removed constraint w_{ij} are considered as candidates for restoration. In the propagation stage, all the neighboring values (k, c) of a variable i having restored values and that have been removed due to the lack of support on the constraint w_{ki} (known through *justification*$\equiv$ killer) are iteratively restored. In a last stage, all restored values need to be checked again for arc consistency.

In the case of VAC, at each iteration, a series of modifications of Bool(P) can occur during Phase 3 through the application of the identified sequence of EPTs. It has been shown in [16] that the global effect of the EPTs applied on Bool(P) in Phase 3 consists *only* in a series of relaxations, at the unary and binary levels. More precisely, when the application of Phase 3 transforms the previous CFN P in an equivalent CFN P', values or pairs with non zero cost in P and therefore forbidden in Bool(P) may become authorized in Bool(P') if their cost becomes 0 in Bool(P').

Therefore, the DnAC algorithm used can be specialized for relaxations in Dynamic VAC. The restoration protocol has 3 stages, as in AC/DC2. The **initialization stage** aims at identifying restorable values. As shown in [16], this stage can be performed after Phase 2 because it is already possible to compute the values of the reformulated cost functions w'_i and w'_{ij} of P'. This stage scans only values on which EPTs will be performed in Phase 3 and restores all restorable values. 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.

The **propagation stage** 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) if it was deleted due to $\overline{w}_{ij}$ and is now supported by a restored value in i.

The final **filtering stage** eliminates 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, this stage is integrated into Phase 1 by adding the neighbour variables of variables having restored values into the revision propagation queue Q_{AC}.

IV. Maintaining Dynamic VAC During Search

The standard way to maintain VAC during search is to rebuild the CN Bool(P) at each new node and then use this new Bool(P) to enforce VAC in the CFN P. If DynVAC has been shown to enhance VAC efficiency for preprocessing CFNs at the root node, DynVAC has never been maintained during search.

Similarly, the simplest way to maintain Dynamic VAC during search consists in enforcing Dynamic VAC at each node. However, maintaining Dynamic VAC during search offers new opportunities for incrementality. In this section, we show that VAC can be incrementally maintained not only between successive iterations of VAC inside a node, but also during search between nodes, by incrementally maintaining an AC-closure of Bool(P) beween search nodes.

Between two consecutive nodes, the CN Bool(P) is only modified according to the changes in P caused by branching operations. The branches out of a node can be the assignment of a variable $(i = a)$ or a domain restriction $(i \neq a)$, $(i < a)$ or $(i > a)$. We expect that maintaining AC with a Dynamic AC algorithm in such a slightly modified Bool(P) will be beneficial compared to a cold restart. Suppose that branching operations transform the current CFN P into $P_{|i=a}, P_{|i\neq a}, P_{|i>a}$ or $P_{|i<a}$. We denote all these different cases as $P_{|i}$. After a branching operation, $P_{|i}$ will have modified domains and also possibly modified cost functions. These will be respectively denoted as D' and w'_i or w'_{ij}. To enforce VAC incrementally between nodes, we need to compute the AC closure of Bool$(P_{|i})$ from the AC closure of Bool(P) using only the modification from P to $P_{|i}$.

When a variable domain D_i is restricted by a branching operation of the form $(i \neq a)$, $(i < a)$ or $(i > a)$ that does not reduce the domain to a singleton, the removal of values leads to a new domain D'_i and thus a new domain, denoted as $\overline{D'}_i$, in the currently computed AC closure of Bool$(P_{|i})$. Some values of the neighboring variables j may have lost their support on $\overline{w'}_{ij}$ in Bool$(P_{|i})$ and thus need to be checked for AC. Similary to the case of constraint restriction, this can be naturally achieved by DnAC by (1) adding the neighbour variables of i in the propagation queue Q_{AC} and (2) restarting propagation to remove all values which are arc inconsistent in Bool$(P_{|i})$ due to this domain restriction.

When instead, an assignment is performed (or equivalently the domain is reduced to a singleton), solvers exploits this very specific situation to directly eliminate the variable i from the problem. Therefore, the modifications caused by a variable assignment $(i = a)$ are more complex than for other domain reductions. Indeed, when the domain of a variable i is restricted to a single value a, the unary cost $w_i(a)$ of (i, a) is immediately projected on $w_\varnothing$. Then binary costs $w_{ij}(a, b)$ are also projected on neighboring values (j, b) of i and the binary cost functions w'_{ij} are removed from the CFN $P_{|i=a}$.

Property 1. *The CFN $P_{|i=a} = (X, D', W')$ which is obtained from P by a variable assignment $(i = a)$ satisfies t he following properties:*

a) $w'_i(a) = 0$.
b) *for every variable j, there is not cost function w'_{ij} connecting i and j.*
c) *for every value (j, b) such that $\exists w_{ij}$ in P, we have $w'_j(b) = w_j(b) + w_{ij}(a, b)$.*

Figure 1 lists all the situations that can happen after a variable assignment $(i = a)$ for the values (j, b) of a neighbor variable j. It shows that, except for the two cases that will be considered later in Corollary 2, the deleted/non deleted status of (j, b) remains unchanged in Bool$(P_{|i=a})$. To show this, we first prove the following Corollary of Property 1:

Corollary 1. *The CN Bool$(P_{|i=a})$ satisfies the following*

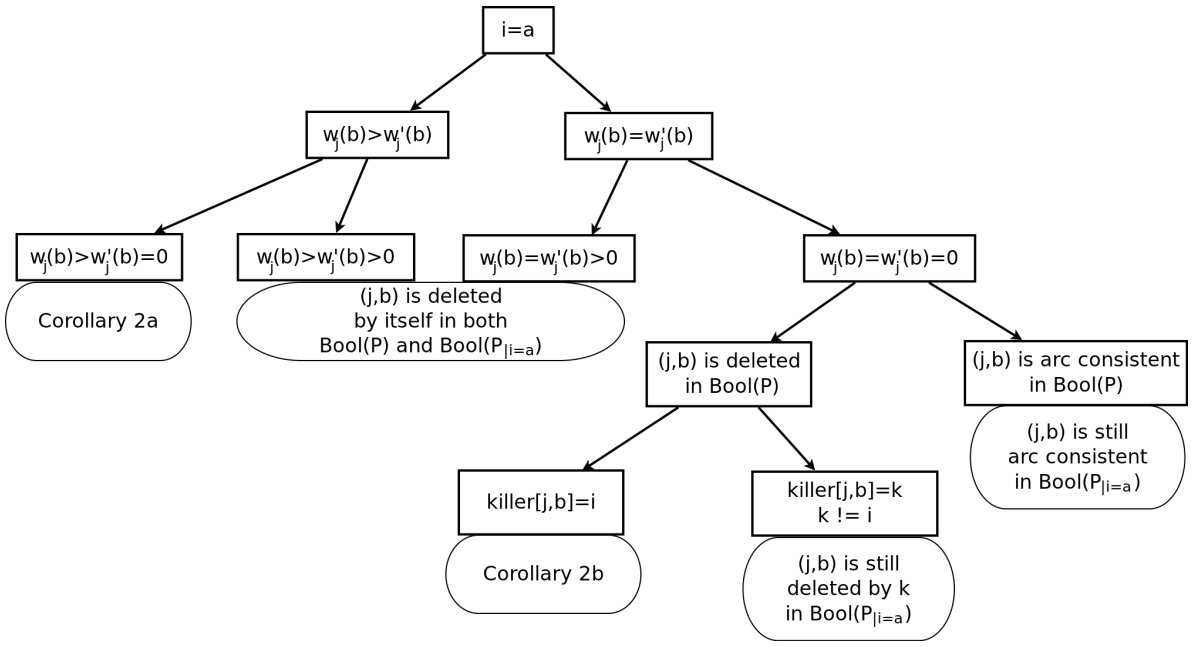

Fig. 1. All possible situations that can happen after a variable assignment

properties:

a) *for every variable $j \neq i, j$ is arc consistent with i.*
b) *(i, a) is an arc consistent value in Bool$(P_{|i=a})$.*

Proof. a) The constraints of Bool$(P_{|i=a})$ are directly defined from the cost functions of $P_{|i=a}$. From Property 1b), we know that there does not exist any w'_{ij} in $P_{|i=a}$ where $j \neq i$. Therefore, there also does not exists any constraint $\overline{w'}_{ij}$ in Bool$(P_{|i=a})$. As a result, j is arc consistent with variable i.

b) In Bool$(P_{|i=a})$, (i, a) cannot be deleted because of its positive cost since $w'_i(a) = 0$ according to Property 1a). Moreover, (i, a) cannot be killed by any other variable j because no constraint links i to other variables in Bool$(P_{|i=a})$.

□

Considering the various the cases where the status of (j, b) does not change, most are straightfoward. The only non obvious case is when $w'_j(b) = w_j(b) = 0$ and (j, b) is arc consistent in Bool(P). This implies that (j, b) is arc consistent with every variable $k \neq i$ in Bool(P). Because the variable assignment does not change cost functions w_{jk}, (j, b) will still be arc consistent with k in Bool$(P_{|i=a})$. In addition, Corollary 1a) indicates that (j, b) is also arc consistent with i in Bool$(P_{|i=a})$. Therefore, being arc consistent with every variable and having a zero unary cost, (j, b) is an arc consistent value in Bool$(P_{|i=a})$ (and it has the same status as in Bool(P)).

Then, as shown in Figure 1, there are two cases where the status of values may change compared to the status in Bool(P), requiring to update Bool$(P_{|i=a})$. This is proved in Corollary 2. In all other cases, as we saw, the status of values in Bool$(P_{|i=a})$ remains the same as in Bool(P).

Corollary 2. *Each variable assignment $(i = a)$ in P can generate both:*

a) *the removal of values (j, b) in Bool$(P_{|i=a})$ such that there exists w_{ij} in P and $w'_j(b) > w_j(b) = 0$.*
b) *the restoration of values (j, b) in Bool$(P_{|i=a})$ which were deleted in Bool(P) by $\overline{w}_{ij}$ and such that $w'_j(b) = 0$.*

Proof. a) From Property 1c), unary costs of values (j, b) that are neighbour with i in P may only increase or remain unchanged in $P_{|i=a}$. Some may therefore go from a zero to a non-zero cost. In this case, the corresponding values are deleted in Bool$(P_{|i=a})$. From the dynamic AC point of view, this can be considered as a restriction, with the addition of a unary constraint on j (i.e., domain restriction).

b) If (j, b) is a value removed in Bool(P) because of $\overline{w}_{ij}$ (killer$(j, b) = i$), this implies that $w_j(b) = 0$ (otherwise, we would have killer$(j, b) = j$). According to Property 1c), the unary cost of (j, b) in $P_{|i=a}$ can remain unchanged, i.e.; $w'_j(b) = 0$, if $w_{ij}(a, b) = 0$. Such a value with zero cost cannot be killed by itself in Bool$(P_{|i=a})$. Furthermore, i is no longer a reason to delete (j, b) because as Corollary 1a) states, (j, b) is arc consistent with i. Hence, (j, b) can become viable in Bool$(P_{|i=a})$. For the dynamic AC point of view, this corresponds to the relaxation of a unary constraint on j (domain relaxation) and (j, b) needs to be considered as restorable.

□

In summary, the change in a cost function network P caused by variable assignments can lead to both the restoration and the removal of values in the CN Bool(P). DnAC can perform

Algorithme 2 : algorithm updating Bool(P) wrt a variable assignment

```
1  Procedure assign(i, a)
2      D_i = D̄_i = {a};
3      UnaryProject(i);
4      foreach w_ij do
5          foreach b ∈ D_j do
6              if w_j(b) = 0 and w_j(b) + w_ij(a, b) > 0 then
7                  remove b from D̄_j;
8                  add j into Q_AC;
9              w_j(b) ← w_j(b) + w_ij(a, b);
10             if w_j(b) = 0 and killer(j, b) = i then
11                 restore(j, b);
12         remove w_ij and w̄_ij;
13     relaxation(RL);
14     AC-revise(Q_AC);
15 Procedure relaxation(RL)
16     while RL ≠ ∅ do
17         i ← RL.pop(); flag ← false;
18         foreach w_ij do
19             foreach b ∈ D_j − D̄_j s.t. killer(j, b) = i do
20                 if ∃a ∈ restored[i] s.t. w_ij(a, b) = 0 then
21                     restore(j, b);
22                     flag ← true;
23         restored[i] ← ∅;
24         if flag then Q_AC ← Q_AC ∪ {j | w̄_ij ∈ W̄};
25 Procedure restore(j, b)
26     add b into D̄_j and restored[j];
27     killer(j, b) = nil;
28     add j into RL;
```

this combined work by using two dedicated queues Q_{AC} and RL, the former for propagating value removals and the second for value restorations. Q_{AC} stores variables whose domain has been reduced while RL stores variables whose domain has been relaxed.

Procedure assign(i, a) in Algorithm 2 is used to update P and Bool(P) when assigning $(i = a)$. The unary cost of (i, a) and binary costs of pairs of values $((i, a), (j, b))$ are respectively projected on $w_\varnothing$ and on $w_j(b)$ (line 3, 9). Only neighboring values (j, b) of i are considered (line 5). New values of non-zero cost (line 6) will be removed from Bool(P) (line 7). The removal of (j, b) can further lead to the removal of other values. Thus, j is pushed into queue Q_{AC} for the future revision for AC (line 8). Conversely, if (j, b) has been removed by i and still has a zero cost (line 10), it is restorable and will be restored (line 11). This is done by Procedure restore(j, b) at line 25 which adds b into array restored$[j]$ and adds j into queue RL for the future propagation of value restorations. Array restored$[j]$ contains values of i that have just restored in Bool(P) and wait for being propagated. After defining all values to be removed and restored because of the value assignment for i, the algorithm will do the propagation for value restorations and then for value removals (line 13, 14 respectively).

AC-revise(Q_{AC}) is simply the usual instrumented AC enforcement used in VAC while relaxation(RL) aims to propagate value restorations in Bool(P). At each iteration of Procedure "relaxation" (line 15), a variable i is popped from RL. $i \in RL$ means that some value of i have been restored. The restored values of i can provide AC supports (line 20) for neighboring values (j, b) which have been deleted due to i (line 19). In this case, (j, b) will be restored (line 21) and "flag" is activated in order to inform that i need to be rechecked for AC later by adding its neighbour variables j into queue Q_{AC} (line 24).

When the search backtracks, the AC-closure of Bool(P) can be simply rebuilt via the restoration of the justification system "killer" through trailing. Indeed, if killer$(i, a) = nil$, this means that (i, a) was consistent in the old Bool(P). In this case, (i, a) will be set as an available value in $\overline{D}_i$. Conversely, if killer$(i, a) = j$ or i, this means that (i, a) was removed because of constraint $\overline{w}_{ij}$ or $\overline{w}_i$. In this case, (i, a) will be removed from $\overline{D}_i$.

A. Example

Consider the CFN P in Figure 2(a). It has four variables i, j, k and l, with domains $D_i = D_k = D_l = \{a, b, c\}, D_j = \{a, b\}$, and four cost functions $w_{ik}, w_{il}, w_{jk}, w_{jl}$. Values are represented as vertices. Numbers displayed besides values represent non-zero unary costs of values. An edge between two vertices indicates that the corresponding pair of values has a binary cost of 1. Zero costs are not represented. Suppose that P is the CFN obtained at some node of the tree search. The AC-closure of Bool(P) for this node is represented in Figure 2(b). Forbidden values are shown as crossed-out and edges represent *forbidden* pairs. The dotted arrows represent the killer data-structure for removed values, pointing to the variable that offered no valid support. A removed value without any justification arrow means that the value is deleted because of its own positive unary cost.

Now we will show how Bool(P) is updated in two cases: a domain restriction $(i \neq a)$ and a variable assignment $(i = a)$.

In the case $(i \neq a)$, DynVAC will propagate the domain reduction of i to neighboring variables k, l. We observe that (i, a) was also deleted in Bool(P) of the parent node. It did not support any value. Thus, the removal of (i, a) does not lead to any support loss. The propagation procedure stops here. We obtain a new Bool(P), which is identical to the old one in Figure 2(b). In this favorable case, DynVAC has almost nothing to do to construct Bool$(P_{|i \neq a})$.

Consider now the case of a variable assignment $(i = a)$. Only the pair $((i, a), (l, c))$ has positive cost and this binary cost will be projected to $(l, c) : w'_l(c) = 1$. Then cost functions w_{ik}, w_{il} are removed from P. The network of $P_{|i=a}$

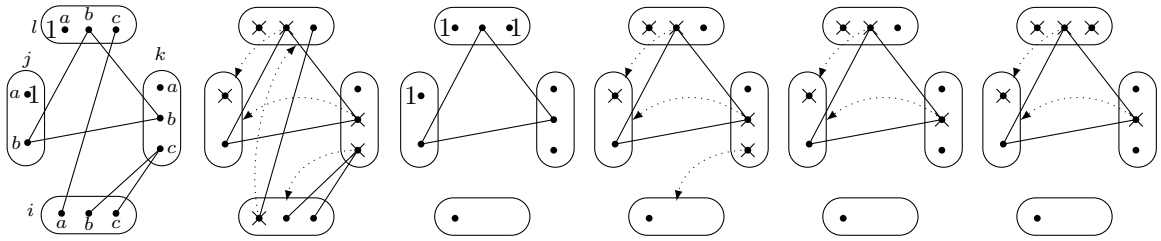

Fig. 2. Updating Bool(P) in the case of a variable assignment

is presented in Figure 2(c). Similarly, following assignment, values $(i, b), (i, c)$ and constraint $\overline{w}_{ik}, \overline{w}_{il}$ will be removed from Bool(P) and value (i, a) will be set as consistent in Bool$(P_{|i=a})$ (Figure 2(d)). In the neighborhood of i, only value (k, c) is restorable because it was removed due to $\overline{w}_{ij}$ but $\overline{w}_{ij}$ has just been removed from Bool(P). Thus, (k, c) is restored (Figure 2(e)). The restoration of (k, c) does not make any other value restorable. At the same time, in the neighborhood of i, only value (l, c) has an increased unary cost. (l, c) will be deleted in Bool$(P_{|i=a})$ because of its positive cost (Figure 2(f)). The domain of l becomes empty and VAC can project unary costs of 1 to $w_\varnothing$. We observe that the result of enforcing AC on the subproblem of Bool(P) defined by variables j, k, l (for constraints $\overline{w}_{jl}, \overline{w}_{jk}, \overline{w}_{kl}$) is preserved and that a variable wipeout is detected early.

V. Experiments

In this section, we will compare the efficiency of VAC when being maintained during search in three different ways:

- Static VAC (VAC): Bool(P) is always rebuilt from scratch whenever a new iteration of VAC or a branching operation is performed.
- Normal Dynamic VAC (norDVAC): Bool(P) is maintained incrementally inside each node of the search (betwwen VAC iterations) but it is rebuilt from scratch when the search makes a branching operation.
- Full DynVAC (fulDVAC): Bool(P) is incrementally maintained both inside each node of the search and when a branching operation is performed.

As described in [6], we remind the reader that when VAC is enforced in practice, iterations are stopped when the increase of the lower bound $w_\varnothing$ becomes less than a threshold ε. This is called VAC$_\varepsilon$. When maintained during search, two different ε thresholds are used. A small value ε_r is used at the root for maximum results. Then a larger value ε_s is used during search.

Furthermore, to accelerate VAC enforcing, [6] suggest to try to collect largest costs first by using relaxations of Bool(P) that only forbids tuples with sufficiently large costs, above a threshold θ. This is denoted as Bool$(P)_\theta$. VAC is enforced with a decreasing sequence of thresholds $(\theta_1, \theta_2, ...\theta_k)$ with $\theta_k = \varepsilon$.

In VAC and Normal DynVAC, θ is reinitialized to θ_1 when the search makes a branching operation (i.e, goes down in the search tree) or when the search backtracks. In Full DynVAC however, in order to incrementally maintain DynVAC during search between nodes, θ is incrementally updated when the search makes a branching operation and is restored by "trailling" when the search backtracks.

Our experiments use a set of benchmarks (in wcsp format) that are collected from different resources:

- maxcsp, planning, celar, tag08, warehouse, bep are extracted from the Cost Function Library (CFLib)[2]
- GeomSurf-7 are collected from Computer Vision and Pattern Recognition (CVPR) OpenGM2 benchmark[3]
- Coloring consists of unsatisfiable binary CN instances with constraints defined in extension representing graph coloring problems.
- ImageAlignment and ProteinFolding are taken from 2011 Probabilistic Inference Challenge.
- CPD are instances of computational protein design problems [3], [17].

Following the result of previous DynVAC experimentations for preprocessing in [16] that showed that DynVAC does not improve VAC on problems with boolean domains, we excluded problems with only boolean variables. The resulting very wide set of benchmarks represents a total of more than 1,000 instances coming from various application areas.

In our experiments, we set $\varepsilon_r = 1$, $\varepsilon_s = 1000$ for problems with very large costs (such as tag08, warehouse, CPD, GeomSurf-7), and $\varepsilon_r = 1$, $\varepsilon_s = 100$ for problems with large cost such as ImageAlignment. For the remaining problems, we use $\varepsilon = 0.0001$, $\varepsilon_S = 0.1$. The same thresholds are used in all algorithms. All experiments are executed on the same hardware using AMD Opteron(tm) 6176 processors.

Table I reports the mean run-time (in seconds) for solving these different categories of benchmarks by enforcing static VAC, normal DynVAC and full DynVAC. Each line corresponds to a category of benchmarks where the number of instances (#inst) and the mean values of problem size (n: number of variables, d: domain size, e: number of cost functions) are given. We also report the mean graph densities

[2]https://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/?root=costfunctionlib
[3]http://hci.iwr.uni-heidelberg.de/opengm2

TABLE I
AVERAGE SOLVING TIME PER CATEGORY OF BENCHMARKS WHEN MAINTAINING VARIANTS OF VAC IN SEARCH.

Categories	# inst	mean n	mean d	mean e	mean dens	Maint VAC	Maint norDVAC	Maint fulDVAC	Speedup /norDVAC	Speedup /VAC
coloring	22	120	4	1203	0.112	**16.70** *17*	26.35 *17*	18.64 *17*	1.41	0.90
GeomSur7	300	505	7	2140	0.011	6.20 *300*	9.36 *300*	**1.08** *300*	8.67	5.74
maxcsp	340	31	10	129	0.213	**58.76** *340*	78.75 *340*	58.99 *340*	1.34	1.00
planning	76	240	14	14746	0.307	20.22 *75*	11.69 *75*	**9.56** *76*	1.22	2.11
Matching	4	19	19	166	0.441	**104.03** *4*	367.40 *3*	281.99 *3*	1.30	0.37
celar	46	137	36	1346	0.507	364.27 *38*	337.31 *37*	**290.78** *38*	1.16	1.25
tag08	82	185	62	8003	0.179	**14.09** *81*	31.42 *80*	27.39 *81*	1.15	0.51
ImageAlignment	10	191	70	1819	0.123	18.58 *10*	23.74 *10*	**3.23** *10*	7.36	5.76
warehouse	57	216	78	18739	0.237	164.26 *54*	289.30 *51*	**90.49** *55*	3.20	1.82
CPD	35	50	148	1561	0.527	**39.30** *24*	46.79 *24*	46.72 *24*	1.00	0.84
bep	4	18	168	160	0.500	3.20 *3*	2.62 *3*	**2.56** *3*	1.02	1.25
ProteinFolding	31	334	221	1583	0.384	27.68 *30*	48.58 *30*	**27.12** *30*	1.79	1.02

per category (dens). The graph density of an instance is defined as the ratio of its number of cost functions with the number of edges in a complete graph.

The three following columns reprots results for our three algorithms. Each box here contains two numbers. The italic number in the bottom right corner represents the number of instances solved by the algorithm in less than one hour while the number in the top left corner represents the mean value of the run-time. This mean is computed only over problems that are solved in less than 1 hour by all three variants of VAC. The best results are in bold. Finally, the last two columns give the speedups obtained by full DynVAC compared respectively to normal DynVAC and VAC.

From these last two columns, we observe that full DynVAC outperforms normal DynVAC on all categories of benchmarks and is more efficient than VAC in most cases.

Compared to normal DynVAC, Full DynVAC shows very significant speed-ups on the categories GeomSur-7, ImageAlignment and warehouse (speed-ups of 8.7, 7.4 and 3.2 respectively). For the other categories (coloring, maxcsp, planning, matching, celar, tag08, ProteinFolding), the speed-up of full DynVAC goes from a factor 1.79 to a factor 1.15. Finally, it has almost the same efficiency on two categories (CPD and bep).

To explain these differences, we have to remember what makes full DynVAC different from normal DynVAC. The difference between full DynVAC and nornal DynVAC comes from the way they update Bool(P) after a branching operation. In full DynVAC, only the variables of Bool(P) that are directly linked to the branching variable (its neighborhood) are modified and restorations are propagated to the other variables before arc consistency is rechecked on the modified variables. In normal DynVAC, the whole arc consistency process is redone from scratch on the reinitialized Bool(P). It is therefore expected that the graph density of the instance solved will have a strong impact on efficiency. If the graph density is low, very few variables will appear in the neighborhood of the modidifed variable and full DynVAc should save a lot of work compared to normal Dyn VAC.

Another important factor for efficiency is the domain size. If we restore a few values incrementally in a domain that was initially large, the cost of repropagating AC can be much lower than if resetting the whole domain from scratch. If we restore a few values in a domain that was initially small, AC will not be significantly faster than if resetting the whole initial domain.

Overall, it is therefore expected that the greatest savings will be obtained by full DynVAC on problems with low graph densities and large domains. If we analyse the characteristics of the categories of problems on which full DynVAC behaves very well, we effectively observe that GeomSur-7 has a very low graph density (0,011 –lowest density of the set of benchmarks), ImageAlignement has a low density (0.123) and a great domain size (70), and warehouse has a medium density (0.237) but is characterized by a large domain size (78). Along the same line, by observing the categories on which full DynVAC is only slightly faster than normal DynVAC, we see that they are characterized by a high graph density (0.527

for CPD and 0.5 for bep).

This impact of density and domain size on the performance of full DynVAC seems to be confirmed on the other categories of problems. Coloring, for instance, has a very low density (0.112), and we would expect a very good performance of full DynVAC. However, the moderately improved performance of full DynVAC on this coloring category ('only' 1.4 times faster than normal DynVAC) can be explained by the very small mean domain size (4).

We now compare the efficiency of full DynVAC to the efficiency of static VAC. We observe that full DynVAC is better than static VAC on 7 of the 12 categories. The categories GeomSur-7, ImageAlignment and warehouse, where full DynVAC was significantly better than normal DynVAC, remain very favorable to full DynVAC when compared to static VAC. This tends to show that the criteria of density and domain size remain true when comparing to static VAC. We also observe that category planning is another example where full DynVAC is significantly faster than static VAC (by a factor 2.11).

Overall, if we compare the efficiency of the two variants of DynVAC to static VAC we observe that when normal DynVAC is faster than static VAC then full DynVAC is even faster. This however happens for only two categories: planning and bep, whereas full DynVAC is faster on 7 categories.

In summary, incrementally maintaining DynVAC during search is in general very beneficial in comparison with restarting from scratch after each branching operation and graph density and domain size are the main criteria that have a visible impact on the performance of full DynVAC.

VI. Conclusion

Following [16] that introduced the Dynamic VAC algorithm combining the idea of dynamic arc consistency algorithms with the iterative VAC algorithm in order to efficiently maintain arc consistency in the CN Bool(P) during VAC enforcing for preprocessing, we show that Dynamic VAC can be also be maintained during search.

In this situation, two approaches can be considered: either Dynamic VAC can be just maintained at each node of the search tree or the incrementality of Dynamic VAC can be exploited also to account for the small problem changes that occur following branching operations in the search tree.

By exploiting both incremental changes caused by branching operations as well as incremental changes of EPTs during sucessive iterations of VAC, the new fully incremental method outperforms both the direct application of dynamic VAC and the usual maintenance of static VAC on a variety of problems. This is especially true for problems having small graph densities and large domains.

There are still more opportunities to accelerate algorithms maintaining VAC during search. Indeed, VAC does not require to enforce AC on Bool(P) but only to detect if the AC closure is empty or not. Therefore, it is sufficient to identify a single viable value in each domain to conclude. This simplified problem has been solved using so-called Lazy AC algorithms [18] that could also be injected in VAC algorithms to increase their efficiency.

References

[1] T. Schiex, H. Fargier, and G. Verfaillie, "Valued constraint satisfaction problems: hard and easy problems," in *Proc. of the 14^{th} IJCAI*, Montréal, Canada, Aug. 1995, pp. 631–637.

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

[3] S. Traoré, D. Allouche, I. André, S. de Givry, G. Katsirelos, T. Schiex, and S. Barbe, "A new framework for computational protein design through cost function network optimization," *Bioinformatics*, vol. 29, no. 17, pp. 2129–2136, 2013.

[4] R. Dechter and R. Mateescu, "And/or search spaces for graphical models," *Artificial intelligence*, vol. 171, no. 2, pp. 73–106, 2007.

[5] S. de Givry, T. Schiex, and G. Verfaillie, "Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP," in *Proc. of the National Conference on Artificial Intelligence, AAAI-2006*, 2006, pp. 22–27.

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

[7] J. Larrosa, F. Heras, and S. de Givry, "A logical approach to efficient max-sat solving," *Artif. Intell.*, vol. 172, no. 2-3, pp. 204–233, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2007.05.006

[8] C. M. Li, F. Manya, and J. Planes, "New inference rules for max-sat." *J. Artif. Intell. Res.(JAIR)*, vol. 30, pp. 321–359, 2007.

[9] H. Wang and K. Daphne, "Subproblem-tree calibration: A unified approach to max-product message passing," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 190–198.

[10] M. C. Cooper and T. Schiex, "Arc consistency for soft constraints," *Artificial Intelligence*, vol. 154, no. 1-2, pp. 199–227, 2004.

[11] 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 19^{th} IJCAI*, Edinburgh, Scotland, Aug. 2005, pp. 84–89.

[12] 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.

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

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

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

[16] H. Nguyen, T. Schiex, and C. Bessiere, "Dynamic virtual arc consistency," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 98–103.

[17] D. Allouche, I. Andr, S. Barbe, J. Davies, S. de Givry, G. Katsirelos, B. O'Sullivan, S. Prestwich, T. Schiex, and S. Traor, "Computational protein design as an optimization problem," *Artificial Intelligence*, vol. 212, pp. 59–79, 2014.

[18] T. Schiex, J.-C. Régin, C. Gaspin, and G. Verfaillie, "Lazy arc consistency," in *Proc. of AAAI'96*. Portland, OR: AAAI Press, Aug. 1996.