Dynamic Virtual Arc Consistency

Hiep Nguyen[1], Thomas Schiex[1], and Christian Bessiere[2]

[1] Unité de Biométrie et Intelligence Artificielle, INRA, Toulouse, France
[2] University of Montpellier, France

Abstract. Virtual Arc Consistency is a recent local consistency for processing cost function networks that exploits a simple but powerful connection between classical constraint networks and cost function networks. The algorithm enforcing virtual arc consistency iteratively 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.

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 [4], captures problems such as weighted MaxSAT/CSP or Maximum Probability Explanation in probabilistic networks. Depth First Branch and Bound search has been largely used to tackle such problems. It has a reasonable space complexity but requires good 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 CFN. Enforcing is done using so-called *Equivalence Preserving Transformations* (EPTs, [2]) which extend usual CSP local consistency operations. EPTs move costs between cost functions while keeping the problem equivalent. They may eventually increase the cost function of empty scope (a constant) to a non naive value. This value provides a lower bound on the optimum cost which can be maintained during branch and bound search.

Virtual arc consistency (VAC), introduced in [1], relies on pre-planned applications of EPTs built from the result of enforcing classical arc consistency (AC) on a constraint network called Bool(P) which forbids combinations of values with non zero costs in the original CFN P. VAC dominates all previously defined chaotic local consistencies, and it can be approximately enforced with a low order polynomial time iterative algorithm. Maintaining VAC during tree search has effectively allowed to close two difficult instances of Radio Link Frequency Assignment instances. Each iteration of VAC incrementally modifies the network P. The next iteration therefore proceeds by enforcing classical AC on a slightly relaxed version of Bool(P). This situation, where AC is iteratively enforced on incrementally modified versions of a constraint network, has been previously considered in dynamic arc consistency algorithms for dynamic CSPs [3]. In this paper we adapt ideas from dynamic AC in the last phase of VAC. We observe that this new version frequently provides significant speedups. This may be, as far as we know, one of the first successful application of dynamic AC algorithms.

2 Background

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. $w_S : (S) \rightarrow [0..m]$ 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 $Val_P(t) = \bigoplus_{w_S \in C} w_S(t[S])$ where $t[S]$ is the projection of t on S. 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. In this paper, we restrict ourselves to binary CFNs.

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 possible increase in the lower bound $w_\varnothing$ on the optimal cost. Enforcing is done by using equivalence-preserving transformations (EPTs) which shift costs between cost functions. There are 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 $w_\varnothing$.

In a classical binary CSP, represented as a CFN with $m = 1$ (cost 1 being associated to forbidden tuples), a value (i, a) is 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.

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 AC closure of Bool(P) is non-empty.*

If P is not VAC, there exists a sequence of EPTs which leads to an increase of $w_\varnothing$ when applied on P. VAC enforcing uses an iterative three-phases process. The first phase 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 $\mathsf{killer}(\mathsf{(i,a)}) = j$ and we delete the value. If no domain wipe-out occurs, P is VAC and we stop. Otherwise, phase 2 identifies the subset of value deletions that are necessary to produce the wipe-out and stores them in a queue R by tracing back the propagation history defined by killer. Phase 2 also 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. 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 quantum 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) 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 [1], page 465).

Phase 3 of VAC modifies the original CFN by applying the EPTs defined by $k()$ 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 Project from w_{ij}. This requires to first extend a cost $k_{ij}(i, a) \times \lambda$ from the invalid supports (i, a) to w_{ij}. The result of this phase is a new problem P', equivalent to P but with an increased lower-bound $w_{\varnothing}$. 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 (DnAC) to enforce VAC. DnAC algorithms is to maintain AC in CSP problems after each constraint addition or retraction.

Several algorithms have been proposed for DnAC. In this paper, we will use AC/DC2 [5]. AC/DC2 uses the data structure *justification*(i, a) to remember the cause of deletion for (i, a). In the initialization stage, only values which have been deleted because of the removed constraint c_{ij} are considered as candidates for restoration. In the propagating stage, a variable i having restored values can check neighboring values (j, b) for restorability just if they have been removed due to the lost of support on the constraint c_{ji} (known through $justification$). In a last stage, all restored values need to be checked again for arc consistency.

3 Dynamic VAC algorithm

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 an AC/DC2 version based on AC2001 instead of AC3. This also has the advantage that the *justification* data-structure of AC/DC-2 is provided for free by the killer array in VAC. DnAC algorithms are usually applied after each constraint removal or addition. In the case of VAC, at each iteration, a series of modifications of Bool(P) can occur during Phase 3. Let us denote by P^t the CFN at the beginning of iteration t. The problem P^t has cost functions w_i^t and w_{ij}^t. We show that the global effect of all EPTs on Bool(P) in Phase 3 can be captured as a list of relaxations only, at the unary and binary levels.

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

Corollary 1. *The EPTs applied in the phase 3 of VAC, transforming Bool*(P^t) *into Bool*(P^{t+1})*, generate only the following types of relaxations 1) values* (i, a) *that become authorized (*$w_i^t(a) > w_i^{t+1}(a) = 0$*). 2) pairs* $((i, a), (j, b)$ *that become authorized (*$w_{ij}^t(a, b) > w_{ij}^{t+1}(a, b) = 0$*).*

Therefore, the DnAC algorithm can be specialized. The restoration protocol will consist of 3 stages , as in AC/DC2 (Algorithm 1). We denote by $\overline{D}_i$ the domain of variable i in the final justified partial AC closure obtained after Phase 1.

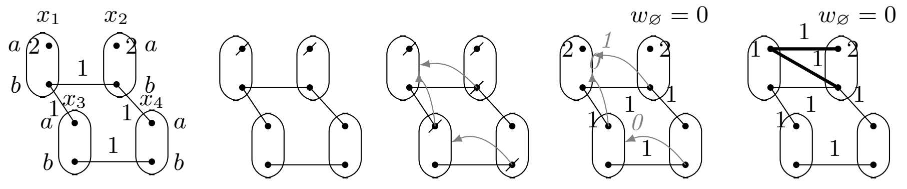
The **initialization stage** scans all the values in the queue R to identify which values should be restored. The wipe-out variable i_0 is processed separately

Algorithm 1: Update Bool(P) at iteration t

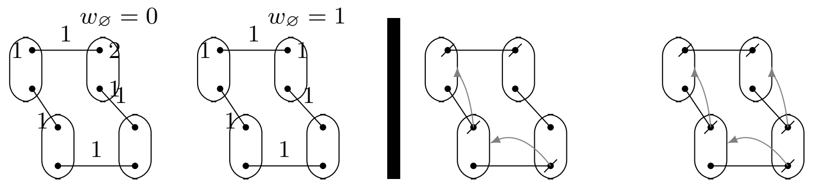
```
1  Procedure Initialization
2      foreach $(j,b) \in R$ do
3          $i \longleftarrow$ killer $[j,b]$;
4          foreach $a \in D_i - \overline{D}_i$ do
5              if $(w_i^t(a) > 0) \wedge (w_i^{t+1}(a) = 0)$ then Restore$(i,a)$;
6              if $b \notin \overline{D}_j \ \wedge \ w_i^{t+1}(a) = 0 \ \wedge \ w_{ij}^{t+1}(a,b) = 0$ then Restore$(j,b)$;
7      foreach $a \in D_{i_0}$ s.t. $w_{i_0}^t(a) > 0 \ \wedge \ w_{i_0}^{t+1}(a) = 0$ do Restore$(i_0,a)$;
8  Procedure Restore$(i,a)$
9      add $a$ into $\overline{D}_i$, restored$[i]$ and add $i$ into $RL$;
10     killer $[i,a] \leftarrow$ nil;
11 Procedure Propagation
12     while $RL \neq \varnothing$ do
13         $i \leftarrow RL.pop()$;
14         foreach $w_{ij} \in W$ do
15             foreach $b \in D_j - \overline{D_j}$ s.t. killer $[j,b]= i$ do
16                 if $\exists a \in$ restored$[i]$ s.t. $w_{ij}^{t+1}(a,b) = 0$ then Restore$(j,b)$;
17         restored$[i] \leftarrow \varnothing$;   $Q_{AC} \leftarrow Q_{AC} \cup \{j \mid w_{ij} \in W\}$
```

(line 7). 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 neighbors of the variables whose domain has been extended. Each such variable i can restore a value (j,b) if it was deleted due to w_{ij} (line 15) and is now supported by a restored value in i (line 16). The **filtering stage** must eliminate the restored values (i,a) which are not arc consistent on some constraint $\overline{w}_{ij}$ and must properly set the associated killer (i,a) to j. This is precisely what is achieved by Phase 1 of VAC. Hence, we integrated this stage into phase 1 by adding the neighbor variables of variables having restored values into the revision propagation queue Q_{AC} (line 17).

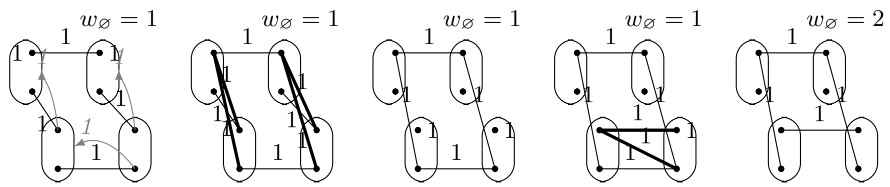
Consider the binary CFN in Fig.a. Only non-zero costs and edges associated with non-zero binary costs are displayed. In Bool(P) (Fig.b), forbidden values are shown as crossed-out and edges represent forbidden pairs. The revision order in Phase 1 is $(w_{13}, w_{34}, w_{12}, w_{24})$. Phase 1 stops when x_2 is wiped-out after revising w_{13}, w_{34}, w_{12} (Fig.c). The gray arrows point to the variable offering no valid support and the associated italic numbers represent $k(i,a)$. In Phase 2 (Fig.d), the deletion of (x_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 $\lambda = 1$. Applying identified EPTs, Phase 3 transforms P into an equivalent problem P^2 with $w_\varnothing = 1$ (Fig.e). Extended costs are shown in bold. To update Bool(P) in Fig.c, we consider $((x_1,a),(x_2,a),(x_2,b))$ since only w_{12} as been modified by EPTs in phase 3. Only (x_2,b) is restored because it has a zero-cost and a



(a) P (b) Bool(P) (c) Phase 1 (d) Phase 2 (e) Phase 3



(f) Phase 3 (g) Phase 3 (h) Updated Bool(P) (i) Phase 1



(j) Phase 2 (k) Phase 3 (l) Phase 3 (m) Phase 3 (n) Phase 3

support (b, b) on w_{12}. This restoration does not lead to further restorations. The constraints of the updated Bool(P^2) are directly defined by P^2. The updated result (Fig.h) has 2 extra deleted values with associated killer. The next phase 1 starts from this updated Bool(P^2). The final problem with $w_\varnothing = 2$ is VAC.

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[3]. For each problem, as in [1], we enforce a limited version of VAC that stops iterating as soon as the increase in $w_\varnothing$ becomes less than $\varepsilon = 0.05$.

The following table 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. 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. However, DynVAC is significantly slower than VAC (respectively 7 and 4 times) for all the maximum clique problems categories we tested: *protein_mc* and *dimacs_mc*. On these problems, we noticed that each iteration leads to the useless restoration of many values in cascade which will again be uselessly deleted

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

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_mc	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_mc	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	$\mathbf{7.24}\times10^6$	**66**	719	7.25×10^6	114.17	790

in the next iteration. In order to improve the efficiency of DynVAC we have used the variable-based revision heuristics proposed in [6]. The corresponding results are presented in the last column of the above table. They show that the heuristics 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.

5 Conclusion

We have proposed an incremental approach for enforcing VAC in CFNs. It combines the idea of DnAC algorithms with the iterative VAC algorithm in order to efficiently maintain arc consistency in the CSP Bool(P) during VAC enforcing. The new algorithm, DynVAC, provides a lower bound of the same quality level as the static VAC algorithm but is faster than static VAC on many problems. 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.

References

1. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artificial Intelligence 174, 449–478 (2010)
2. Cooper, M.C., Schiex, T.: Arc consistency for soft constraints. Artificial Intelligence 154(1-2), 199–227 (2004)
3. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: Proc. of AAAI'88. pp. 37–42. St. Paul, MN (1988)
4. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: Proc. of the 14th IJCAI. pp. 631–637. Montréal, Canada (Aug 1995)
5. Surynek, P., Barták, R.: A new algorithm for maintaining arc consistency after constraint retraction. In: Proc. Principles and Practice of Constraint Programming – CP 2004. pp. 767–771. No. 3258 in LNCS, Toronto, Canada (2004)
6. Wallace, R., Freuder, E.: Ordering heuristics for arc consistency algorithms. In: Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence. pp. 163–163 (1992)