Dynamic virtual arc consistency
Thi Hông Hiêp Nguyên, Thomas Schiex, Christian Bessière

To cite this version:
Thi Hông Hiêp Nguyên, Thomas Schiex, Christian Bessière. Dynamic virtual arc consistency. 28th Annual ACM Symposium on Applied Computing, Mar 2013, Coimbra, Portugal. pp.6, 10.1145/2480362.2480384. hal-02745649

HAL Id: hal-02745649
https://hal.inrae.fr/hal-02745649
Submitted on 3 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 Frequence 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 \(t(S)\) 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 \(w_\emptyset\). 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 \(w_\emptyset\) on the optimal cost. Enforcing is done by using equivalence-preserving transformations (EPTs) which shift costs between cost functions. There are three basic EPTs. \(\text{Project}(w_{ij}, i, a, \alpha)\) moves an amount of cost \(\alpha\) from a binary cost function to a unary one. Conversely, \(\text{Extend}(i, a, w_{ij}, \alpha)\) sends an amount of cost \(\alpha\) from a unary cost function to a binary one. Finally, \(\text{UnaryProject}(i, \alpha)\) projects an amount of cost \(\alpha\) from a unary cost function to the nullary cost function \(w_\emptyset\).

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 \(\text{Bool}(P) = (X, D, \overline{W}, 1)\) is such that \(\exists w_S \in \overline{W} \iff \exists w_S \in W\), \(S \neq \emptyset\) and \(\overline{w}_S(t) = 1 \Rightarrow w_S(t) \neq 0\). A CFN \(P\) is virtual arc consistent (VAC) iff the AC closure of \(\text{Bool}(P)\) is non-empty.

If \(P\) is not VAC, there exists a sequence of EPTs which leads to an increase of \(w_\emptyset\) when applied on \(P\). VAC enforcing uses an iterative three-phases process. The first phase is an instrumented AC enforcing on the CSP \(\text{Bool}(P)\) 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 \(\text{killer}(i, a) = j\) 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 \(w_\emptyset\), denoted \(\lambda\), 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 \(\lambda\) 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 \((w_{ij}(b) = 0)\), the amount of cost that arrives at \((j, b)\) by \(\text{Project}\) is exactly the amount of cost that leaves \((j, b)\) by \(\text{Extend}\) (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 $k(j, b) \times \lambda$ from 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 $w_\emptyset$. Ultimately, VAC iterations enforce AC on a sequence
of slightly modified CSPs: $\text{Bool}(P)$, $\text{Bool}(P')$, $\ldots$ This motivates the use of ded-
cicated 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 $\text{justification}(i, a)$ 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 $\text{justification}$).
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 $\text{Bool}(P)$ 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 $\text{justification}$ 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
$\text{Bool}(P)$ 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 $w^t_i$ and $w^t_{ij}$. We show that
the global effect of all EPTs on $\text{Bool}(P)$ 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) $\forall (i, a) : w^t_i(a) \leq w^{t+1}_i(a)$. 2) $\forall (i, a)$ and $(j, b)$ : if $w^{t+1}_{ij}(a, b) \neq w^t_{ij}(a, b)$ then $(i, a)$ or
$(j, b)$ is deleted in the current justified partial AC closure of $\text{Bool}(P^t)$.

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

Therefore, the DnAC algorithm can be specialized. The restoration protocol will
consist of 3 stages, as in AC/DC2 (Algorithm 1). We denote by $D_i$ 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. $i \leftarrow$ killer $[j, b]$;
      4. foreach $a \in D_i - D_j$ do
         5. if $(w^i_t(a) > 0) \land (w^{i+1}_i(a) = 0)$ then Restore($i, a$);
         6. if $b \notin D_j \land w^{i+1}_i(a) = 0 \land w^{i+1}_{ij}(a, b) = 0$ then Restore($j, b$);
      7. foreach $a \in D_i$ s.t. $w^i_t(a) > 0$ and $w^{i+1}_i(a) = 0$ do Restore($i, a$);

2. Procedure Restore($i, a$)
   3. add $a$ into $D_i$, restored[$i$] and add $i$ into $RL$;
   4. killer $[i, a] \leftarrow$ nil;

3. Procedure Propagation
   5. while $RL \neq \emptyset$ do
      6. $i \leftarrow RL.pop();$
      7. foreach $w_{ij} \in W$ do
         8. foreach $b \in D_j - D_i$ s.t. killer $[j, b] = i$ do
            9. if $\exists a \in restored[i]$ s.t. $w^{i+1}_{ij}(a, b) = 0$ then Restore($j, b$);
      10. restored[$i$] $\leftarrow \emptyset$; $Q_{AC} \leftarrow Q_{AC} \cup \{j \mid w_{ij} \in W\}$

(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 $w_{ij}$ 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 $w_\emptyset$ with a maximal amount of cost $\lambda = 1$. Applying identified EPTs, Phase 3 transforms $P$ into an equivalent problem $P^2$ with $w_\emptyset = 1$ (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
support \((b, b)\) on \(w_{12}\). This restoration does not lead to further restorations. The constraints of the updated \(\text{Bool}(P^2)\) 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 \(\text{Bool}(P^2)\). The final problem with \(w_{\emptyset} = 2\) is VAC.

4 Experiments

In this section, we compare the efficiency of DynVAC and VAC used as preprocessing 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 \(w_{\emptyset}\) 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 \(\text{celar, tagsnp, warehouse}\) 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: \(\text{protein\_mc}\) and \(\text{dimacs\_mc}\). 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
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