Cohérence d'arc virtuelle dynamique

Hiep Nguyen, Thomas Schiex, Christian Bessière

Cohérence d'arc virtuelle dynamique

Hiep Nguyen[1] **Christian Bessiere**[2] **Thomas Schiex**[1]

[1] Unité de Biométrie et Intelligence Artificielle, INRA, Toulouse, France
[2] CNRS, Université de Montpellier, France
{hnguyen,tschiex}@toulouse.inra.fr bessiere@lirmm.fr

Résumé

La cohérence d'arc virtuelle (VAC) est une cohérence récente pour les réseaux de fonctions de coût (ou réseaux de contraintes pondérées). VAC exploite une relation simple mais puissante avec les réseaux de contraintes classiques. VAC a permis de clore des problèmes d'affectation de fréquences difficiles, et est capable de résoudre directement les réseaux de fonctions de coût sous-modulaires. L'algorithme pour établir VAC est un algorithme itératif qui résout une séquence de réseaux de contraintes classiques. Dans cet article, nous montrons que les techniques utilisées dans les algorithmes de cohérence d'arc dynamique peuvent être injectées dans l'algorithme itératif de cohérence d'arc virtuelle. Cette intégration donne une accélération importante.

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.

1 Introduction

L'analyse des modèles graphiques, et en particulier des réseaux de contraintes, est un problème important en intelligence artificielle. L'optimisation du coût combiné de fonctions de coût locales, problème central dans le cadre des CSP valués [11], permet de capturer des problèmes variés, tels que MaxSAT pondéré, CSP pondéré ou le problème de recherche d'une explication de probabilité maximum dans les modèles graphiques stochastiques (réseaux bayésiens, champs aléatoires de Markov). Elle a des applications en *allocation de ressources*, *enchères combinatoires*, *bioinformatique*, etc.

Les approches de programmation dynamique, basées sur l'élimination de variables ou sur les arbres de jonction, ont été largement utilisées pour résoudre de tels problèmes. Cependant, elles sont intrinsèquement limitées par leur complexité exponentielle en espace et en temps dans la largeur d'arbre (*treewidth*) des problèmes traités. Au contraire, la recherche arborescente en profondeur d'abord par "Branch and Bound" (séparation et évaluation) permet de conserver une complexité en espace raisonnable. Elle nécessite cependant de bons minorants (forts et peu coûteux) pour être efficace.

Ces dernières années, des minorants de qualité croissante ont été définis en appliquant des cohérences locales dans des Réseaux de Fonctions de Coût (ou CFN pour "Cost Function Networks"). Ces cohérences sont établies en appliquant itérativement des opérations appelées *Transformations Préservant l'Equivalence* (EPT, *Equivalence Preserving Transformations*, [7]) qui étendent les opérations de cohérence locale usuelles utilisées dans les CSP classiques. Les EPT déplacent des coûts entre des fonctions d'arité différente tout en préservant l'équivalence du problème. En déplaçant finalement des coûts vers une fonction d'arité nulle, les EPT sont capables de fournir un minorant du coût optimum qui peut être maintenu de manière incrémentale pendant la recherche arborescente.

Les cohérences locales traditionnelles des CFN telles qu'AC*, DAC*, FDAC* ou EDAC* [10] appliquent les EPT dans un ordre arbitraire. Par contre, la Cohérence d'Arc Virtuelle (VAC pour *Virtual Arc Consistency* [5, 6]) planifie une séquence d'EPT à appliquer. Elle le fait en se basant sur le résultat de l'établissement de la cohérence d'arc classique dans un réseau

de contraintes classique qui interdit toutes les combinaisons de valeurs de coût non nul. VAC est non seulement plus fort que ces cohérences locales mais est capable de résoudre les réseaux formés de fonctions de coût sous-modulaires. Une itération de VAC peut être appliquée via un algorithme polynomial d'ordre faible et VAC a permis de clore des problèmes d'affectation de fréquences difficiles [5]. Cependant, il est souvent trop couteux pour être utilisé de façon générale.

Dans cet article, l'efficacité de VAC est améliorée en exploitant son comportement itératif. Chaque itération de VAC demande d'établir la cohérence d'arc classique dans la version durcie du réseau. Néanmoins, chacun de ces réseaux est juste le résultat des modifications incrémentales effectuées par les EPT appliquées dans l'itération précédente. Cette situation, où la cohérence d'arc est établie itérativement dans les versions modifiées de manière incrémentale d'un réseau de contraintes, a été déjà considérée dans les algorithmes de Cohérence d'Arc Dynamique [1, 3] pour les CSP dynamiques [8].

En pratique, nous observons que l'utilisation de la cohérence d'arc dynamique dans VAC accélère la recherche pour de nombreux problèmes. Ceci est peut-être l'une des premières applications réelles des algorithmes de Cohérence d'Arc Dynamique.

2 Préliminaires

2.1 Réseaux de Fonctions de Coût

Un réseau de fonctions de coût, ou CSP pondéré, est quadruplet (X, D, W, m) où X est un ensemble de n variables. Chaque variable $i \in X$ a un domaine $D_i \in D$ d'au plus d valeurs. Étant donné un sous-ensemble de variables $S \subseteq X$, on note $\ell(S)$ l'ensemble des n-uplets définis sur S. W est un ensemble de e fonctions de coût. Chaque fonction de coût $w_S \in W$ est définie sur un ensemble de variables S, appelé sa portée, et supposé différent pour chaque fonction de coût. Une fonction de coût w_S assigne un coût à chaque affectation de variables dans S, c'est à dire $w_S : \ell(S) \rightarrow [0..m]$ où $m \in \{1, ..., +\infty\}$. Le coût m représente un coût intolérable, associé à une valeur ou un n-uplet interdit. Les coûts sont combinés par l'addition bornée $\oplus$, définie par $a \oplus b = \min(a + b, m)$. Un coût b peut être soustrait d'un coût plus grand a en utilisant la soustraction $\ominus$ définie par : $a \ominus b = a - b$ si $a < m$ et m sinon. Le coût d'un n-uplet complet t est la somme des coûts $Val_P(t) = \bigoplus_{w_S \in W} w_S(t[S])$ où $t[S]$ est la projection de t sur S. Dans cet article, nous considérons des réseaux binaires, formés de fonctions d'arité au plus 2. Nous dénotons par w_i et w_{ij} les fonctions de coût unaires et binaires définies sur la variable i et sur les variables i, j respectivement. Nous faisons l'hypothèse qu'une fonction de coût unaire, notée w_i, existe pour toute variable i ainsi qu'une fonction de coût d'arité nulle notée $w_\varnothing$. Ce coût positif constant définit un minorant du coût de toutes les solutions.

L'établissement d'une cohérence locale donnée dans un CFN P consiste à transformer le CFN P en un problème P' équivalent à P : ($Val_P(t) = Val_{P'}(t)\ \forall t$) menant à un éventuel accroissement du minorant de coût optimum défini par $w_\varnothing$. Il s'appuie sur l'application des transformations préservant l'équivalence (EPT) qui déplacent les coûts entre les fonctions de portée différente. L'algorithme 1 introduit trois EPT élémentaires. Project(w_{ij}, i, a, α) déplace une quantité de coût α d'une fonction binaire w_{ij} vers une fonction unaire w_i, sur une valeur $a \in D_i$. Un appel à Extend(i, a, w_{ij}, α) effectue le travail inverse. Enfin, UnaryProject(i, α) projette une quantité de coût α d'une fonction unaire vers la fonction de coût d'arité nulle $w_\varnothing$.

Algorithme 1 : Les EPTs élémentaires

```
1 Procédure Project(w_ij, i, a, α)
2 |   w_i(a) ⟵ w_i(a) ⊕ α ;
3 |   foreach b ∈ D_j do  w_ij(a, b) ⟵ w_ij(a, b) ⊖ α;
4 Procédure Extend(i, a, w_ij, α)
5 |   foreach b ∈ D_j do  w_ij(a, b) ⟵ w_ij(a, b) ⊕ α;
6 |   w_i(a) ⟵ w_i(a) ⊖ α ;
7 Procédure UnaryProject(i, α)
8 |   foreach a ∈ D_i do  w_i(a) ⟵ w_i(a) ⊖ α;
9 |   w_∅ ⟵ w_∅ ⊕ α ;
```

Notez qu'un CSP binaire classique peut être représenté comme un CFN avec $m = 1$ (le coût 1 est assigné aux n-uplets interdits). Comme d'habitude, une valeur (i, a) est dite arc cohérente (AC) sur w_{ij} ssi il y a une paire (a, b) (support) qui satisfait w_{ij} et que $b \in D_j$ (valide). Un CSP est AC si toutes ses valeurs sont AC sur toutes les contraintes. L'établissement de AC dans un CSP P produit sa fermeture AC, un CSP qui est équivalent à P et qui est AC.

2.2 Cohérence d'Arc Virtuelle

Définition 1. *Étant donné un CFN* $P = (X, D, W, m)$*, le problème* $Bool(P) = (X, D, \overline{W}, 1)$ *est un CSP qui satisfait :* $\exists \overline{w}_S \in \overline{W}$ *ssi* $\exists w_S \in W$*,* $S \neq \varnothing$ *et* $\overline{w}_S(t) = 1 \Leftrightarrow w_S(t) \neq 0$*. Un CFN* P *est arc cohérent virtuel (VAC) ssi la fermeture arc cohérente classique du CSP* $Bool(P)$ *n'est pas vide [6].*

Bool(P) est un CSP dont les solutions sont exactement les n-uplets complets ayant un coût de $w_\varnothing$

dans P. Si P n'est pas VAC, l'établissement de AC dans Bool(P) va générer un domaine vide. Dans ce cas, comme indiqué dans [6], il existe une séquence d'EPT dont l'application mène à une augmentation de $w_\varnothing$. L'algorithme d'établissement de VAC utilise cette propriété dans un processus itératif en trois phases.

La phase 1 consiste à établir AC dans le CSP Bool(P). Dans cette phase, les effacements de valeurs sont mémorisés dans une structure de données notée tueur. Lorsqu'une valeur (i, a) n'a pas de support valide sur $\overline{w}_{ij}$, on affecte tueur $(i, a) = j$ et on efface (i, a). Si aucun domaine de variable n'est vidé, P est VAC et le processus d'établissement de VAC s'arrête.

Sinon, la phase 2 identifie le sous-ensemble des valeurs effacées qui sont nécessaires pour produire le domaine vide et les stocke dans une pile R. En retraçant l'histoire des propagations de la phase précédente définie par tueur, à partir de la variable dont le domaine a été vidé jusqu'aux coûts non nuls, il est possible de déterminer les effacements indispensables. Par ailleurs, cette phase évalue la quantité de coût maximale qu'il est possible de déplacer sur $w_\varnothing$ (notée λ) et définit l'ensemble d'EPT à appliquer dans P afin d'atteindre cette augmentation. Comme présenté dans [6], toutes les quantités de coût déplacées par EPT sont stockées dans deux tableaux de nombres entiers $k(j, b)$ et $k_{ij}(j, b)$. Ils représentent respectivement le nombre de demandes de coût d'un montant λ à projeter sur (j, b) et à étendre de (j, b) sur w_{ij}. Notez que le déplacement de ces coûts respecte une règle de conservation simple. Pour chaque valeur (j, b) d'une variable non vide j qui n'est pas la source de coût ($w_j(b) = 0$), la quantité de coût que (j, b) reçoit par Project est exactement la quantité de coût sortant de (j, b) par Extend (voir [6], page 465).

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

La phase 3 de VAC, comme décrite en détail dans l'algorithme 2, modifie le CFN original en appliquant les EPT définies par les structures de données k et k_{ij} sur toutes les valeurs effacées indispensables stockées dans R. Une valeur (j, b) effacée par $\overline{w}_{ij}$ va recevoir un coût de $k(j, b) \times \lambda$ par Project depuis w_{ij} (ligne 7). Néanmoins, il faut tout d'abord étendre un coût de $k_{ij}(i, a) \times \lambda$ à partir des supports invalides (i, a) sur w_{ij} (ligne 5). Cette phase produit finalement un nouveau problème P' équivalent à P avec un minorant $w_\varnothing$ augmenté (ligne 8).

Notez qu'à la fin de la phase 1, lorsqu'il y a une variable de domaine vide i_0, le problème obtenu n'est pas forcément la fermeture arc cohérente maximum de Bool(P). Il y a peut-être quelques valeurs qui n'ont pas été encore supprimées bien qu'elles n'aient aucun support valide, parce que le domaine concerné n'a pas encore été révisé. Une telle variable est encore dans la file de propagation Q_{AC} qui contient les variables restant à propager. Quant aux autres valeurs, soit elles ont un support valide, soit elles ont été supprimées et leur tueur associé n'est pas vide. Un tel problème est appelé une fermeture arc cohérente *partielle justifiée* de Bool(P). Aucun domaine dans ce problème ne peut être plus large que dans Bool(P), les valeurs effacées n'ont pas de support valide et sont justifiés correctement par les tueur.

Algorithme 2 : Phase 3 de VAC : Application des EPTs

```
1 tant que R ≠ ∅ faire
2     (j, b) ⟵ R.pop();
3     i ⟵ tueur [j, b];
4     pour chaque a ∈ D_i s.t. k_ij(i, a) ≠ 0 faire
5         Extend(i, a, w_ij, λ × k_ij(i, a));
6         k_ij(i, a) ⟵ 0;
7     Project(w_ij, j, b, λ × k(j, b));
8 UnaryProject(i_0, λ);
```

L'algorithme de VAC a une complexité spatiale en $O(ed)$ parce que les structures de données tueur, k et k_{ij} nécessitent une mémoire de $O(nd)$, $O(nd)$ et $O(ed)$ respectivement. La complexité en temps de chaque itération de VAC est en $O(ed^2)$ tant qu'un algorithme de AC optimal est utilisé dans la phase 1.

Les itérations de VAC établissent AC sur une séquence de CSP légèrement modifiés : Bool(P), Bool(P'), ... Cela motive l'idée d'utiliser des algorithmes d'AC dynamique pour appliquer AC à chaque itération.

2.3 Cohérence d'Arc Dynamique

Définition 2 ([8]). *Un CSP dynamique est une séquence $P_0, P_1, ..., P_n$ de CSP où chaque P_i est un CSP résultant de l'addition ou du retrait d'une contrainte dans P_{i-1}.*

Les algorithmes de Cohérence d'Arc Dynamique (DnAC) consistent à maintenir la cohérence d'arc dans la séquence des problèmes P_i. L'établissement d'AC est naturellement incrémentale pour la restriction (addition d'une contrainte) : il est suffisant d'établir AC (phase 1) avec une file de propagation Q_{AC} contenant les variables de la contrainte qui vient d'être ajoutée. Cependant, AC n'est pas incrémental pour la relaxation (retrait d'une contrainte). On a donc besoin d'un processus supplémentaire pour régler ce cas : il faut

restaurer les valeurs supprimées directement ou indirectement à cause de la contrainte retirée dès lors qu'il n'y pas d'autre raison de les supprimer.

Différentes familles d'algorithmes ont été proposées pour DnAC. Dans cet article, nous nous appuyons sur AC/DC2 [1] qui est une amélioration de AC/DC [2] s'appuyant sur des structures de données persistantes très limitées. La plus importante d'entre-elles est un tableau *justification*(i, a) (comme dans DnAC4 [3]) qui mémorise la source de l'effacement de (i, a). Cette structure correspond exactement à la structure tueur de VAC [1].

Lors du retrait d'une contrainte donnée w_{ij}, AC/DC2 se déroule en 3 étapes : 1) initialisation : toutes les valeurs des variables i et j supprimées par w_{ij} sont candidates pour être restaurées et marquées en "Propageable". On peut déterminer exactement ces valeurs en se basant sur le tableau *justification*. 2) propagation : chaque valeur propageable est propagée vers ses variables voisines en vérifiant si elle offre un nouveau support aux valeurs dont elle était le tueur (valeurs qui ont été effacées par le manque de support sur la variable propageable) et qui seront alors aussi marquées comme "Propageable". Notez que grâce à *justification*, quand on propage la variable "Propageable" i, on ne considère que les valeurs (j, b) effacées par w_{ji} comme candidates à la restauration, au lieu de tester toutes les valeurs du voisinage. Lorsqu'une valeur est propagée sur toutes les variables de son voisinage, elle est marquée "Restaurable" et sera restaurée. 3) filtrage : toutes les valeurs restaurées ont besoin d'être revérifiées pour la cohérence d'arc. Ceci est accompli en établissant l'AC via la file de propagation Q_{AC}, initialisée de façon à imposer la révision des variables ayant des valeurs restaurées.

La complexité spatiale d'un algorithme AC/DC basé sur AC3 est en $O(nd + e)$. La complexité en temps de AC/DC2 est définie par l'algorithme de filtrage utilisé dans la dernière étape : en $O(ed^3)$ pour AC-3 et en $O(ed^2)$ pour AC2001 [1].

3 Algorithme de VAC Dynamique

En établissant VAC, le CFN P est incrémentalement modifié dans la phase 3 de chaque itération de VAC. Nous proposons donc une version améliorée de VAC, appelée VAC dynamique (DynVAC), qui utilise AC dynamique pour maintenir AC dans les Bool(P) successifs au lieu de filtrer à partir de zéro à chaque itération comme le fait l'algorithme VAC standard. Nous utilisons AC/DC2 [1] basé sur AC2001, pour son optimalité. Un avantage de l'utilisation d'AC/DC2 est que la structure de données *justification* est fournie gratuitement par la structure tueur de VAC.

3.1 Propriétés et algorithme

Dans les CSP dynamiques traditionnels, les algorithmes de DnAC sont appliqués après chaque addition ou suppression d'une contrainte. Dans le cas de VAC, la situation est plus compliquée parce qu'une série de modifications de Bool(P) a lieu durant la phase 3 via l'application des EPT. Un appel de Project(w_{ij}, i, a, α) a deux effets : 1) l'augmentation du coût unaire $w_i(a)$ et 2) la diminution de coûts dans la fonction de coût binaire w_{ij}. Si un coût $w_i(a)$ nul augmente et devient strictement positif, la valeur (i, a) est enlevée de Bool(P), et ceci correspond à une restriction. Au contraire, si la paire (a, b) de coût strictement positif voit son coût atteindre zéro, cette paire précédemment interdite dans $\overline{w}_{ij}$ devient autorisée et ce fait correspond à une relaxation. Au lieu d'appliquer un algorithme de DnAC à chaque opération Project, Extend et UnaryProject, une approche plus rationnelle consiste à appliquer les principes de DnAC une seule fois après la phase 3 pour éviter les restaurations/suppressions inutiles.

Chaque itération de VAC transforme le CFN P actuel en un problème modifié P' dont les fonctions de coûts sont notées w'_i et w'_{ij}. Il est possible de calculer les valeurs w'_i et w'_{ij} dès la fin de la phase 2 puisqu'elles sont définies par une séquence connue d'opérations Project, Extend et UnaryProject appliquées sur w_i et w_{ij}. Par exemple, toutes les valeurs a d'une variable non vide i satisfont l'équation suivante :

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

De manière similaire, il est possible de calculer les coûts de la variable de domaine vide et des w'_{ij}. Nous allons maintenant prouver que l'effet global des toutes les EPT sur Bool(P) dans la phase 3 est simplement un ensemble de relaxations au niveau des fonctions unaires et binaires.

Propriété 1. *Dès la phase 2, nous savons que :*

(a) $\forall (i, a)$ *:* $w'_i(a) \leq w_i(a)$.

(b) $\forall (i, a)$ *et* (j, b) *: si* $w'_{ij}(a, b) \neq w_{ij}(a, b)$ *alors* (i, a) *ou* (j, b) *est effacée de la fermeture arc cohérente partielle justifiée actuelle de Bool(P).*

Démonstration. (a) Dans VAC, la seule opération qui peut augmenter les coûts unaires est l'opération Project Cependant, selon l'équation 1, si une valeur (i, a) reçoit un coût via Project, elle va faire un Extend de

1. AC/DC2 introduit une autre structure *time-stamp*(i, a) qui mémorise l'ordre des effacements. Le premier auteur de AC/DC2 nous a confirmé que cette structure est subsumée par *justification*(i, a) (communication privée) et est donc inutile.

la même quantité de coût (vers d'autres fonctions de coût binaires ou vers $w_\varnothing$). En conséquence, les coûts unaires ne peuvent pas augmenter.

(b) Il n'y a que deux façons pour changer un coût binaire $w_{ij}(a,b)$. C'est d'exécuter une opération Project à partir de w_{ij} ou une opération Extend sur lui. Néanmoins, la phase 3 de VAC n'applique Project et Extend qu'avec des valeurs extraites de la file des valeurs supprimées R (construite dans la phase 2). Donc, quand le coût d'une paire (a,b) change, il faut que (i,a) ou (j,b) soit effacée. □

Corollaire 1. *Les EPT appliquées dans la phase 3 de VAC transformant Bool(P) en Bool(P') ne génèrent que les types de relaxation suivants :*

(1) *des valeurs (i,a) deviennent autorisées $(w_i(a) > w'_i(a) = 0)$.*

(2) *des paires $((i,a),(j,b))$ deviennent autorisées $(w_{ij}(a,b) > w'_{ij}(a,b) = 0)$.*

Démonstration. Selon la Propriété 1(a), nous savons que les coûts unaires ne peuvent que diminuer. Certains coûts non nuls peuvent diminuer à zéro. Donc, les valeurs correspondantes réapparaissent dans Bool(P'). Ce fait peut être considéré comme la rétraction de contraintes unaires.

Selon la Propriété 1(b), le coût des paires peut augmenter ou diminuer. Si un coût binaire (a,b) augmente de zéro à non nul, cela ne peut détruire aucun support valide parce qu'une des deux valeurs est effacée dans la fermeture partielle actuelle. Le support n'est donc pas valide. Au contraire, si le coût de (a,b) diminue, cela peut créer un nouveau support pour a ou b. □

En conséquence, l'algorithme DnAC utilisé peut être spécialisé pour un ensemble de relaxations (Algorithme 3). Le protocole de restauration se compose de 3 étapes, comme dans AC/DC2. Notez que Bool(P) est maintenu après la phase 3 de chaque itération. $\overline{D}_i$ mentionné dans l'algorithme 3 représente le domaine de la variable i dans la fermeture arc cohérente partielle justifiée finale obtenue après la phase 1.

L'étape **initialisation** parcourt toutes les valeurs dans la file R dans le même ordre de parcours que la phase 3 pour identifier les valeurs qui doivent être restaurées (ligne 2). La variable i_0, dont le domaine a été vidé, est traitée séparément. Selon le corollaire 1, il y a deux cas possibles : (1) quand une valeur (i,a) devient autorisée $w_i(a) > w'_i(a) = 0$, elle sera restaurée (ligne 5), (2) quand un nouveau support valide apparaît pour la valeur (j,b) en satisfaisant $(w_{ij}(a,b) \oplus w_i(a)) > (w'_{ij}(a,b) \oplus w'_i(a)) = 0$ et tueur $[j,b] = i$, (j,b) sera restaurée (ligne 7). Lorsqu'une valeur (i,a) est restaurée, elle est stockée dans un tableau *restauré*$[i]$ et la variable i est ajoutée dans la liste RL qui contient les variables dont la restauration doit être propagée.

Algorithme 3 : Mise à jour de Bool(P)

```
1  Procédure Initialisation
2      pour chaque (j,b) ∈ R faire
3          i ⟵ tueur [j,b];
4          pour chaque a ∈ D_i − D̄_i faire
5              si (w_i(a) > 0) ∧ (w'_i(a) = 0) alors
               Restaurer(i,a);
6              si b ∉ D̄_j ∧ w'_i(a) = 0 ∧ w'_ij(a,b) = 0
               alors Restaurer(j,b);
7      pour chaque a ∈ D_{i0} tel que
       w_{i0}(a) > 0 ∧ w'_{i0}(a) = 0 faire Restaurer(i0,a);
8  Procédure Restaurer(i,a)
9      ajouter a dans D̄_i et restauré[i];
10     ajouter i dans RL;
11     tueur [i,a] ← nil;
12 Procédure Propagation
13     tant que RL ≠ ∅ faire
14         i ← RL.pop();
15         pour chaque w_ij ∈ W faire
16             pour chaque b ∈ D_j − D̄_j s.t. tueur
               [j,b]= i faire
17                 si ∃a ∈ restauré[i] s.t. w'_ij(a,b) = 0
                   alors Restaurer(j,b);
18         restauré[i] ← ∅;
19         Q_AC ← Q_AC ∪ {j | w_ij ∈ W}
```

L'étape **propagation** propage les restaurations de valeurs vers les variables voisines, comme dans AC/DC2 (ligne 14,15). Chaque variable i peut restaurer une valeur (j,b) dans une variable voisine j si cette valeur a été supprimée à cause de w_{ij} (ligne 16) et qu'elle maintenant supportée par une valeur restaurée dans i (ligne 17). Après avoir propagé toutes les valeurs restaurées, la liste *restauré* est vidée (ligne 18) pour éviter de repropager les valeurs qui ont été déjà propagées.

L'étape **filtrage** doit éliminer les valeurs restaurées (i,a) qui ne sont pas arc cohérentes sur une contrainte $\overline{w}_{ij}$ et assigner j à tueur $[i,a]$. Cette tâche est exactement le travail de la phase 1 de VAC. En conséquence, nous avons intégré cette étape dans la phase 1 en ajoutant les variables voisines des variables ayant des valeurs restaurées dans la file de révision Q_{AC} (ligne 19).

Il est possible de prouver la correction de l'algorithme DynVAC en montrant que cet algorithme fournit exactement une fermeture arc cohérente partielle justifiée de Bool(P') pour l'itération suivante de VAC.

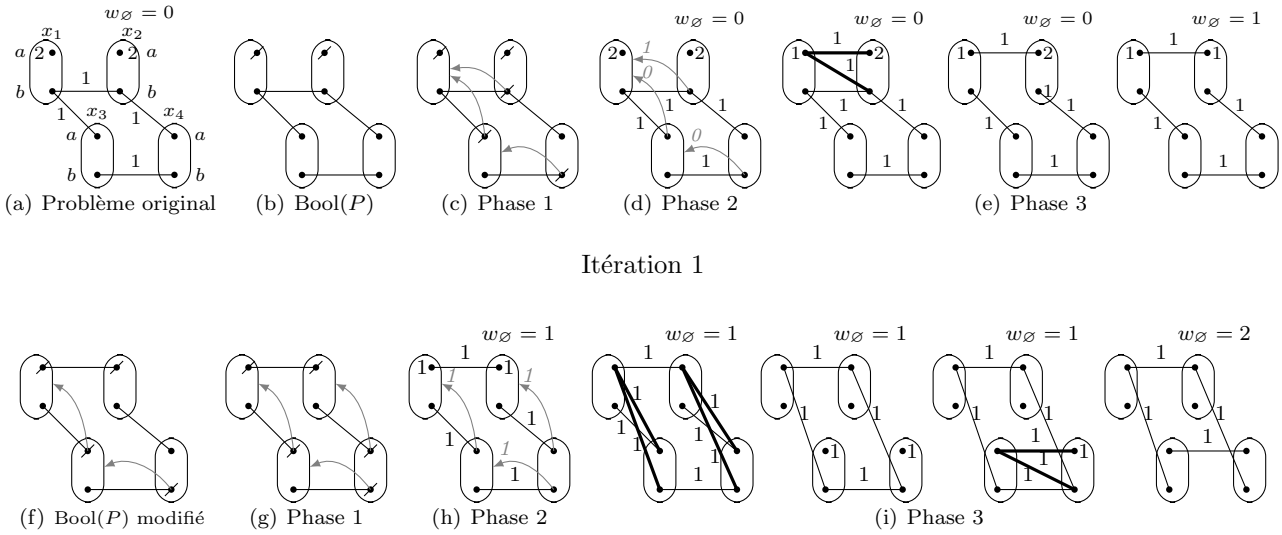

FIGURE 1 – Exemple du fonctionnement de VAC dynamique durant les itérations

Nous ne présentons pas la preuve dans cet article.

3.2 Exemple

L'avantage de DynVAC par rapport à VAC est que la liste des variables à réviser utilisée dans la phase 1 est maintenue pendant toutes les itérations, au lieu d'être réinitialisée à l'ensemble complet de toutes les variables X à chaque itération, comme fait l'algorithme VAC de [6]. Ce fait, illustré dans l'exemple suivant, permet d'éviter de répéter des filtrages inutiles.

Soit le CFN binaire de la Figure 1(a). Chaque variable a deux valeurs a et b, représentées par des sommets. Les coûts unaires non nuls sont affichés à côté des valeurs. Une arête entre deux sommets implique un coût binaire non nul de la paire correspondante. Les coûts nuls ne sont pas affichés. Dans Bool(P) (Figure 1(b)), les valeurs interdites sont barrées et les arêtes représentent les paires interdites.

Supposons que les contraintes soient révisées dans la phase 1 dans l'ordre (w_{13}, w_{34}, w_{12}, w_{24}). Après avoir révisé w_{13}, w_{34}, w_{12}, les valeurs $(3, a)$, $(4, b)$ et $(2, b)$ sont respectivement effacées de Bool(P). Dès que la variable 2 a un domaine vide, la phase 1 s'arrête (Figure 1(c)). Les flèches en gris représentent la structure de données tueur pointant sur la variable qui a causé la perte de support valide. Dans la phase 2 (Figure 1(d)), l'effacement de $(2, b)$ est suffisant pour produire un domaine vide. $w_\varnothing$ peut augmenter d'une quantité de coût d'au maximum $\lambda = 1$ en utilisant les coûts non nuls de $w_{12}(b, b)$ et $w_1(a)$. Les nombres en italique associés aux flèches indiquent la valeur correspondante de $k(i, a)$. En appliquant les EPT précédemment identifiées, la phase 3 (Figure 1(e)) transforme P en un problème équivalent P' avec $w'_\varnothing = 1$. Les coûts étendus sont affichés en gras.

L'établissement de VAC se poursuit car P' n'est pas encore VAC. Bool(P') n'est pas construit à partir de P' mais inféré à partir de Bool(P). Pour mettre à jour Bool(P), au lieu de considérer toutes les valeurs effacées, nous considérons uniquement les valeurs $(1, a), (2, a)$ et $(2, b)$ pour restauration car seule la contrainte w_{12} a été modifiée par des EPT dans la phase 3. Parmi les trois valeurs effacées considérées, seule $(2, b)$ est restaurée car elle a un coût final nul et un nouveau support (b, b) dans w_{12}. Cette restauration ne provoque aucune autre restauration. Les contraintes du problème Bool(P') sont inférées directement à partir des coûts non nuls de P' [2]. En fait, le résultat de la mise à jour, tel que décrit dans la Figure 1(f), est déjà une fermeture arc cohérente partielle justifiée de Bool(P'), avec deux valeurs effacées supplémentaires $(3, a)$ et $(4, b)$, et les tueur associés définis. La phase 1 de l'itération suivante peut démarrer à partir de ce problème. $(4, a)$ est effacée après la révision de w_{24} et le domaine de la variable 4 devient vide (Figure 1(g)). La phase 2 et la phase 3 fonctionnent de la même manière qu'à l'itération précédente. Le problème final (Figure 1(i)) avec $w''_\varnothing = 2$ est VAC.

2. En pratique, aucune contrainte n'est créée. On se contente de tester la positivité stricte du coût dans P'.

3.3 Maintien de VAC dynamique pendant la recherche

On peut maintenir VAC durant la recherche arborescente en maintenant le problème Bool(P), défini par la liste des valeurs effacées et leurs justifications, de manière incrémentale. Lors d'une affectation $i = a$, il est habituel de propager les fonctions binaires contenant i dans leur portée et de les déconnecter du réseau car la propagation en a extrait toute l'information. Dans Bool(P), la valeur (i, a) sélectionnée pouvait être effacée par absence de support valide. Après la suppression des fonctions binaires contenant i dans leur portée, une telle valeur peut redevenir viable dans Bool(P) et il faut donc envisager sa restauration. Lors d'une restriction de domaine ($i \neq a$, $i > a$ ou $i < a$), des valeurs de variables voisines peuvent perdre leur support, il faut donc refaire une partie de la propagation. Au lieu de reconstruire Bool(P) pour chaque nouveau nœud de l'arbre de recherche, le problème Bool(P) du nœud parent est mis à jour en ajoutant une variable affectée dans la liste RL ou les variables réduites dans la file de propagation Q_{AC}. En remontant dans l'arbre de recherche, la fermeture de Bool(P) est reconstruite via la seule restauration des `tueur` qui ont été sauvegardés par "trailing". En effet, dans une fermeture justifiée, `tueur` $[i, a]$ = null si et seulement si (i, a) n'a pas été effacée.

Comme dans [6], afin d'accélérer l'algorithme de VAC dynamique pendant le recherche, nous avons remplacé Bool(P) par une version relaxée mais de plus en plus stricte, notée Bool$(P)_\theta$, qui permet de collecter rapidement des contributions importantes au minorant $w_\varnothing$. Un n-uplet t est interdit dans Bool$(P)_\theta$ ssi son coût dans P est plus grand que θ. L'ensemble des coûts binaires c_{ij} non nuls du problème sont triés dans un nombre fixé k de groupes en temps linéaire (via l'utilisation d'un algorithme de type "*bucket sort*"). Les coûts minimum de chaque groupe définissent une séquence de seuils $(\theta_1, \theta_2, ...\theta_k)$. En partant de θ_1, les itérations de VAC fonctionnent avec un seuil fixé jusqu'à ce qu'il n'y ait plus de domaine vide. On contraint alors le problème en utilisant le seuil θ_{i+1}. Après le dernier θ_k, une stratégie géométrique, définie par $\theta_{i+1} = \theta_i/2$, est utilisée et s'arrête lorsque θ_i est plus petit qu'une valeur T donnée. Ce seuil T permet d'obtenir un algorithme qui se termine dans tous les cas. Dans VAC dynamique, le seuil θ est maintenu de manière incrémentale pendant la recherche avec l'objectif d'hériter du travail fait dans les nœuds parents. θ est également sauvegardé/restauré durant les backtracks par "*trailing*".

3.4 Complexité

DynVAC a la même complexité spatiale que VAC, en $O(ed)$. L'étape initialisation a une complexité en temps de $O(nd)$ parce qu'il y a au maximum $n \times d$ valeurs ayant un coût strictement positif (ligne 5, Algorithme 3). La complexité en temps dans le pire des cas de l'étape "propagation" est en $O(ed^2)$ parce que chaque paire de valeurs de chaque contrainte est testée au maximum une fois (ligne 17, 18, Algorithme 3). Donc, la mise à jour de Bool(P) a la même complexité en temps que la phase 3 de VAC : $O(ed^2)$. En conséquence, chaque itération de DynVAC a une complexité en temps en $O(ed^2)$ comme VAC, tant qu'un algorithme AC optimal est utilisé dans la phase 1.

Bien que DynVAC n'améliore pas la complexité asymptotique de VAC, les expérimentations présentées dans la prochaine section montrent qu'il peut fournir des accélérations importantes en pratique.

4 Expérimentations

Dans cette section, nous comparons l'efficacité de DynVAC à VAC que ce soit en pré-traitement ou maintenu pendant la recherche, sur un ensemble de problèmes extraits de la "Cost Function Library"[3]. Les expérimentations sont implémentées dans le solveur `toulbar2` et exécutées sur un processeur Intel(R) Core(TM)2@2.66GHz avec 4GB de RAM. Comme dans [5,6], pour chaque problème, nous établissons une version limitée de VAC qui s'arrête dès que l'augmentation de $w_\varnothing$ reste plus petite qu'une valeur ε donnée sur un nombre donné d'itérations successives. Nous avons fixé $\varepsilon = 0.05$ pour les expérimentations de prétraitement (Table 1) tandis que dans les expérimentations en recherche (Table 2) $\varepsilon = \frac{1}{10000}$ à la racine de l'arbre de recherche et $\varepsilon = 0.1$ pendant la recherche.

Pour le simple pré-traitement des problèmes, comme espéré, DynVAC est plus rapide que VAC sur une large partie des instances extraites de l'entrepôt CFLib. La table 1 affiche les moyennes du temps total (en secondes), du minorant (lb) et du nombre d'itérations (iter) pour l'établissement de VAC en utilisant l'algorithme VAC statique usuel et le nouvel algorithme DynVAC. Chaque ligne correspond à une classe de #inst problèmes. Le meilleur temps ou minorant est indiqué à chaque fois en gras. Les expérimentations montrent que DynVAC est respectivement 1,6, 3 et 5 fois plus rapide que VAC pour les classes *celar, tagsnp, warehouse* tandis qu'il fournit des minorants de même qualité. Nous observons que le nombre d'itérations en moyenne de DynVAC peut augmenter par rapport à

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

TABLE 1 – Les valeurs du minorant, le temps, et le nombre d'itérations de VAC nécessaires pour traiter des problèmes réels et fabriqués de la "*Cost Function Library*".

classe	#inst	VAC_ε			$DynVAC_\varepsilon$			$DynVAC_\varepsilon$ avec heuristique		
		lb	temps	iter	lb	temps	iter	lb	temps	iter
celar	32	6.180	3,14	382	**6.204**	**1,92**	418	5.892	1,12	319
prot_maxclique	10	1.016	**51,00**	1.022	1.016	364,00	1.022	1.016	56,95	1.022
tagsnp_r0.5	25	$1{,}43\times10^6$	364,31	8.798	$1{,}43\times10^6$	**116,57**	4.653	$1{,}43\times10^6$	81,46	5.810
tagsnp_r0.8	82	$1{,}11\times10^6$	4,64	155	$1{,}11\times10^6$	**1,53**	120	$1{,}11\times10^6$	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	$7{,}23\times10^6$	341,00	946	$\mathbf{7{,}24}\times10^6$	**66,00**	719	$7{,}25\times10^6$	114,17	790

celui de VAC. Mais le temps total en moyenne diminue comme espéré : il y a moins de travail à faire à chaque itération dans DynVAC parce que les effacements de valeurs sont hérités des itérations précédentes. Notez que le minorant final obtenu par DynVAC et VAC n'est pas obligatoirement identique. Après la première itération, VAC et DynVAC peuvent réviser les contraintes dans un ordre différent dans la phase 1 menant à des séquences d'EPT différentes. Le critère d'arrêt peut alors agir différemment selon les cas.

Néanmoins, DynVAC est plus lent que VAC (7 et 4 fois respectivement) pour toutes les catégories des problèmes de type "clique maximum" testées : *protein_maxclique* et *dimacs_maxclique*. Ces problèmes ont une structure spécifique avec des domaines booléens, des contraintes de différence binaires et des fonctions de coût (non dures) seulement au niveau unaire. Du fait de cette structure spécifique, chaque itération restaure de nombreuses valeurs en cascade mais de façon inutile, car elles vont être à nouveau effacées, pour une autre raison, dans l'itération suivante. Afin d'améliorer l'efficacité de DynVAC sur ces problèmes défavorables, nous avons utilisé une heuristique d'ordonnancement des révisions orientée variable, proposée dans [12]. Nous l'appliquons durant l'établissement de AC dans la phase 1. L'heuristique sélectionne premièrement dans la file Q_{AC} la variable dont la taille de domaine est la plus petite dans Bool(P). Ensuite, les contraintes sont traitées en ordre ascendant en terme de la taille des domaines des variables opposées. Le résultat obtenu, présenté dans la dernière colonne de la table 1, montre que cette heuristique permet d'améliorer considérablement la performance de DynVAC sur les problèmes de type "clique maximum". L'heuristique intégrée dans DynVAC fournit une performance comparable à celle de VAC statique dans ces cas extrêmement défavorables (20% de dépassement en temps qui est probablement causé par le calcul de l'heuristique).

Finalement, nous testons les performances de DynVAC et VAC lorsqu'elles sont maintenues pendant la recherche d'une solution de coût optimal, sur des problèmes d'affectation de fréquence (*celar*, [4]) et des problèmes d'affectation d'entrepôts (uncapacited warehouse location problems *warehouse*, [9]). Ces problèmes, qui ont été déjà utilisés pour l'évaluation de VAC dans [6], sont issus de problèmes difficiles, de taille parfois importante et possèdent de grands domaines, ce qui constitue sans doute un terrain de prédilection pour DynVAC. Sur ces problèmes, VAC est effectivement dominé par DynVAC au niveau du prétraitement. La table 2 affiche le temps (en secondes) et le nombre de nœuds visités durant la recherche, jusqu'à l'obtention et la preuve d'une solution optimale, en utilisant VAC statique ou DynVAC. La dernière colonne présente le ratio de temps CPU de DynVAC par rapport à celui de VAC. Comme dans le cas du pré-traitement, VAC dynamique continue à être plus efficace sur la plupart des instances de ces classes. Il est en moyenne 40% plus rapide que VAC et visite 18% de moins de nœuds sur des instances *celar* (correspondant à la section en haut de la table). Cependant, il y a quelques instances *celar* qui sont plus longues à résoudre avec DynVAC. Le résultat est encore plus visible avec la classe *warehouse* où DynVAC est de 2 à 5 fois plus rapide que VAC, sur toutes les instances de cette classe (sauf pour "capb") en visitant souvent un peu moins de nœuds.

5 Conclusion

Cet article présente une approche incrémentale pour l'établissement de VAC dans les réseaux de fonctions de coût. Notre approche combine l'idée des algorithmes d'arc consistance dynamique avec l'algorithme itératif VAC afin de maintenir efficacement la cohérence d'arc dans le CSP Bool(P) durant l'établissement de VAC.

Le nouvel algorithme fournit un minorant de même qualité qu'avec l'algorithme VAC statique mais il est plus efficace sur de nombreux problèmes, particuliè-

TABLE 2 – Résultats de DynVAC et VAC pendant la recherche sur un sous-ensemble des instances.

	VAC_ε		$DynVAC_\varepsilon$		ratio
	temps	nœuds	temps	nœuds	temps
cl6-1-24	12	9.686	18	13.962	1,5
cl6-1	26	14.871	38	17.131	1,46
cl6-2	29	11.228	46	13.081	1,59
cl6-3	161	54.083	133	35.614	0,83
$cl6\text{-}4_r$	160	47.646	138	30.399	0,86
cl6-4	210	52.724	270	47.806	1,29
cl7-1	194	17.951	84	8.733	0,43
cl7-2	1.076	33.415	773	33.545	0,72
cl7-3	4.983	142.755	2.979	85.626	0,6
cl7-4	9.193	380.566	7.426	207.207	0,81
$gr11_{rm}$	6	618	5	639	0,83
gr11	470	6.600	366	4.292	0,78
$gr13_{rm}$	34	50	20	11	0,59
gr13	1.431	5.921	1.144	3.022	0,8
$sc06\text{-}16_r$	884	142.740	625	170.851	0,71
sc06-16	1.873	283.872	1.052	290.993	0,56
$sc06\text{-}18_r$	677	143.386	360	118.060	0,53
sc06-18	1.511	276.610	736	213.282	0,49
$sc06\text{-}20_r$	595	125.674	358	124.856	0,6
sc06-20	765	130.260	508	164.515	0,66
$sc06\text{-}22_r$	374	96.270	177	79.211	0,47
sc06-22	433	91.202	235	74.481	0,54
$sc06\text{-}24_r$	156	45.558	81	36.407	0,52
sc06-24	165	40.045	89	37.170	0,54
$sc06\text{-}30_r$	5	1826	3	1.697	0,6
sc06-30	14	6.990	10	6.184	0,71
$sc06_r$	3.629	310.034	2.183	270.387	0,6
sc06	5.227	347.925	2.454	275.992	0,47
capa	2.462	1.100	1.013	1.101	0,41
capb	3.019	1.350	6.168	1.826	2,04
capc	2.027	1.650	1.228	1.233	0,61
capmo1	123	2.254	95	2.671	0,77
capmo2	64	407	21	328	0,33
capmo3	93	1.105	32	993	0,34
capmo4	74	846	18	574	0,24
capmo5	75	412	14	305	0,19
capmp1	1.573	3.030	920	3.554	0,58
capmp2	1.063	1.611	452	2.130	0,43
capmp3	1.123	1.454	443	1.474	0,39
capmp4	1.340	2.585	680	1.518	0,51
capmp5	984	1.932	311	1.541	0,32
capmq1	5.111	3.546	2.374	2.419	0,46
capmq2	6.520	4.408	3.209	4.633	0,49
capmq3	6.310	3.943	2.743	3.115	0,43
capmq4	7.530	7.208	4.295	6.124	0,57
capmq5	13.938	13.980	7.219	11.629	0,52

rement sur les problèmes avec coûts élevés et grands domaines. Cependant, DynVAC peut être ralenti pour certains problèmes spécifiques comme les problèmes de la classe *clique maximum*. L'intégration d'une heuristique de révision orientée variable dans l'algorithme AC permet d'éviter ce comportement pathologique.

Dans l'avenir, nous comptons étendre DynVAC aux fonctions de coût d'arité arbitraire et identifier des heuristiques de révision appropriées qui pourraient améliorer la performance de DynVAC et en même temps améliorer le minorant produit.

Références

[1] Roman Barták and Pavel Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. In *Proc. of the* 18^{th} *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* 6^{th} *IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

[3] Christian 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.P. 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] Rina Dechter and Avi 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* 19^{th} *IJCAI*, pages 84–89, Edinburgh, Scotland, August 2005.

[11] 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, August 1995.

[12] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence*, pages 163–163, 1992.