Cohérence d'arc existentielle : un pas de plus vers la cohérence d'arc complète

Matthias Zytnicki[1], Federico Heras[2], Simon de Givry[1], Javier Larrosa[2]
[1]INRA Toulouse – BIA
[2]UPC – LSI
zytnicki@toulouse.inra.fr

Résumé

Les réseaux de contraintes pondérées offrent un cadre de représentation et de résolution permettant d'exprimer des contraintes dites « molles » utiles pour modéliser un très large champ d'applications. La plupart des solveurs que l'on trouve actuellement implantent un algorithme de séparation et évaluation en profondeur d'abord qui maintient une cohérence locale durant la phase d'exploration de l'arbre. Les résultats empiriques tendent à prouver que maintenir FDAC* semble être le meilleur choix en pratique. Nous présentons dans cet article une forme d'arc cohérence plus forte, appelée *cohérence d'arc existentielle et directionnelle*, ainsi qu'un algorithme permettant de transformer un réseau de contraintes pondérées en un réseau équivalent, satisfaisant cette propriété. L'algorithme s'avère efficace sur de nombreuses classes de problèmes.

Abstract

The weighted CSP framework is a soft constraint framework with a wide range of applications. Most current state-of-the-art complete solvers can be described as a basic depth-first branch and bound search that maintain some form of arc consistency during the search. Empirical results indicate that maintaining FDAC* seems to be the best in practice. In this paper we introduce a new stronger form of arc consistency, that we call *existential directional arc consistency* and we provide an algorithm to enforce it. The efficiency of the algorithm is empirically demonstrated in a variety of domains.

1 Introduction

Les réseaux de contraintes pondérées (RCP) constituent une extension bien connue du formalisme des réseaux de contraintes (RC) et permettent de traiter efficacement un grand nombre d'applications pratiques. Récemment, la propriété de cohérence d'arc a été généralisée au formalisme des RCP [2, 9]. Il existe quatre généralisations connues se réduisant à la cohérence d'arc dans le cadre des RC :

- AC* [7], qui transpose la cohérence d'arc au cas valué,
- DAC* [8], qui transpose la cohérence d'arc directionnelle au cas valué,
- FDAC* [8], définie comme la conjonction des deux propriétés précédentes,
- FAC*, où chaque valeur de chaque variable doit posséder un support complet.

Malheureusement, toutes les instances de RCP ne peuvent être transformées en une instance possédant la propriété FAC*, ce qui rend la propriété inutile en pratique. FDAC* est la plus forte propriété qui puisse être établie sur un RCP. Dans [8], un algorithme permettant d'établir FDAC* a été proposé avec une complexité de $\mathcal{O}(end^3)$ (n représentant le nombre de variables de l'instance, e le nombre de contraintes et d la taille du plus grand domaine). De plus, l'article a montré que maintenir FDAC* durant l'exploration est un meilleur choix que maintenir AC*, plus faible.

Dans notre article, nous présentons une nouvelle cohérence locale appelée *cohérence d'arc existentielle et directionnelle* (EDAC*). Nous montrons que EDAC* est plus forte que FDAC*, notamment par le fait que la première exige l'*existence* d'une valeur pour laquelle tous les supports doivent être des supports complets. Intuitivement, la différence entre l'algorithme proposé et ceux précédemment cités est que les derniers s'attachent à obtenir une cohérence locale par rapport à chaque contrainte binaire considérée indépendamment, alors que le premier prend en compte toutes les contraintes qui s'appliquent sur une variable dans leur ensemble.

Nous présentons aussi un algorithme qui établit cette propriété en temps $\mathcal{O}(ed^2 \times \max\{nd, \top\})$, $\top$ étant la borne supérieure de l'instance du RCP. Cette complexité, qui est une borne supérieure dont on ne sait pas si elle est effectivement atteinte, utilise les structures de données de AC2001 (cf. [1]), sur la proposition de [9]. Toutefois, d'autres structures de données peuvent éventuellement donner de meilleurs résultats en pratique, même si la complexité théorique ne semble pas être modifiée.

Un des intérêts majeurs de EDAC* est qu'il peut être inclus dans un algorithme de séparation et évaluation (SE) qui *maintient* la propriété à chaque nœud visité. Nous avons évalué expérimentalement cette idée sur des instances de Max-SAT [3], Max-CSP [8] et des problèmes du placement d'entrepôts. Les tests montrent que maintenir EDAC* est toujours au moins aussi bon que maintenir FDAC*.

2 Définitions

Une *structure de valuation* est une structure algébrique permettant de spécifier les coûts dans les problèmes de satisfaction de contraintes valuées [11]. Elle est définie par le tuple $\mathcal{S} = \langle E, \oplus, \prec \rangle$, où :

- E est l'ensemble des coûts,
- $\prec$ est une relation qui ordonne totalement E et définit ainsi son plus petit élément ($\bot$) et son plus grand élément ($\top$),
- $\oplus$ est une opération interne de E utilisée pour combiner les coûts.

D'après [9], la structure de valuation d'un RCP est $\mathcal{S} = \langle [0..k], \oplus, < \rangle$ où :

- $E \subset \mathbb{N}$ et k est un entier naturel positif,
- $\oplus$ est défini par : $\forall (a,b) \in E^2, a \oplus b = \min\{a + b, k\}$,
- $<$ est la relation d'ordre habituelle sur les entiers.

Dans ce cas, $0 = \bot$ et $k = \top$. Il est de plus utile de définir la soustraction $\ominus$ des coûts :

$$a \ominus b = \begin{cases} a - b & \text{si } a \neq k \\ k & \text{sinon} \end{cases}$$

Un *réseau de contraintes pondérées* est un tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{S} \rangle$ où :

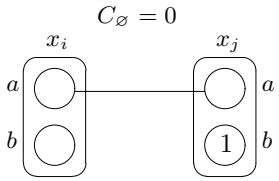
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est l'ensemble des *variables* ;
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ est l'ensemble des *domaines* finis associés aux variables, chaque domaine D_i est l'ensemble des valeurs que peut prendre x_i ;
- $\mathcal{C}$ est l'ensemble des contraintes pondérées unaires et binaires, c'est-à-dire l'ensemble des fonctions de coûts sur $\mathcal{S}$.

Une contrainte unaire C_i est une fonction de la forme : $D_i \to E$, une contrainte binaire C_{ij} est de la forme : $D_i \times D_j \to E$. On suppose qu'il existe pour chaque variable x_i une contrainte unaire C_i (éventuellement la fonction nulle sur D_i) et une contrainte d'arité nulle $C_\varnothing$ (éventuellement nulle).

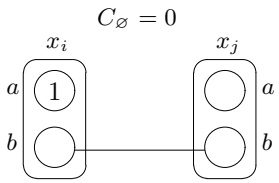
Lorsqu'une contrainte C assigne à un tuple t le coût de $\top$, cela signifie que t est interdit par C. Si le coût est différent, l'affectation est autorisée, avec un coût $C(t)$. Le *coût* d'une affectation $t = (v_1, \ldots, v_n)$ de $D_i \times \ldots \times D_n$, notée $\mathcal{V}(t)$, est la somme de tous les coûts :

$$\mathcal{V}(t) = \bigoplus_{\substack{C_{ij} \in \mathcal{C} \\ i<j}} C_{ij}(v_i, v_j) \oplus \bigoplus_{C_i \in \mathcal{C}} C_i(v_i) \oplus C_\varnothing$$

Un tuple t est *cohérent* si $\mathcal{V}(t) \prec \top$. Le problème général qui consiste à trouver *un tuple cohérent de coût minimal* est NP-difficile. Un RCP dont $\top$ vaut 1 est un réseau de contraintes classiques.



(a) le réseau originel



(b) après projection

FIG. 1 – Deux instances équivalentes de RCP ($\top = 2$).

Exemple 1 *Considérons le problème décrit* FIG. *1(a). Il a deux variables x_i et x_j, chacune ayant deux valeurs a et b. Les coûts des contraintes unaires sont donnés dans les cercles (par défaut, la valeur est de 0). Les contraintes binaires sont représentées par des arcs reliant une paire de valeurs, l'arc est étiqueté par le coût de cette paire (la valeur par défaut étant 1). S'il n'existe pas d'arc entre deux valeurs, le coût est nul pour la contrainte. Dans les deux exemples, le coût optimal est 0, obtenu par le tuple (b, a).*

3 Quelques cohérences locales de RCP

Deux instances de RCP définies sur le même ensemble de variables sont dites *équivalentes* si elles

donnent la même distribution de coût sur toute instanciation complète. Les cohérences locales sont largement utilisées pour transformer des problèmes en des problèmes équivalents, plus simples. En général, les contraintes sont données explicitement par des tables de coûts, ou implicitement par des expressions mathématiques ou des procédures algorithmiques. Par souci de clarté, nous supposerons l'expression explicite des contraintes. La plus simple des cohérences locales est la *cohérence de nœud* :

Définition 1 [7] *La variable x_i est nœud-cohérente ssi :*
- $\forall v \in D_i, C_\varnothing \oplus C_i(v) \prec \top$,
- $\exists v \in D_i, C_i(v) = \bot$.

Un RCP est nœud-cohérent ssi toutes ses variables le sont.

Un RCP peut facilement être transformé en un RCP nœud-cohérent équivalent si l'on projette toutes les contraintes unaires sur le $C_\varnothing$ et en éliminant toutes les valeurs interdites. C'est ce que font les fonctions ProjetteUnaire et ÉlagueVariable (Alg. 1).

Les propriétés de cohérence d'arc sont basées sur la notion de *support* et de *support complet*. Pour une contrainte binaire C_{ij}, $w \in D_j$ est un support de $v \in D_i$ si $C_{ij}(v, w) = \bot$. C'est un support complet si $C_{ij}(v, w) \oplus C_j(w) = \bot$

Définition 2 [7] *La variable x_i est arc-cohérente si pour toute contrainte binaire C_{ij}, toute valeur de son domaine $v \in D_i$ possède un support sur C_{ij}. Un RCP est arc-cohérent (AC*) si toutes ses variables le sont.*

Définition 3 *La variable x_i est complètement arc-cohérente si pour toute contrainte binaire C_{ij}, toute valeur de son domaine $v \in D_i$ possède un support complet sur C_{ij}. Un RCP est complètement arc-cohérent (FAC*) si toutes ses variables le sont.*

Les supports pour $v \in D_i$ par rapport à D_j peuvent être trouvés par projection des coûts binaires $C_{ij}(v, \cdot)$ sur $C_i(v)$. C'est ce qui est fait par la procédure Projette (Alg. 1). La procédure ChercheSupport (Alg. 2) trouve un support sur D_j pour chaque valeur de D_i.

Exemple 2 *Le RCP décrit* Fig. *2(a) n'est pas AC* car la valeur $a \in D_i$ n'a pas de support par rapport à la variable x_j. La propriété est obtenue en exécutant ChercheSupport(i, j) (ce qui nous donne l'instance* Fig. *2(b)) puis ProjetteUnaire(i) (instance* Fig. *2(c)).*

Pour obtenir un support complet pour les valeurs de D_i par rapport à D_j (assuré par la fonction ChercheSupportComplet de Alg. 2), il faut étendre les coûts unaires de $C_j(\cdot)$ sur $C_{ij}(\cdot, \cdot)$ (procédure Étend de Alg. 1). Enfin, les coûts binaires sont projetés sur $C_i(v)$.

Dans le cas des RC (où $\top = 1$), être un support pour $v \in D_i$ est équivalent à être un support complet et les deux notions se réduisent à la même propriété. C'est pourquoi AC* et FAC* se réduisent au même RC. Ce n'est pas le cas dans le cadre des RCP et alors que, dans [8], il a été prouvé qu'un RCP pouvait être transformé en une instance AC* équivalente en $\mathcal{O}(ed^3)$, il n'est pas toujours possible d'en trouver une qui satisfasse la propriété FAC*.

Exemple 3 *Le problème* Fig. *1(a) est AC*, mais pas FAC* car la valeur $a \in D_i$ n'a pas de support complet. On peut alors appeler la fonction ChercheSupportComplet(i, j), ce qui donne le problème* Fig. *1(b). Il n'est pas FAC* non plus car $b \in D_j$ n'a pas de support complet. Appeler ChercheSupportComplet(i, j) (cf.* Alg. *2) nous ramènerait à l'instance précédente. On peut facilement prouver que ce problème n'a pas d'équivalent FAC*.*

C'est pourquoi FAC* n'est pas une bonne propriété. Pour tenter de résoudre le problème, une propriété plus faible a été proposée. Dans la suite, nous supposerons que l'ensemble des variables $\mathcal{X}$ est totalement ordonné par $<$.

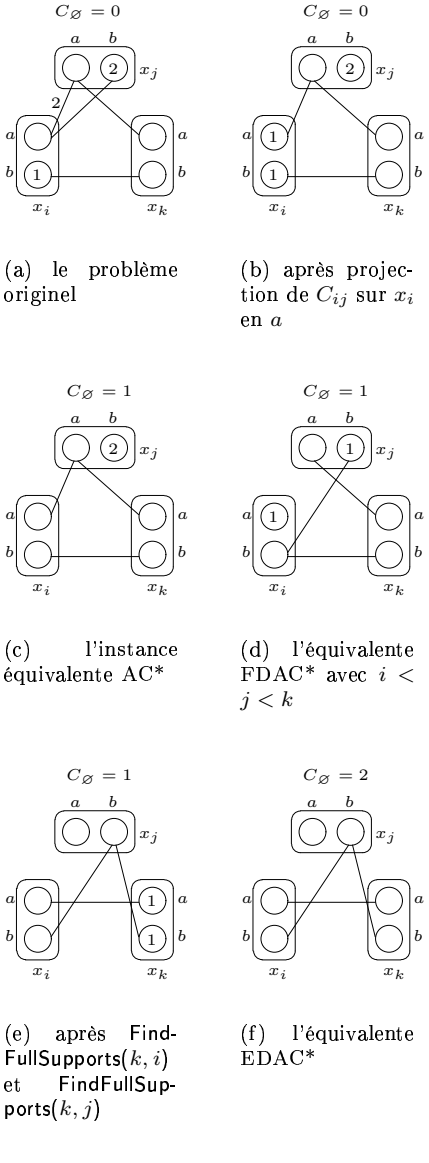
Définition 4 [8] *La variable x_i est* directionnellement arc-cohérente *si toutes ses valeurs $v \in D_i$ ont un support complet pour chaque contrainte binaire C_{ij} telle que $j > i$. Elle est* complètement et directionnellement arc-cohérente *(FDAC*) si elle est de plus arc-cohérente. Un RCP est complètement et directionnellement arc-cohérente si chaque variable l'est.*

Exemple 4 *Le problème* Fig. *2(c) est AC* mais pas FDAC* car la valeur $a \in D_i$ n'a pas de support complet par rapport à la variable x_j. La propriété FDAC* est établie par l'exécution de ChercheSupportComplet(i, j) (cf.* Alg. *2), qui produit* Fig. *2(d).*

Tout RCP peut être transformé en une instance équivalente satisfaisant FDAC* en temps $\mathcal{O}(end^3)$ [8]. Dans le cas des RC, FDAC* se ramène aussi à la cohérence d'arc. Dans le cas général, FAC* est plus fort que FDAC*, qui est plus fort que AC*. Il est alors intéressant de savoir si on peut trouver une propriété plus forte que FDAC*, tout en garantissant l'existence d'une instance possédant cette propriété pour tout problème.

4 La cohérence d'arc existentielle : une propriété plus forte

Considérons le problème FDAC* Fig. 2(d). Notons que la valeur $a \in D_k$ possède un support en x_j (ce qui



(a) le problème originel

(b) après projection de C_{ij} sur x_i en a

(c) l'instance équivalente AC*

(d) l'équivalente FDAC* avec $i < j < k$

(e) après FindFullSupports(k, i) et FindFullSupports(k, j)

(f) l'équivalente EDAC*

FIG. 2 – Six instances équivalentes (avec $\top = 4$)

est exigé par FDAC*) mais pas de support complet. Nous avons la même situation pour la valeur $b \in D_k$ et x_i. Ainsi, le coût unaire de ces deux valeurs peut être augmenté en établissant les supports complets (FIG. 2(e)). Comme tous les coûts unaires des valeurs de x_k sont plus grands que $\perp$, la cohérence de nœud est perdue et la borne inférieure peut être augmentée (FIG. 2(f)). D'une manière générale, les supports complets peuvent être établis sans erreur possible dans les deux directions si l'on augmente $C_\varnothing$. Nous montrerons ensuite qu'il existe une cohérence locale naturelle derrière cette observation. Nous l'appellerons cohérence d'arc existentielle (EAC*).

Définition 5 *La variable x_i est* existentiellement arc-cohérente *s'il existe au moins une valeur $v \in D_i$ telle que $C_i(v) = \perp$ et si cette valeur possède un support complet pour toutes les contraintes C_{ij}. Un RCP est existentiellement arc-cohérent si toutes ses variables le sont.*

Toutes les cohérences locales précédemment évoquées demandent à chaque valeur de chaque variable une certaine propriété. Pour EAC*, on demande dans chaque variable l'*existence* d'une valeur possédant certaines propriétés. Il est important de noter que si une variable x_i n'est pas EAC*, alors pour toutes les valeurs $v \in D_i$ telles que $C_i(v) = \perp$, il existe x_j tel que $\forall w \in D_j, C_{ij}(v, w) \oplus C_j(w) \succ \perp$. L'établissement d'un support complet sur x_i violera donc la propriété NC* et il sera possible d'augmenter $C_\varnothing$.

D'autre part, il est possible d'intégrer EAC* à FDAC* pour exploiter les avantages de chacun. On parle alors de *cohérence d'arc existentielle et directionnelle.*

Définition 6 *Un RCP est EDAC* s'il est FDAC* et EAC*.*

Ainsi, EDAC* impose que chaque valeur ait un support complet dans une direction et un support simple dans l'autre (pour satisfaire FDAC*). De plus, une valeur au moins de chaque variable doit avoir un support complet dans les deux directions (pour satisfaire EAC*). Cette dernière valeur est appelée valeur *complètement supportée.* Notons que dans le cas des RC, EDAC* se réduit aussi en cohérence d'arc.

EDAC* est par construction plus forte que FDAC*. Elle est plus faible que FAC* car pour chaque variable, seule une de ses valeurs est totalement supportée. L'exemple 3 rappelle que trouver un support complet pour plus d'une valeur n'est pas toujours possible.

5 Algorithme

Nous présentons ici un algorithme qui établit EDAC* dans le cas des RCP binaires. EDAC* (ALG. 3) transforme un problème arbitraire en une instance équivalente vérifiant la cohérence locale EDAC*. L'idée générale en est la suivante :

- On tente de trouver pour chaque variable x_i sa valeur complètement supportée pour établir EAC* ; s'il n'y en a pas, on propage les coûts de façon à en trouver un.
- On tente de trouver pour chaque variable x_j un support complet par rapport à toute contrainte binaire qui s'applique sur elle et une variable x_i telle que $i < j$ pour établir DAC*.
- On tente de trouver pour chaque variable x_j un support complet par rapport à toute contrainte binaire qui s'applique sur elle pour établir AC*.
- On tente de trouver pour chaque variable toutes ses valeurs dont le coût unaire est trop élevé.

On peut accélérer l'algorithme en remarquant que l'établissement de DAC* sur x_i par rapport à une contrainte C_{ij} telle que $i < j$ rend inutile la recherche de support pour AC* ou de support existentiel pour EAC* dans cette direction.

De plus, établir une propriété peut en casser une autre. Pour remédier à celà, on rentre les variables dont une propriété est éventuellement violée dans une des trois files Q, R et S qui sont implantées comme des files de priorité telles que la plus petite valeur (ou la plus grande) peut être retirée en temps constant. La signification des files est la suivante :

- Si x_j est dans Q, alors une de ses valeurs a été supprimée et dans ce cas les plus grands voisins de x_j peuvent avoir perdu leur support et doivent être vérifiés (traitement de la propriété AC* pour $j > i$).
- Si x_j est dans R, alors le coût unaire d'une de ses valeurs est passé de $\bot$ à une valeur de coût strictement supérieure et dans ce cas les plus petits voisins de x_j peuvent avoir perdu leur support complet et doivent être vérifiés (traitement de la propriété DAC* pour $j < i$).
- Si x_i est dans P, alors soit le coût unaire de la valeur complètement supportée pour x_i a été augmenté, soit le coût unaire d'une des valeurs de x_j (un plus petit voisin de x_i) est passé de $\bot$ à une valeur de coût strictement supérieure et, dans ce cas, x_i peut avoir perdu le support complet de sa valeur complètement supportée et doit être vérifié (traitement de la propriété EAC* pour $j < i$).

Il existe de plus une autre file auxiliaire, S, qui sert à remplir efficacement P.

L'algorithme est constitué d'une boucle principale et quatre boucles intérieures. Les boucles **while** aux lignes 5, 8 et 10 établissent respectivement EAC*, DAC* et AC*. La ligne 11 établit NC*. À chaque fois qu'un coût est projeté pour l'établissement d'une propriété, une autre propriété peut être violée. Les variables dont une cohérence locale est à vérifier sont stockées dans les files. L'établissement de NC*, AC* et DAC* est fait comme dans [8]. Nous nous concentrerons donc sur EAC*.

Cette propriété est principalement établie par ChercheSupportExistentiel(i), qui trouve un support existentiel pour x_i en donnant un support complet par rapport à chaque plus petit voisin x_j (l'établissement d'un support complet sur les plus grands voisins se fait avec DAC*). Lorsque l'on établit le support existentiel de x_i, la fonction de coût C_j diminue ou reste constante. Nous n'aurons donc pas à vérifier la cohérence existentielle de x_j ni la cohérence d'arc directionnelle des plus petits voisin de x_j. De plus, comme la cohérence existentielle est plus forte que AC*, il est inutile d'établir la cohérence d'arc entre x_i et x_j. Cependant, la cohérence d'arc directionnelle de la variable x_i doit être vérifiée car une valeur de x_i peut avoir vu son coût unaire augmenter (ligne 6). De même, la cohérence existentielle des plus grands voisins de x_i doit aussi être revue (ligne 7).

ÉlagueVariable() établit la cohérence de nœud après qu'un coût unaire ou $C_\varnothing$ a été augmenté (ligne 11). Cette fonction peut être exécutée plus souvent pour éviter certains examens superflus dans les boucles **tant que** sans que cela ne change la complexité dans le pire cas. Par souci de clarté de notre exposé, nous ne décrirons pas ici le cas où le problème est incohérent et où $C_\varnothing$ atteint $\top$.

Theorème 1 *La complexité de EDAC* est de $\mathcal{O}(ed^2 \max\{nd, \top\})$ en temps et $\mathcal{O}(ed)$ en espace.*

Preuve 1 *En ce qui concerne l'espace, on utilise la structure donnée par [2] pour avoir une complexité en $\mathcal{O}(ed)$.*

De plus, [8] prouve les points suivants :

- *ProjetteUnaire, Projette, Étend et ÉlagueVariable ont une complexité en $\mathcal{O}(d)$;*
- *ChercheSupport et ChercheSupportComplet ont une complexité en $\mathcal{O}(d^2)$.*

*Etudions maintenant la complexité de la boucle **tant que** dans EDAC*. La boucle à la ligne 8 établit DAC*. Comme décrit dans l'article précédemment cité, elle est en $\mathcal{O}(ed^2)$ car chaque variable est stockée dans R au maximum une fois, ce qui entraîne que chaque contrainte C_{ij} est vérifiée une seule fois (ligne 9). De même, la boucle ligne 10 qui établit AC* dans la direction des plus grandes variables est aussi en $\mathcal{O}(ed^2)$.*

Algorithme 1 : Algorithme propageant les coûts.

```
[Supprime les valeurs dont les coûts de la contrainte
 unaire C_i sont trop élevés ]
Fonction ÉlagueVariable(i) : booléen
    drapeau ← faux ;
    pour chaque a ∈ D_i faire
        si (C_∅ ⊕ C_i(a) ⪰ ⊤) alors
            D_i ← D_i\{a} ;
            drapeau ← vrai ;
    retourner drapeau ;

[Projette le minimum du coût de la contrainte
 unaire C_i sur la contrainte d'arité nulle C_∅ ]
Procédure ProjetteUnaire(i)
    α ← min_{a∈D_i}{C_i(a)} ;
    C_∅ ← C_∅ ⊕ α ;
    pour chaque a ∈ D_i faire C_i(a) ← C_i(a) ⊖ α ;

[Projette le coût α de la contrainte binaire C_ij sur la
 contrainte unaire C_i en a ]
Procédure Projette(i, a, j, α)
    C_i(a) ← C_i(a) ⊕ α ;
    pour chaque b ∈ D_j faire
        C_ij(a, b) ← C_ij(a, b) ⊖ α ;

[Projette le coût α de la contrainte unaire C_i en a
 sur la contrainte binaire C_ij ]
Procédure Étend(i, a, j, α)
    pour chaque b ∈ D_j faire
        C_ij(a, b) ← C_ij(a, b) ⊕ α ;
        C_i(a) ← C_i(a) ⊖ α ;
```

Algorithme 2 : Algorithmes établissant les supports

```
[Établit un support sur x_i par rapport à C_ij ]
Fonction ChercheSupport(i, j) : booléen
    drapeau ← faux ;
    pour chaque a ∈ D_i faire
        α ← min_{b∈D_j}{C_ij(a, b)} ;
        si (α ≻ ⊥ ∧ C_i(a) = ⊥) alors drapeau ← vrai ;
        Projette(i, a, j, α) ;
    ProjetteUnaire(i) ;
    retourner drapeau ;

[Établit un support complet sur x_i par rapport à C_ij]
Fonction ChercheSupportComplet(i, j) : booléen
    drapeau ← faux ;
    pour chaque a ∈ D_i faire
        P[a] ← min_{b∈D_j}{C_ij(a, b) ⊕ C_j(b)} ;
        si (P[a] ≻ ⊥ ∧ C_i(a) = ⊥) alors
            drapeau ← vrai ;
    pour chaque b ∈ D_j faire
        E[b] ← max_{a∈D_i}{P[a] ⊖ C_ij(a, b)} ;
    pour chaque b ∈ D_j faire Étend(j, b, i, E[b]) ;
    pour chaque a ∈ D_i faire Projette(i, a, j, P[a]) ;
    ProjetteUnaire(i) ;
    retourner drapeau ;

[Établit un support existentiel sur x_i ]
Fonction ChercheSupportExistentiel(i) : booléen
    drapeau ← faux ;
1   α ←
    min_{a∈D_i}{C_i(a) ⊕_{C_ij∈C, j<i} min_{b∈D_j}{C_ij(a, b) ⊕
    C_j(b)}};
    si (α ≻ ⊥) alors
2       pour chaque C_ij ∈ C, j < i faire
            drapeau ← drapeau ∧
            ChercheSupportComplet(i, j) ;
    retourner drapeau ;
```

La boucle de la ligne 5 établit EAC dans la direction des plus grandes variables. En un seul parcours de la boucle, une variable n'entrera jamais deux fois dans P car les variables x_i qui en ont été extraites sont toujours les plus petites de la file et les variables qui entrent dans la file sont plus grandes que x_i. La boucle* ***tant que*** *itère donc au maximum n fois. Cela donne une complexité de $\mathcal{O}(e)$ pour la ligne 7. La complexité amortie de ChercheSupportExistentiel est de $\mathcal{O}(ed^2)$ car cette fonction peut être appelée avec chaque variable en paramètre, et chaque contrainte binaire est examinée une fois aux lignes 1 et 2. La complexité de EAC* est donc aussi de $\mathcal{O}(ed^2)$.*

La ligne 4 prend un temps $\mathcal{O}(n)$. Le ***pour chaque*** *de la ligne 11 est en temps $\mathcal{O}(nd)$ (qui est inférieur à $\mathcal{O}(ed)$ car le graphe est supposé connexe). Au vu de ces différents résultats, la complexité à l'intérieur du* ***tant que*** *de la ligne 3 est de $\mathcal{O}(ed^2)$. Elle itère lorsque :*

- *soit Q n'est pas vide : le* ***pour chaque*** *de la ligne 11 a donc éliminé une valeur et ceci ne peut pas être fait plus de nd fois ;*
- *soit R n'est pas vide : AC* a inséré une variable dans R et ceci n'est possible que $\mathcal{O}(nd)$ fois, d'après le point précédent ;*
- *soit S n'est pas vide : un élément a été inséré lors de l'établissement de AC* ou DAC*. Dans le premier, on a vu que ce ne peut pas être fait plus de $\mathcal{O}(nd)$ fois. Le second cas se produit $\mathcal{O}(\max\{nd, \top\})$ fois car la condition de la ligne 8 est vraie lorsque :*
  - *soit AC* a introduit un élément dans R ($\mathcal{O}(nd)$ fois)*
  - *soit EAC* a introduit un élément ($\mathcal{O}(\top)$ fois car chaque fois que EAC* est violé, $C_\varnothing$ augmente).*

La complexité de l'algorithme dans le pire cas est donc $\mathcal{O}(ed^2 \max\{nd, \top\})$.

Algorithme 3 : Etablissement de EDAC*, initialement, $Q = R = S = \mathcal{X}$.

```
[Établit la propriété EDAC*                                          ]
Procédure EDAC*
3   tant que (Q ≠ ∅ ∨ R ≠ ∅ ∨ S ≠ ∅) faire
4     P ← {l | i ∈ S, l > i, C_il ∈ C} ∪ S ;
      S ← ∅ ;
5     tant que (P ≠ ∅) faire
        i ← popMin(P) ;
        si (ChercheSupportExistentiel(i)) alors
6         R ← R ∪ {i} ;
7         pour chaque C_ij ∈ C, j > i faire
            P ← P ∪ {j} ;
8     tant que (R ≠ ∅) faire
        j ← popMax(R) ;
9       pour chaque C_ij ∈ C, i < j faire
          si (ChercheSupportComplet(i, j)) alors
            R ← R ∪ {i} ;
            S ← S ∪ {i} ;
10    tant que (Q ≠ ∅) faire
        j ← popMin(Q) ;
        pour chaque C_ij ∈ C, i > j faire
          si (ChercheSupport(i, j)) alors
            R ← R ∪ {i} ;
            S ← S ∪ {i} ;
11    pour chaque i ∈ X faire
        si (ÉlagueVariable(i)) alors Q ← Q ∪ {i} ;
```

6 Résultats expérimentaux

Dans cette partie, nous dresserons une comparaison empirique entre MEDAC* et MFDAC* pour mettre en évidence l'intérêt du premier. Le filtrage EDAC* ayant tout intérêt à être utilisé dans un algorithme de séparation et d'évaluation, l'idée est de *maintenir* EDAC* durant l'exploration (d'où le nom MEDAC*). Nous avons implémenté l'algorithme en C en utilisant les tables de supports telles que décrites pour AC2001 [1]. Dans les problèmes non binaires, on attend l'instanciation partielle des contraintes jusqu'à ce qu'elles deviennent binaires. Pour la sélection de variable, on utilise la simple heuristique dom/deg qui donne la variable qui possède le plus faible ratio taille de domaine sur degré. Lorsque l'on a le choix, la valeur sélectionnée est celle qui possède le coût unaire le plus faible. L'ordre des variables est l'ordre naturel de leur indice. L'optimum de chaque instance est donné à l'algorithme SE comme étant la première borne supérieure. Les résultats sont donnés sur trois domaines facilement modélisables par des RCP.

Pour une formule en CNF donnée, **Max-SAT** doit trouver une instanciation complète comportant le plus grand nombre de clauses satisfaites possible. Nous avons généré et résolu des instances de Max-2SAT à 80 variables et des instances de Max-3SAT à 40 variables en variant le nombre de clauses grâce à $Cnfgen$[1], un générateur aléatoire de Max-kSAT. Il est à noter que ce générateur ne produit pas deux littéraux identiques ou opposés dans une même clause mais éventuellement deux clauses identiques.

Pour un RC sur-contraint donné, **Max-CSP** doit trouver une instanciation complète comportant le plus grand nombre de contraintes satisfaites possible. Nous avons généré 6 classes de problèmes binaires aléatoires dont les domaines possèdent 10 valeurs, comme proposé dans [8]. Les problèmes peuvent être :

- peu contraints si $e = 2,5 \times n$, très contraints si $e = \frac{n(n-1)}{8}$ ou fortement connexes ;
- lâches si $t = t^o$, resserrés si $t = d^2 - \frac{t^o}{4}$ (t représente le nombre de tuples interdits dans le problème et t^o, le plus petit t tel que le problème soit sur-contraint).

Dans Max-SAT, chaque classe possède 10 instances et dans Max-CSP, 50. On ne mentionne ici que les résultats moyens.

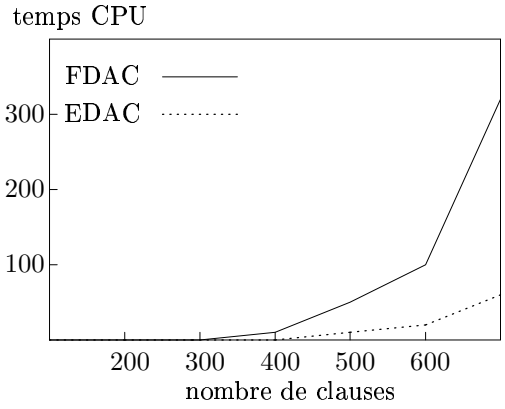
Dans le problème du *placement d'entrepôts* (UWLP[2]), une entreprise propose d'ouvrir des entrepôts à certains endroits pour fournir les magasins existants. L'objectif est de déterminer quels entrepôts ouvrir et à quel entrepôt assigner un magasin pour minimiser le coût de la maintenance et des coûts d'approvisionnement, sachant que chaque magasin est approvisionné par un unique entrepôt. Ce problème est modélisé par l variables booléennes pour les lieux candidats, s variables entières pour les magasins dont la taille du domaine est fixée à l, $l + s$ contraintes molles pour les coûts et $l \times s$ contraintes binaires dures reliant les magasins aux entrepôts. Nous avons résolu les problèmes cap71-134[3] des bancs de tests de la librairie standard OR pour UWLP ainsi que les instances MO*-MP* fournies dans [6]. Les instances M* sont particulièrement intéressantes lorsque résolues par programmation linéaire car elles ont un grand nombre de solutions sous-optimales.

Les résultats ont été obtenus sur un Pentium 4 à 2,8 GHz pour Max-SAT et Max-CSP et un Xeon 2,4 GHz pour UWLP. Les résultats sont donnés FIG. 3, 4, 5 et TAB. 5. Dans toutes les classes Max-SAT, la phase d'exploration semble grandir exponentiellement

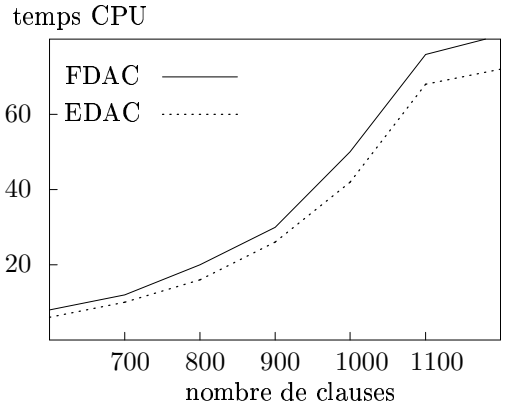
[1] A. van Gelder `ftp ://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances`

[2] CSPLib `http ://www.csplib.org/`, problème 034.

[3] J. E. Beasley `http ://www.dcc.unicamp.br/~aprox/facility/instancias/uncapacitated/uncapinfo.html.gz`.

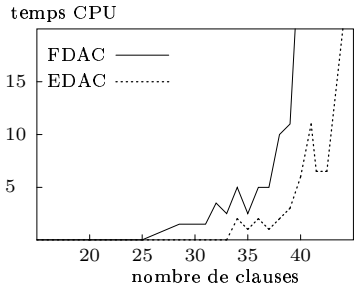


(a) CNF2 (80 variables)

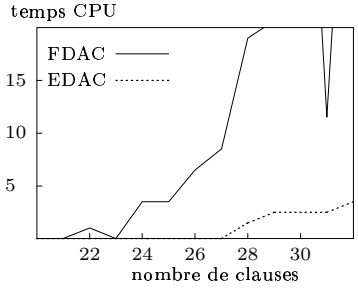


(b) CNF3 (40 variables)

FIG. 3 – Temps en secondes pour trouver l'instanciation optimale de problèmes Max-2SAT et Max-3SAT aléatoires.

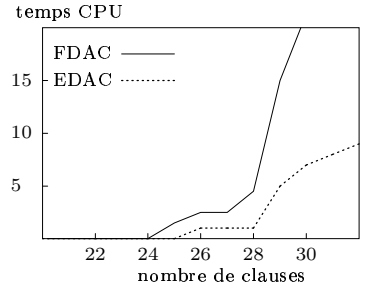


(a) peu contraint/lâche

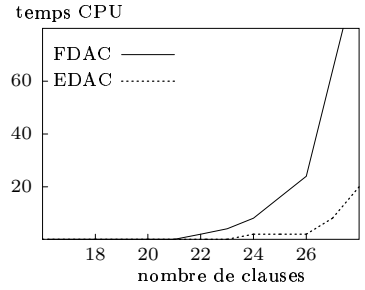


(b) peu contraint/resserré

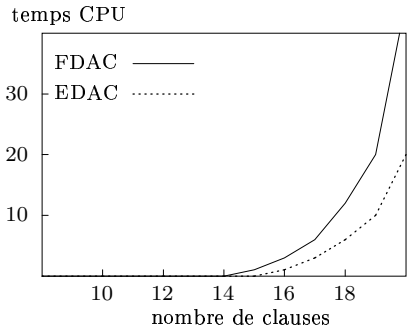
FIG. 4 – Temps en secondes pour trouver l'instanciation optimale de problèmes Max-CSP binaires aléatoires.



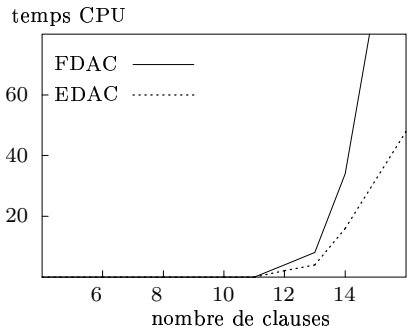
(a) très contraint/lâche



(b) très contraint/resserré



(c) fortement connexe/lâche



(d) fortement connexe/resserré

FIG. 5 – Temps en secondes pour trouver l'instanciation optimale de problèmes Max-CSP binaires aléatoires (suite).

avec le nombre de clauses, et pour Max-CSP, exponentiellement avec le nombre de variables. Nous avons sommé les temps de calcul pour tous les exemples qui ont été complètement résolus par les deux algorithmes et calculé le ratio MFDAC*/MEDAC*. Dans le cas de Max-2SAT, MEDAC* est 5,7 fois plus rapide que MFDAC*. L'amélioration est moindre dans Max-3SAT et nous supposons que cela est dû au fait que nous attendons l'instanciation des variables pour prendre en compte les contraintes d'arité supérieure à deux. Pour Max-CSP, MEDAC* est plus rapide que MFDAC* avec un facteur allant de 2,01 (pour « fortement connexe/beaucoup de tuples interdits » ou *Complete Tight*) à 13,8 (« peu contraint/beaucoup de tuples interdits » ou *Sparse Tight*). Des résultats identiques ont été observés lors de l'analyse du nombre de nœuds explorés par MEDAC* et MFDAC*. Pour résumer, le gain est d'autant plus substantiel que les instances sont peu contraintes.

Problème	Taille	MFDAC*	MEDAC*	GLPK
cap71	16 × 50	0,02	0,01	< 1
cap72	16 × 50	0,01	0,01	< 1
cap73	16 × 50	0,01	0,01	< 1
cap74	16 × 50	0,02	0,01	< 1
cap101	25 × 50	0,03	0,02	< 1
cap102	25 × 50	0,1	0,03	< 1
cap103	25 × 50	0,29	0,02	< 1
cap104	25 × 50	0,22	0,02	< 1
cap131	50 × 50	53	0,09	< 1
cap132	50 × 50	127	0,11	< 1
cap133	50 × 50	157	0,14	< 1
cap134	50 × 50	38	0,1	< 1
MO1	100 × 100	-	187	444
MO2	100 × 100	10035	39	86
MO3	100 × 100	12567	60	381
MO4	100 × 100	2890	25	229
MO5	100 × 100	1942	27	136
MP1	200 × 200	-	3668	7813
MP2	200 × 200	-	1488	8329
MP3	200 × 200	-	1061	5318
MP4	200 × 200	-	1889	8436
MP5	200 × 200	-	1189	5469

TAB. 5 – Temps en secondes pour trouver l'instanciation optimale pour UWLP. La taille du problème est $l \times s$, où l est le nombre d'entrepôts et s le nombre de magasins. Le signe « - » signifie que l'instance n'a pas été résolue en moins de 6 heures.

Pour UWLP, MEDAC* est de plusieurs ordres meilleur que MFDAC* pour des problèmes de taille supérieure ou égale à $l \times 50$. La borne inférieur calculée par EDAC* pour UWLP est donc meilleure et dépend moins de l'ordre des variables. Bien que UWLP soit un problème rapidement résolu par des modélisations dédiées ([4, 5]) ou par des heuristiques *ad hoc* ([6, 10]), on peut noter que notre algorithme générique est capable de résoudre le problème pour des instances de taille raisonnable dans un temps relativement court

(ici moins d'une heure). Nous avons confronté nos résultats avec le logiciel de programmation linéaire en nombres entiers GLPK 4.8[4] lancé avec les paramètres par défaut et sans borne supérieure initiale, en utilisant une formulation directe du problème. Le solveur Ilog CPLEX 9.0 donne pour l'instance MP5 avec la même formulation l'optimum en 2708 secondes sur un Sun Blade 100 (502 MHz sur SPARCV9). En tentant de rapprocher les résultats, CPLEX est approximativement 10 fois plus rapide que GLPK et 2,5 fois plus rapide que MEDAC*.

7 Conclusions et perspectives

Nous avons présenté dans cet article la nouvelle cohérence locale appelée EDAC*, adaptée aux RCP. Nous avons étudié sa complexité et, malgré son fort coût théorique dans le pire cas, il est montré que maintenir EDAC* donne toujours de meilleurs résultats que maintenir FDAC*, et ceci pour plusieurs classes de problèmes. Dans un futur proche, nous souhaiterions étudier une version paresseuse de EDAC* pour l'appliquer à d'autres classes de problèmes.

Références

[1] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *IJCAI-2001*, 2001.

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

[3] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving Max-SAT as weighted CSP. In *CP-03*, pages 363–376, 2003.

[4] D. Erlenkotter. A Dual-Based Procedure for Uncapacitated Facility Location. *Operations Research*, 26(6) :992–1009, 1978.

[5] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 39 :157–173, 1989.

[6] 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.

[7] J. Larrosa. On arc and node consistency in weighted CSP. In *Proc. AAAI'02*, 2002.

[8] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *IJCAI-03*, 2003.

[9] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[10] L. Michel and P. Van Hentenryck. A Simple Tabu Search for Warehouse Location. *European Journal on Operations Research*, 157(3) :576–591, 2004.

[11] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : hard and easy problems. In *IJCAI-95*, pages 631–637, 1995.

[4] A. Makhorin http ://www.gnu.org/software/glpk/glpk.html.