Une étude des supports résiduels pour la consistance d'arc

Christophe Lecoutre, Fred Hemery

Une étude des supports résiduels pour la consistance d'arc

Christophe Lecoutre **Fred Hemery**

CRIL - CNRS FRE 2499
Université d'artois
rue de l'université, SP 16
62307 Lens cedex, France

{lecoutre,hemery}@cril.univ-artois.fr

Résumé

Pour un algorithme établissant la consistance d'arc (AC), un support résiduel, ou résidu, est un support qui a été trouvé et enregistré lors d'une exécution de la procédure qui détermine si une valeur est supportée par une contrainte. Le point important est qu'un résidu n'offre pas la garantie de représenter un minorant du plus petit support courant de la valeur en question. Dans cet article, nous étudions l'impact théorique d'exploiter des résidus au niveau de l'algorithme élémentaire AC3. Tout d'abord, nous prouvons que AC3r(m) (i.e. AC3 exploitant des résidus) est optimal pour une dureté de contrainte faible ou élevée. Ensuite, nous montrons que MAC2001 présente, par rapport à MAC3r(m), un sur-coût en O(μed) par branche de l'arbre binaire construit par MAC, avec μ représentant le nombre de réfutations de la branche, e le nombre de contraintes et d la taille du plus grand domaine. L'une des conséquences est que, MAC3r(m) admet une complexité temporelle (dans le pire des cas) meilleure que MAC2001 pour une branche impliquant μ réfutations lorsque $\mu > d^2$ ou lorsque $\mu > d$ et que la dureté de chaque contrainte est soit faible soit élevée. Nos résultats expérimentaux montrent clairement que le fait d'exploiter des résidus permet d'améliorer l'efficacité des algorithmes MAC et SAC embarquant des algorithmes AC à gros grain.

Abstract

In an Arc Consistency (AC) algorithm, a residual support, or residue, is a support that has been stored during an execution of the procedure which determines if a value is supported by a constraint. The point is that a residue is not guaranteed to represent a lower bound of the smallest current support of a value. In this paper, we study the theoretical impact of exploiting residues with respect to the basic algorithm AC3. First, we prove that AC3r(m) (i.e. AC3 exploiting residues) is optimal for low and high constraint tightness. Second, we show that MAC2001 presents, with respect to MAC3r(m), an overhead in O(μed) per branch of the binary tree built by MAC, where μ denotes the number of refutations of the branch, e the number of constraints and d the greatest domain size. One consequence is that, MAC3r(m) admits a better worst-case time complexity than MAC2001 for a branch involving μ refutations when either $\mu > d^2$ or $\mu > d$ and the tightness of any constraint is either low or high. Our experimental results clearly show that exploiting residues allows enhancing MAC and SAC algorithms embedding coarse-grained AC algorithms.

1 Introduction

Il est bien connu que la consistance d'arc (AC pour Arc Consistency) joue un rôle central pour la résolution d'instances du problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem). En effet, l'algorithme MAC, i.e., l'algorithme qui maintient la consistance d'arc au cours de la recherche d'une solution est toujours considéré comme l'approche générique la plus efficace pour résoudre des instances difficiles de grande taille. De plus, AC est au coeur d'une consistance plus forte appelée singleton consistance d'arc (SAC pour Singleton Arc Consistency) qui a été l'objet récemment d'une certaine attention (e.g., [2, 8]).

Sur plus de 20 ans, de nombreux algorithmes ont été proposés pour établir la consistance d'arc. Aujourd'hui, les al-

gorithmes les plus référencés et utilisés sont certainement AC3 [10] grâce à sa simplicité et AC2001/3.1 [3] grâce à son optimalité (tout en n'étant pas trop complexe). La complexité temporelle dans le pire des case de AC3 et AC2001 est respectivement O(ed^3) et O(ed^2) où e représente le nombre de contraintes et d la taille du plus grand domaine. L'intérêt d'un algorithme optimal tel que AC2001 réside dans le fait qu'il est robuste. Cela signifie que, sur certaines instances, AC2001 peut être largement plus rapide qu'un algorithme tel que AC3 alors que le contraire n'est pas vrai. Cette situation apparaît lorsque la dureté des contraintes est élevée, comme ceci est le cas pour la contrainte d'égalité (i.e. la contrainte de type $X = Y$). En effet, comme on peut naturellement s'y attendre et comme cela est démontré plus loin, AC3 admet alors en pratique un comportement qui est proche du pire des cas, et la différence d'un facteur d entre les complexités théoriques devient une réalité.

Dans cet article, nous nous intéressons aux supports résiduels pour les algorithmes AC. Un support résiduel, plus simplement appelé résidu, est un support qui a été trouvé et enregistré lors d'une exécution de la procédure qui détermine si une valeur est supportée par une contrainte. Le point important est qu'un résidu n'offre pas la garantie de représenter un minorant du plus petit support courant de la valeur en question. L'algorithme élémentaire AC3 peut être affiné en exploitant les résidus comme suit : avant de rechercher un support pour une valeur en partant de zéro, la validité du résidu associé à la valeur est testée. On obtient alors un algorithme appelé AC3r, et lorsque la multi-directionalité est exploitée, un algorithme appelé AC3rm.

En fait, AC3r est un algorithme qui peut être avantageusement remplacé par AC2001 lorsque AC doit être simplement appliqué sur un réseau de contraintes donné. Cependant, lorsque AC doit être maintenu durant la recherche, MAC3r qui correspond à mac3.1residue [9] devient tout à fait intéressant et compétitif. De son côté, AC3rm est un algorithme de consistance d'arc à part entière car il exploite les résidus *multi-directionnels* tout comme AC3.2 [7]. Mais, observons l'intérêt d'exploiter les résidus.

Tout d'abord, nous prouvons dans cet article que AC3r et AC3rm, contrairement à AC3, admettent un comportement optimal lorsque la dureté des contraintes est élevée. Pour illustrer ceci, considérons le problème Domino introduit dans [3]. Toutes les contraintes (sauf une) correspondent à des contraintes d'égalité. Les résultats que nous obtenons lorsqu'on exécute AC3, AC2001, AC3.2 et le nouvel algorithme AC3rm sur certaines instances de ce problème sont données par le tableau 1. Nous ne considérons pas AC3r car celui-ci est toujours moins efficace que AC2001 (même si, sur ces instances très particulières, ils ont en fait le même comportement). Le temps en secondes (cpu) et le nombre de tests de consistance (ccks) sont donnés pour chaque instance de la forme $domino$-n-d où n représente le nombre de variables et d le nombre de valeurs dans chaque domaine. Clairement, AC3rm est une réponse à la faiblesse de AC3 pour les duretés élevées.

Instances		*AC3*	*AC3rm*	*AC2001*	*AC3.2*
domino	*cpu*	1.81	0.16	0.23	0.18
100-100	*ccks*	$18M$	$990K$	$1485K$	$990K$
domino	*cpu*	134	3.40	6.01	3.59
300-300	*ccks*	$1377M$	$27M$	$40M$	$27M$
domino	*cpu*	951	15.0	21.4	15.2
500-500	*ccks*	$10542M$	$125M$	$187M$	$125M$
domino	*cpu*	6144	60	87	59
800-800	*ccks*	$68778M$	$511M$	$767M$	$511M$

TAB. 1 – Coût pour établir AC sur les instances Domino

Ensuite, nous analysons le coût de gérer les structures de données par rapport aux retours-arrières. D'un côté, il est très facile d'embarquer AC3, AC3r ou AC3rm dans un algorithme de type MAC ou SAC car aucune maintenance des structures de données n'est nécessaire pendant la recherche effectuée par MAC ou l'inférence effectuée par SAC. De l'autre, embarquer un algorithme optimal tel que AC2001 implique un effort de développement non négligeable, avec en plus, un sur-coût à l'exécution. Pour MAC2001, ce sur-coût est O(μed) par branche d'un arbre binaire construit par MAC puisque nous devons prendre en compte la réinitialisation d'une structure (appelée $last$) qui enregistre les plus petits supports trouvés. Ici, μ représente le nombre de réfutations de la branche, e le nombre de contraintes et d la taille du plus grand domaine du réseau de contraintes.

Cet article est organisé comme suit. Tout d'abord, nous introduisons les réseaux de contraintes et présentons le nouvel algorithme AC3rm. Ensuite, nous montrons l'intérêt théorique d'utiliser AC3/AC3rm dans un algorithme de type MAC ou SAC. Après avoir présenté les résultats d'une expérimentation que nous avons menée, nous concluons.

2 Réseaux de contraintes

Un réseau (discret) de contraintes P est un couple $(\mathscr{X}, \mathscr{C})$ tel que $\mathscr{X}$ est un ensemble fini de n variables et $\mathscr{C}$ est un ensemble fini de e contraintes. Chaque variable $X \in \mathscr{X}$ possède un domaine *dom*(X) représentant l'ensemble des valeurs pouvant être affectées à X. Chaque contrainte $C \in \mathscr{C}$ implique un sous-ensemble de variables de $\mathscr{X}$, noté *vars*(C), et à une relation associée *rel*(C), qui représente l'ensemble des tuples autorisés pour les variables de *vars*(C). Le domaine initial (resp. courant) d'une variable X est noté dom$^{init}(X)$ (resp. dom(X)). Pour chaque contrainte C d'arité r tel que vars(C) = $\{X_1, \ldots, X_r\}$, nous avons : rel$(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$ où $\prod$ représente le produit cartésien. Également, pour tout élément $t = (a_1, \ldots, a_r)$, appelé tuple, de $\prod_{i=1}^{r} dom^{init}(X_i)$, $t[X_i]$ représente la valeur a_i. Il est également important de noter que, en considérant un ordre total sur les domaines, les tuples peuvent être ordonnés en utilisant un ordre lexi-

cographique $\prec$. Pour simplifier la présentation de certains algorithmes et sans perte de généralité, nous utiliserons deux valeurs spéciales $\perp$ et $\top$ telles que tout tuple t est tel que $\perp \prec t \prec \top$.

Définition 1. *Soit C une contrainte d'arité r telle que vars(C) = $\{X_1, \ldots, X_r\}$, un tuple t de $\prod_{i=1}^{r} dom^{init}(X_i)$ est dit :*
- *autorisé par C ssi $t \in rel(C)$,*
- *valide ssi $\forall X_i \in vars(C)$, $t[X_i] \in dom(X_i)$,*
- *un support dans C ssi il est autorisé par C et valide.*

Un tuple t sera dit être un support de (X_i, a) dans C lorsque t est un support dans C tel que $t[X_i] = a$. Déterminer si un tuple est autorisé est appelé un test de consistance, et déterminer si un tuple est valide est appelé un test de validité. Une solution est une assignation de valeurs à l'ensemble des variables telle que toutes les contraintes soient satisfaites. Un réseau est satisfiable lorsqu'il admet au moins une solution. Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem), qui consiste à déterminer si un réseau de contraintes donné est satisfiable, est NP-complet. Un réseau de contraintes est également appelé instance CSP.

La consistance d'arc (AC) reste la propriété centrale des réseaux de contraintes, et établir AC sur un réseau donné P consiste à éliminer toutes les valeurs qui ne sont pas arc-consistantes.

Définition 2. *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Un couple (X, a), avec $X \in \mathscr{X}$ et $a \in dom(X)$, est arc-consistant (AC) ssi $\forall C \in \mathscr{C} \mid X \in vars(C)$, il existe un support de (X, a) dans C. P est AC ssi $\forall X \in \mathscr{X}$, dom(X) $\neq \emptyset$ et $\forall a \in dom(X)$, (X, a) est AC.*

Les définitions suivantes seront utiles plus tard pour analyser la complexité temporelle dans le pire des cas de certains algorithmes.

Définition 3. *Une cn-valeur est un triplet de la forme (C,X,a) avec $C \in \mathscr{C}$, $X \in vars(C)$ et $a \in dom(X)$.*

Définition 4. *Soit (C,X,a) une cn-valeur telle que vars(C) = $\{X, Y\}$.*
- *Le nombre de supports de (X, a) dans C, noté $s_{(C,X,a)}$, correspond à la taille de l'ensemble $\{b \in dom(Y) \mid (a, b) \in rel(C)\}$.*
- *Le nombre de conflits de (X, a) dans C, noté $c_{(C,X,a)}$, correspond à la taille de l'ensemble $\{b \in dom(Y) \mid (a, b) \notin rel(C)\}$.*

On remarquera que le nombre de cn-valeurs qui peuvent être construites à partir d'un réseau de contraintes binaire est O(ed) avec d représentant la taille du plus grand domaine. Pour prendre en compte toutes les cn-valeurs, on notera : $\sum_{C,X,a}$.

Algorithme 1 doAC (P = $(\mathscr{X}, \mathscr{C})$) : booléen

```
1: Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ vars(C)}
2: while Q ≠ ∅ do
3:    choisir puis éliminer (C, X) de Q
4:    if revise(C,X) then
5:       if dom(X) = ∅ then retourner faux
6:          Q ← Q ∪ {(C', Y) | C' ∈ 𝒞, C' ≠ C, Y ≠ X, {X, Y} ⊆ vars(C')}
7: retourner vrai
```

3 Algorithmes AC à gros grain

Dans cette section, nous introduisons AC3 ainsi que AC3rm, et nous proposons une analyse précise de leurs complexités respectives. Mais, tout d'abord, il est important de remarquer que nos algorithmes sont donnés pour le cas général (i.e. ils peuvent être appliqués à des instances impliquant des contraintes d'arité quelconque). A strictement parler, leur description correspond à GAC3 et GAC3rm puisque pour des contraintes non binaires, la consistance d'arc est dite généralisée (GAC pour Generalized Arc Consistency). Toutefois, pour simplifier notre analyse, les complexités seront données pour le cas binaire. Plus précisément, pour les résultats théoriques, nous considérerons donné un réseau de contraintes binaire $P = (\mathscr{X}, \mathscr{C})$ tel que, pour simplifier et sans perte de généralité, chaque domaine contient exactement d valeurs.

Pour établir la consistance d'arc (généralisée) sur un réseau donné, on peut appeler la fonction $doAC$ (algorithme 1). Celle-ci retourne $vrai$ lorsque le réseau de contraintes donné peut être rendu arc-consistant. Elle est décrite dans le contexte d'un algorithme à gros grain, i.e. un algorithme qui fonctionne sur la base de couples de la forme (C,X) avec X représentant une variable impliquée dans la contrainte C. Initialement, tous les couples (C, X), appelés arcs, sont placés dans un ensemble Q. Une fois Q initialisée, chaque arc est révisé à tour de rôle (ligne 4), et quand une révision est effective (i.e. quand, au moins, une valeur a été éliminée), l'ensemble Q est mis à jour (ligne 6). Une révision est effectuée par un appel à la fonction $revise$ qui est spécifique à l'algorithme à gros grain choisi, et consiste à éliminer les valeurs de X qui sont inconsistantes par rapport à C. Cette fonction retourne $vrai$ quand la révision est effective. L'algorithme se termine lorsqu'un domaine vide est détecté (ligne 5) ou que l'ensemble Q devient vide.

3.1 AC3

Pour AC3 [10], chaque révision est effectuée par un appel à la fonction $revise3(C, X)$, décrite par l'algorithme 2. Cette fonction appelle de façon itérative la fonction $seekSupport3$ qui détermine à partir de zéro si, oui ou non, il existe un support (X, a) dans C. Cette dernière utilise $setNextTuple$ qui retourne soit le plus petit tuple valide

Algorithme 2 revise3(C, X) : booléen

1: nbElements $\leftarrow$ | $dom(X)$ |
2: **for** chaque $a \in dom(X)$ **do**
3: **if** seekSupport3$(C, X, a) = \top$ **then**
4: éliminer a de $dom(X)$
5: retourner nbElements $\neq$ |$dom(X)$|

Algorithme 3 seekSupport3(C, X, a) : Tuple

1: $t \leftarrow \bot$
2: **while** $t \neq \top$ **do**
3: **if** $C(t)$ **then** retourner t
4: $t \leftarrow setNextTuple(C, X, a, t)$
5: retourner $\top$

t' construit à partir de C tel que $t \prec t'$ et $t'[X] = a$, soit $\top$ si il n'en existe pas. Il est aussi à noter que $C(t)$ doit être compris comme un test de consistance et que $C(\bot)$ retourne $faux$.

AC3 a une complexité temporelle dans le pire des cas en $O(ed^3)$ [11]. Cependant, il est possible d'affiner ce résultat en se focalisant sur le coût cumulé de recherche de supports successifs pour un couple (X,a) par rapport à une contrainte C. Nous avons le résultat suivant[1].

Proposition 1. *Dans AC3, la complexité temporelle cumulée, dans le pire des cas, de $seekSupport3$ pour une cn-valeur (C,X,a) donnée est O($cd + s$) avec c = $c_{(C,X,a)}$ et s = $s_{(C,X,a)}$.*

Preuve. Évaluons le nombre maximum de tests de consistances qui peuvent être effectués (pendant le processus complet de propagation) lorsqu'on appelle itérativement $seekSupport3(C, X, a)$. Soient vars(C) = $\{X, Y\}$, c = $c_{(C,X,a)}$ et s = $s_{(C,X,a)}$. Notons que $d = c + s$. Le pire des cas se produit lorsque 1) une seule valeur est éliminée de dom$^{init}(Y)$ entre deux appels à $revise3(C, X)$, 2) les valeurs de dom$^{init}(Y)$ sont ordonnées de telle sorte que les c premières valeurs correspondent à des valeurs qui ne supportent pas a et les s dernières valeurs correspondent à des valeurs qui supportent a, 3) les $s - 1$ premières valeurs éliminées de dom$^{init}(Y)$ correspondent à des valeurs qui supportent a et les c valeurs suivantes éliminées de dom$^{init}(Y)$ correspondent à des valeurs qui ne supportent pas a. Ainsi, pour les s premiers appels à $seekSupport3(C, X, a)$, nous obtenons $s * (c + 1)$ tests de consistance. Pour les c appels suivants, nous obtenons $c + (c - 1) + \cdots + 1$ tests. Aussi, nous avons une complexité cumulée dans le pire des cas en O($sc + s + c^2$) = O($c(s + c) + s$)) = O($cd + s$). □

Il est intéressant de noter que c représente un paramètre de dureté (voir également [5], page 61). Lorsque la dureté des contraintes est faible (plus précisément, lorsque c est O(1)), la complexité temporelle cumulée dans le pire des cas devient $O(d)$, ce qui est optimal. Cependant, lorsque la dureté des contraintes est élevée (lorsque c est O(d)), elle devient $O(d^2)$. Sans grande surprise, ce résultat indique que AC3 est adapté aux instances impliquant des contraintes de faible dureté. On déduit le résultat suivant.

Proposition 2. *La complexité temporelle, dans le pire des cas, de AC3 est :*
O($d * \sum_{C,X,a} c_{(C,X,a)} + \sum_{C,X,a} s_{(C,X,a)}$).

3.2 AC3rm

En suivant le principe utilisé dans AC3.2 [7], nous proposons un mécanisme pour bénéficier partiellement de la multi-directionalité (positive). L'idée est que, lorsqu'un support t est trouvé, il peut être enregistré pour toutes les valeurs apparaissant dans t. Par exemple, considérons une contrainte binaire C telle que *vars*(C) = $\{X, Y\}$. Si (a, b) est trouvé dans C lorsqu'on recherche un support de (X, a) ou de (Y, b), dans les deux cas, il peut être enregistré comme étant le dernier support trouvé de (X, a) dans C et le dernier support trouvé de (Y, b) dans C. En fait, on peut simplement enregistrer pour toute cn-valeur (C,X,a) le dernier support trouvé de (X,a) dans C. Cependant, ici, à la différence de AC2001, en exploitant la multi-directionalité, nous ne pouvons plus bénéficier de l'uni-directionalité. Cela signifie que, lorsque le dernier support trouvé n'est plus valide, il est nécessaire de chercher un nouveau support à partir de zéro. En effet, en utilisant la multi-directionalité, nous n'avons pas la garantie que le dernier support trouvé corresponde au dernier plus petit support trouvé. Ce nouvel algorithme exige l'introduction d'un tableau à trois dimensions, appelé *supp*. Cette structure de données est utilisée pour enregistrer pour toute cn-valeur (C,X,a) le dernier support trouvé de (X,a) dans C. Initialement, tout élément de la structure *supp* doit être égal à $\bot$. Chaque révision (voir l'algorithme 4) implique de tester pour toute valeur la validité du dernier support trouvé (ligne 3) et si, cela n'est pas positif, une recherche pour un nouveau support est commencée à partir de zéro (à noter l'appel à la fonction $seekSupport3$). Si cette recherche aboutit, les structures correspondant aux derniers supports trouvés sont mises à jour (ligne 8).

Pour résumer, la structure *supp* permet d'enregistrer ce que nous appelons des résidus *multi-directionnels*. Bien sûr, il est possible d'exploiter des résidus plus simples [9], appelés ici résidus *uni-directionnels*, en n'exploitant pas la multi-directionalité. On peut dériver un nouvel algorithme, appelé AC3r, en remplaçant la ligne 8 de l'algorithme 4 par : $supp[C, X, a] \leftarrow t$

Cependant, avec AC3r, lorsque AC doit être simplement établi (et non pas maintenu) sur un réseau, plutôt que de

[1] Il est à noter que s ne peut être ignoré car c peut être égal à 0.

chercher un nouveau support à partir de zéro lorsque la valeur résiduelle n'est plus valide, il est plus naturel et efficace de réaliser la recherche en utilisant la valeur résiduelle comme point de départ. C'est exactement ce que fait AC2001. Cela signifie qu'en pratique, AC3r n'est intéressant (comme nous le verrons) que lorsqu'il est embarqué dans MAC [9] ou dans un algorithme SAC.

AC3rm a une complexité spatiale en $O(ed)$ et une complexité temporelle, dans le pire des cas, en $O(ed^3)$. Cependant, il est possible d'affiner ce résultat comme suit :

Proposition 3. *Dans AC3rm (et AC3r), la complexité temporelle cumulée, dans le pire des cas, de $seekSupport3$ pour une cn-valeur (C,X,a) donnée est O($cs+d$) avec c = $c_{(C,X,a)}$ et s = $s_{(C,X,a)}$.*

Preuve. Comme pour AC3, le pire des cas en terme de tests de consistance se produit lorsque, 1) seulement une valeur est éliminée de dom$^{init}(Y)$ entre deux appels à revise3rm(C,X), 2) les valeurs de dom$^{init}(Y)$ sont ordonnées de telle sorte que les c premières valeurs correspondent à des valeurs qui ne supportent pas a et les s dernières valeurs correspondent à des valeurs qui supportent a. Mais, à la différence de AC3, le pire des cas en terme de test de consistance est lorsque les s premières valeurs éliminées de dom$^{init}(Y)$ correspondent systématiquement aux derniers supports trouvés enregistrés par AC3rm (jusqu'à ce que le domaine devienne vide). Pour ces $s+1$ appels (à noter l'appel initial) à $seekSupport3(C,X,a)$, nous obtenons $s*(c+1)+c$ tests de consistance. Par ailleurs, le nombre des autres opérations (tests de validité et mises à jour de la structure supp) dans revise3rm réalisé par rapport à (X,a) est borné par d. Aussi, nous obtenons un complexité cumulée, dans le pire des cas, en O($sc+s+c+d$) = O($cs+d$). □

Ce qui est intéressant avec AC3rm, c'est le fait que, même si cet algorithme n'est pas optimal, il est adapté à des instances impliquant des contraintes de dureté faible ou élevée. En effet, lorsque la dureté des contraintes est faible (plus précisément, lorsque c est O(1)) ou élevée (lorsque s est O(1)), la complexité temporelle cumulée, dans le pire des cas, devient $O(d)$, ce qui est optimal. Par ailleurs, sc prend une valeur maximale lorsque $c=s=d/2$, ce qui correspond à une dureté de contraintes moyenne. Toutefois, on peut espérer que AC3rm ait un bon comportement en pratique pour les contraintes de dureté moyenne puisque l'espérance mathématique du nombre de tests de consistance pour trouver un support pour des contraintes aléatoires de dureté 0.5 est égale à 2. Nous pouvons déduire le résultat suivant.

Algorithme 4 revise3rm(C, X) : booléen

```
1: nbElements ← | dom(X) |
2: for chaque a ∈ dom(X) do
3:    if supp[C, X, a] est valide then continue
4:    t ← seekSupport3(C, X, a)
5:    if t = ⊤ then
6:       éliminer a de dom(X)
7:    else
8:       for chaque Y ∈ vars(C) do supp[C, Y, t[Y]] ← t
9: retourner nbElements ≠ | dom(X) |
```

	Espace	Temps
AC3	$O(e)$	O($d*\sum_{C,X,a} c_{(C,X,a)} + \sum_{C,X,a} s_{(C,X,a)}$)
AC3r(m)	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
AC2001	$O(ed)$	$O(ed^2)$
AC3.2	$O(ed)$	$O(ed^2)$

TAB. 2 – Complexités (pire des cas) pour établir AC.

	Dureté			
	Quelconque	Faible	Moyenne	Haute
AC3	$O(cd+s)$	$O(d)$	$O(d^2)$	$O(d^2)$
AC3r(m)	$O(cs+d)$	$O(d)$	$O(d^2)$	$O(d)$
AC2001	$O(d)$	$O(d)$	$O(d)$	$O(d)$
AC3.2	$O(d)$	$O(d)$	$O(d)$	$O(d)$

TAB. 3 – Complexités cumulées (pire des cas) pour rechercher successivement les supports d'une cn-valeur (C,X,a) donnée. Nous avons $c+s=d$.

Proposition 4. *La complexité temporelle, dans le pire des cas, de AC3rm (et AC3r) est :*
O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$).

Finalement, on peut remarquer que nous n'avons pas introduit AC3rm par rapport au cadre générique AC-* [12] puisque AC3rm n'est pas une instance de AC-*. Cependant, nous pensons qu'il devrait être possible d'étendre ce cadre pour prendre en compte le concept de résidu.

3.3 Rappel des complexités

Le tableau 2 indique les complexités d'ensemble dans le pire des cas pour établir la consistance d'arc en utilisant les algorithmes AC3, AC3r(m)[2], AC2001 et AC3.2 (à cause du manque de place, AC2001 et AC3.2 ne sont pas décrits dans cet article).

Il est également intéressant de regarder les complexités temporelles cumulées pour rechercher les supports successifs par rapport à une cn-valeur (C,X,a) donnée. Même si cela n'a pas été introduit ici, il est facile de montrer que les algorithmes optimaux admettent une complexité cumulée en O(d). En observant le tableau 3, nous apprenons que AC3 et AC3r(m) sont optimaux lorsque la dureté est faible

[2] AC3r(m) désignera de façon interchangeable les algorithmes AC3r et AC3rm.

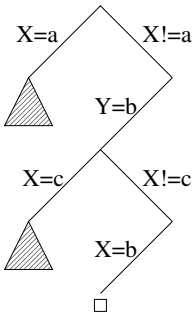

FIG. 1 – Une branche dans un arbre binaire

(i.e. c est O(1)), et que, contrairement à AC3, AC3r(m) est également optimal lorsque la dureté est élevée (i.e. s est O(1)).

4 Maintenir la consistance d'arc

Dans cette section, nous nous focalisons sur le maintien de la consistance d'arc au cours de la recherche. Plus précisément, nous étudions l'impact, en termes de temps et d'espace, que représente le fait d'embarquer les différents algorithmes AC dans MAC. L'algorithme MAC a pour objectif de résoudre des instances CSP et réalise une recherche en profondeur d'abord avec retours-arrières. A chaque étape de la recherche, une assignation de variable est effectuée suivie par un processus de filtrage qui correspond à établir (maintenir) la consistance d'arc.

Lorsqu'on mentionne MAC, il est important d'indiquer quelle politique de branchement est utilisée. En effet, il est possible de considérer un branchement binaire (2-way) ou non binaire (d-way). Ces deux schémas ne sont pas équivalents puisqu'il a été montré que le branchement binaire est plus puissant que le branchement non binaire [6]. Avec le branchement binaire, à chaque étape de la recherche, un couple (X,a) est sélectionné où X représente une variable non assignée et a une valeur dans dom(X), et deux cas sont considérés : le premier correspond à l'assignation $X = a$ et le second à la réfutation $X \neq a$. La figure 1 dépeint une branche dans un arbre binaire construit par MAC. Cette branche qui mène à une solution (représentée par le petit carré) inclut deux assignations de variable ($Y = b$ et $X = b$) et deux réfutations de valeurs ($X \neq a$ et $X \neq c$). Ces réfutations sont les conséquences de l'exploration de deux sous-arbres (à partir de $X = a$ et $X = c$).

Par ailleurs, il est important de remarquer que tous les algorithmes AC connus (incluant AC3r et AC3rm) sont incrémentaux. Un algorithme de consistance d'arc est incrémental si la complexité temporelle, dans le pire des cas, est la même qu'il soit appliqué une fois sur un réseau donné P ou qu'il soit appliqué jusqu'à $n(d-1)$ fois sur P avec, entre deux exécutions consécutives, au moins une valeur éliminée du domaine d'une variable. Du à l'incrémentalité, on obtient la même complexité, en termes de tests de consistance, pour une branche de l'arbre de recherche que pour le noeud racine de l'arbre de recherche.

Pour AC3 et AC3r(m), la complexité temporelle, dans le pire des cas, pour chaque branche de l'arbre de recherche est garantie (par l'incrémentalité) même si, entretemps, des sous-arbres ont été explorés et des retours-arrières effectués. Cependant, pour les algorithmes optimaux AC2001 et AC3.2, il est important de gérer la structure de données, appelé $last$, de manière à redémarrer la recherche, après avoir exploré un sous-arbre, comme si une exploration suivie d'un retour-arrière, n'avait jamais eu lieu. Dans ce papier, MAC2001 et MAC3.2 correspondent aux algorithmes qui enregistrent les plus petits supports qui ont été successivement trouvés tout au long de la branche courante. Notons que ceci est au prix d'une complexité spatiale en O(min(n,d)ed^2) [14]. Bien qu'une approche élégante pour éviter ce coût additionnel a été proposé dans [13], la méthode proposée, qui consiste à recalculer les derniers supports les plus petits, est complexe et les résultats préliminaires donnés par l'auteur sont décevants. Dans tous les cas (c'est à dire, quelque soit la variante), il faut faire attention à prendre en compte le coût temporel associé à l'exigence de maintenir la structure de données $last$ chaque fois que l'algorithme AC est appelé.

Proposition 5. *Dans MAC2001 et MAC3.2, la complexité temporelle cumulée, dans le pire des cas, de réinitialiser la structure $last$ est O(μed) pour toute branche impliquant μ réfutations.*

Preuve. Pour toute réfutation sur une branche, nous devons au préalable restaurer la structure $last$ puisque, sans cela, nous ne pourrions continuer à explorer l'arbre de recherche. Dans le pire des cas, nous avons au plus $e * 2 * d$ opérations puisque pour chaque cn-valeur (C,X,a), nous devons réinitialiser $last[C, X, a]$ à une valeur empilée (ou, pour les variantes, à $\perp$ ou une nouvelle valeur recalculée). En conséquence, nous obtenons (μed). □

Lorsque $\mu = 0$, cela signifie qu'une solution a été trouvée sans aucun retour-arrière. Dans ce cas, il n'est nul besoin de restaurer la structure $last$ puisque l'instance est résolue. A l'inverse, nous savons que la plus longue branche qui peut être construite contient nd arcs comme suit : pour chaque variable X, il y a exactement $d - 1$ arcs qui correspondent à des réfutations et seulement un arc qui correspond à une assignation. Aussi, nous obtenons une complexité temporelle cumulée pour réinitialiser la structure $last$ en O(end^2) et bien que cela soit omis ici, nous pouvons aussi prouver qu'elle est $\Omega(end^2)$.

L'une des caractéristiques intéressantes de AC3r(m) est que, lorsqu'il est embarqué dans MAC, aucune initialisation n'est nécessaire à chaque étape puisque le principe de

	Espace	Temps (par branche)
MAC3	$O(e)$	O($ed^2 + d * \sum_{C,X,a} c_{(C,X,a)}$)
MAC3r(m)	$O(ed)$	O($ed^2 + \sum_{C,X,a} c_{(C,X,a)} * s_{(C,X,a)}$)
MAC2001	$O(min(n,d)ed)$	$O(ed(d+\mu))$
MAC3.2	$O(min(n,d)ed)$	$O(ed(d+\mu))$

TAB. 4 – Complexités (pire des cas) de MAC. La complexité temporelle est donnée pour une branche impliquant μ réfutations.

cet algorithme est d'enregistrer le dernier support trouvé qui ne correspond pas nécessairement au dernier plus petit support trouvé. En fait, nous avons rapporté dans [7] que cela valait la peine de laisser inchangé les derniers supports trouvés (avec AC3.2) lorsqu'un retour-arrière apparaissait, ayant alors le bénéfice d'un effet mémoire. Cela signifie qu'un support trouvé à une profondeur donnée de la recherche a l'opportunité d'être encore valide à un noeud moins profond de la recherche (après retours-arrières). En d'autres termes, cela vaut la peine d'exploiter des résidus pendant la recherche. L'importance de limiter (voire, de supprimer) dans MAC la maintenance des structures de données utilisées par les algorithmes AC embarqués a été précisée dans [9] (mais aucun résultat de complexité n'a été donné). En fait, MAC3r correspond à l'algorithme mac3.1residue introduit dans [9].

En prenant en compte la proposition 5 et le tableau 2, nous obtenons les résultats donnés par le tableau 4. Il apparaît que, pour la plus longue branche, lorsque $\mu > d^2$, MAC3 et MAC3r(m) ont une complexité temporelle[3] meilleure que MAC2001 et MAC3.2 puisque nous savons que, pour toute branche, due à l'incrémentalité, MAC3 et MAC3r(m) sont O(ed^3). Par ailleurs, si la dureté de chaque contrainte est soit faible soit élevée (plus précisément, si pour toute cn-valeur (C,X,a), soit $c_{(C,X,a)}$ est O(1) soit $s_{(C,X,a)}$ est O(1)), alors MAC3r(m) admet une complexité temporelle en O(ed^2) par branche. Dans ce cas, MAC3r(m) surclasse MAC2001 dés que $\mu > d$. Ces observations suggèrent que MAC3r(m) devrait être très compétitif.

5 Établir la singleton consistance d'arc

Il y a un intérêt récent autour des singleton consistances, et plus précisément, autour de SAC (Singleton Arc Consistency), comme cela est illustré par certains travaux récents [2, 8]. Un réseau de contraintes est singleton arc-consistant ssi tout test singleton n'aboutit pas à une inconsistance, i.e., ssi après avoir effectué n'importe quelle assignation de variables, établir la singleton consistance d'arc sur le réseau obtenu n'entraîne pas un domaine vide. Une question intéressante est la suivante : Qu'en est-t-il du coût de restaurer les structures AC dans le contexte des algorithmes SAC ?

Par manque de place, nous nous focaliserons sur SAC-1 [4]. En fait, puisque n'importe quel algorithme AC peut être embarqué dans SAC-1, la question soulevée ci-dessus est pertinente lorsqu'on considère un algorithme optimal tel que AC2001. Le pire des scénarios (cas) revient à considérer O(n^2d^2) tests singletons, chacun ayant un coût en O(ed^2). Cependant, de manière à utiliser un algorithme AC optimal sans sacrifier l'espace mémoire, on doit réinitialiser la structure $last$ chaque fois qu'un test singleton a été réalisé. Pour SAC-1, la complexité temporelle cumulée, dans le pire des cas, de réinitialiser les structures de AC2001 structures est alors O(en^2d^3). Le même résultat est valable pour SAC-2 [1]. Bien que ce soit moins fort que O(en^2d^4), la complexité temporelle de SAC-1, ceci peut avoir un impact significatif sur la complexité temporelle moyenne. Nous verrons ceci avec les expérimentations.

6 Expérimentations

Pour comparer les différents algorithmes mentionnés dans cet article, nous avons mené une expérimentation (sur un PC Pentium IV 2,4GHz 512MB sous Linux) par rapport à des problèmes réels, académiques et aléatoires. Les performances ont été mesurées en termes de temps CPU en secondes (cpu) et du nombre de tests de consistance (ccks). Pour MAC et SAC-1, nous avons utilisé une version légèrement modifiée de AC3r(m) : les plus petits supports trouvés lors de la phase de pré-traitement sont enregistré de manière à sauver des tests de consistance. Un nouvelle recherche pour un support ne commence donc pas à zéro mais à partir de ces valeurs enregistrées. Il faut noter que les complexités temporelles et spatiales restent les mêmes.

Pour commencer, nous avons testé MAC (equipé avec dom/deg) en considérant 7 classes d'instance aléatoires binaires situées à la transition de phase pour la recherche. Pour chaque classe $\langle n,d,e,t\rangle$, définie de façon usuelle, 100 instances ont été générées. La dureté t représente la probabilité qu'un couple de valeurs soit autorisé par une relation. Ce qui est intéressant ici, c'est qu'un échantillon représentatif d'instances avec des domaines de taille différente, des densités différentes et des duretés différentes est considéré. Dans le tableau 5, on peut observer les résultats obtenus avec MAC embarquant différents algorithmes de consistance d'arc. Clairement, les meilleurs algorithmes à embarquer sont AC3 et AC3r(m) lorsque la dureté est faible (ici 0.1) et AC3r(m) lorsque la dureté est élevée (ici, 0.9). AC3r(m) est également le meilleur lorsque la dureté est moyenne (ici 0.5) comme on pouvait s'y attendre sur les instances aléatoires. Tous ces résultats sont confirmés pour les instances réelles et académiques que nous avons sélectionnées (et qui sont représentatives du type de résultats que nous avons obtenus). Il est clair que MAC3r(m) domine tous les autres algorithmes MAC pour le temps d'exécution tandis que MAC3.2 est le meilleur (quoique

[3] On considère ici qu'un test de consistance est O(1) pour une contrainte binaire.

		$MAC\ embarquant$				
		$AC3$	$AC3r$	$AC3rm$	$AC2001$	$AC3.2$
Classes d'instances aléatoires (résultats moyens pour 100 instances)						
$\langle$40-8-753-0.1$\rangle$	cpu	22.68	21.71	22.96	34.38	33.35
	$ccks$	$81M$	$17M$	$17M$	$24M$	$16M$
$\langle$40-11-414-0.2$\rangle$	cpu	21.19	18.76	19.23	27.91	26.34
	$ccks$	$97M$	$23M$	$22M$	$28M$	$19M$
$\langle$40-16-250-0.35$\rangle$	cpu	21.86	18.03	18.31	25.18	23.38
	$ccks$	$121M$	$28M$	$28M$	$33M$	$24M$
$\langle$40-25-180-0.5$\rangle$	cpu	37.35	24.52	25.53	35.30	32.27
	$ccks$	$233M$	$57M$	$56M$	$60M$	$45M$
$\langle$40-40-135-0.65$\rangle$	cpu	37.62	26.74	26.62	35.98	34.45
	$ccks$	$344M$	$85M$	$83M$	$82M$	$64M$
$\langle$40-80-103-0.8$\rangle$	cpu	89.01	50.37	51.62	67.74	61.48
	$ccks$	$1072M$	$243M$	$240M$	$225M$	$180M$
$\langle$40-180-84-0.9$\rangle$	cpu	166.12	75.12	76.99	98.69	87.50
	$ccks$	$2540M$	$514M$	$506M$	$479M$	$381M$
Instances académiques						
ehi-85-12	cpu	394	362	377	557	511
	$ccks$	$642M$	$58M$	$60M$	$190M$	$83M$
geo-50-20-19	cpu	194	148	157	278	263
	$ccks$	$1117M$	$249M$	$244M$	$284M$	$199M$
qa-5	cpu	31.60	27.21	28.31	37.49	36.02
	$ccks$	$130M$	$37M$	$36M$	$38M$	$27M$
qcp-819	cpu	139	136	143	215	208
	$ccks$	$116M$	$21M$	$21M$	$41M$	$25M$
Instances réelles						
$fapp$01-0200-9	cpu	0.54	0.39	0.37	0.60	0.60
	$ccks$	$6905K$	$3310K$	$3080K$	$3018K$	$2778K$
js-$enddr$2-3	cpu	53.66	28.24	29.08	39.24	29.60
	$ccks$	$596M$	$104M$	$104M$	$88M$	$48M$
$scen$-11	cpu	15.67	10.92	11.88	16.26	14.58
	$ccks$	$92M$	$18M$	$18M$	$15M$	$10M$
$graph$-10	cpu	0.64	0.53	0.54	0.69	0.68
	$ccks$	$4842K$	$2563K$	$2216K$	$2228K$	$1925K$

TAB. 5 – Coût de MAC

		$SAC-1\ embarquant$				
		$AC3$	$AC3r$	$AC3rm$	$AC2001$	$AC3.2$
Instances académiques						
$domino$-300-300	cpu	446.32	14.40	9.56	14.40	9.67
	$ccks$	$1376M$	$40M$	$26M$	$40M$	$26M$
$domino$-500-100	cpu	4.37	0.71	0.53	0.71	0.57
	$ccks$	$88M$	$7425K$	$4950K$	$7425K$	$4950K$
geo-50-20-19	cpu	1.18	0.73	0.74	1.49	1.24
	$ccks$	$9525K$	$1032K$	$1165K$	$2671K$	$1157K$
qa-5	cpu	1.22	0.82	0.89	1.91	1.34
	$ccks$	$10M$	$2700K$	$3104K$	$5001K$	$3085K$
Instances réelles						
$fapp$01-0200-9	cpu	637	153	158	312	192
	$ccks$	$10047M$	$838M$	$905M$	$1795M$	$904M$
js-$enddr$2-3	cpu	58.95	13.11	12.35	24.66	14.38
	$ccks$	$980M$	$55M$	$54M$	$128M$	$55M$
$graph$-10	cpu	980	424	439	836	581
	$ccks$	$12036M$	$1073M$	$1307M$	$2467M$	$1303M$
$scen$-11	cpu	44.89	20.72	21.07	56.03	53.21
	$ccks$	$479M$	$30M$	$33M$	$52M$	$37M$

TAB. 6 – Coût de SAC-1

battu sur quelques instances par MAC3rm) pour le nombre de tests de consistance. De façon générale, on peut remarquer que MAC3r(m) et MAC2001 réalisent à peu prés le même nombre de tests de consistance. Comme, par ailleurs, MAC3r(m) n'exige pas la maintenance de structures de données, cela explique pourquoi il est le plus rapide. Ces résultats confirment ceux obtenus pour MAC3r (mac3.1residue) dans [9].

Ici, nous devons mentionner que MAC3rm et MAC3r (mac3.1residue) [9] ont des performances assez proches en ce qui concerne les instances binaires. Sur l'ensemble des résultats que nous avons collectés (et non tous présentés ici bien sûr), il apparaît que MAC3r est plus rapide que MAC3rm d'environ 5% sur les series $bqwh$, ehi, frb, geo, qa, qk, et que MAC3rm est plus rapide que MAC3r d'environ 5% sur les séries $hanoi$ et $pigeons$, et d'environ 20% sur la série $domino$ (où MAC3rm tire profit de la multi-directionalité).

Ensuite, nous avons testés les différents algorithmes pour SAC-1. Dans le tableau 6, on peut observer une nouvelle fois que, sur les instances $domino$ qui contiennent des contraintes de dureté élevée, AC3r(m) démontre sa supériorité à AC3. Pour les instances réelles, l'écart entre AC3r(m) et les autres algorithmes semble augmenter. Par exemple, au coeur de SAC-1, AC3r(m) est environ trois fois plus efficace que AC2001 sur $scen11$ et environ quatre fois plus efficace que AC3 sur $fapp$01-0200-9.

7 Résidus pour le cas non binaire

On peut légitimement se poser la question du comportement d'un algorithme qui exploite des résidus quand il est appliqué à des instances non binaires. Tout d'abord, il est important de remarquer que rechercher un support à partir de zéro pour une cn-valeur nécessite de passer en revue O(d^{r-1}) tuples dans le pire des cas pour une contrainte d'arité r. On obtient alors une complexité temporelle cumulée, dans le pire des cas, pour rechercher des supports successifs à une cn-valeur donnée en O(r^2d^r) pour GAC3 et O(rd^{r-1}) pour GAC2001 [3] puisque nous considérons qu'un test de consistance est O(r) et puisqu'il y a O(rd) appels potentiels à la fonction $seekSupport$ spécifique. Aussi, on peut observer qu'il y a une différence d'un facteur dr entre les 2 algorithmes qui croit linéairement avec r (si on considère d constant). On peut obtenir pour GAC3r(m) des résultats assez semblables à ceux obtenus pour le cas binaire. Toutefois, par manque de place, nous ne les détaillons pas.

Nous avons réalisé une expérimentation préliminaire en maintenant GAC durant la recherche sur des instances non binaires aléatoires. Ces instances appartiennent aux classes de la forme $\langle r, n, d, e, t\rangle$ où r représente l'arité des contraintes tandis que les autres paramètres sont définis de façon habituelle. Ici, nous avons choisi des contraintes d'arité 6 et étudié le comportement de l'algorithme pour une dureté $t \in \{0.55, 0, 75, 0.95, 0.99\}$. Pour les faibles valeurs de t, nous avons observé (comme dans le cas binaire) que la différence entre les différents algorithmes était limité. Sur ces instances aléatoires, on peut observer dans le tableau 7 que GAC3rm et GAC3.2 sont les algorithmes embarqués les plus efficaces. Lorsque la dureté est élevée, GAC3 est pénalisé et GAC3r est plus efficace que GAC3rm puisqu'il bénéficie de la multi-directionalité. Sur les instances structurées non binaires de la compétition, on peut également observer le bon comportement de GAC3r et GAC3rm. Bien sûr, nous pensons qu'un étude expérimentale plus poussée incluant des instances structurées comportant des contraints d'arité importante reste à effectuer.

8 Conclusion

Dans ce papier, nous avons introduit des résultats théoriques concernant l'utilisation de supports résiduels, ou résidus, pour les algorithmes de consistance d'arc. La notion de résidu a été introduite sous sa forme multi-directionnelle dans [7] et sous sa forme uni-directionnelle dans [9]. Nous avons tout d'abord prouvé que l'algorithme de base AC3, qui est optimal pour une dureté de contrainte faible, devient également optimal pour une dureté de contrainte élevée lorsqu'il est étendu pour prendre en compte les résidus multi-directionnels ou uni-directionnels. De plus, on peut espérer de ces extensions à AC3, respectivement appelés AC3r et AC3rm, un bon comportement en pratique pour un dureté de contrainte moyenne puisque, assymptotiquement, pour une contrainte aléatoire, 2 tests de consistance sont nécessaires pour trouver un support lorsque la dureté est 0.5. Ensuite, nous avons montré que MAC3r(m) admet une complexité temporelle, dans le pire des cas, meilleure que celle de MAC2001 pour une branche (incluant μ réfutations) de l'arbre de recherche lorsque $\mu > d^2$ ou lorsque $\mu > d$ et que la dureté de chaque contrainte est soit faible soit élevée.

Sur le plan pratique, nous avons effectué une campagne expérimentale en testant les algorithmes MAC et SAC-1 sur des instances binaires et non binaires. Les résultats que nous avons obtenus montrent clairement l'intérêt d'exploiter des résidus puisque AC3r(m) (embarqué dans MAC ou SAC-1) fût toujours l'algorithme le plus rapide (seulement battu par AC3.2 sur certaines instances non binaires). Cela confirme pour MAC3r (mac3.1residue) les résultats présentés dans [9]. Une expérimentation plus poussée mérite toutefois d'être menée sur les instances non binaires, en particulier en prenant en compte des contraintes de grande arité. En termes de tests de consistance, il apparaît que AC3r(m) est très proche de AC2001 (mais, souvent battu par AC3.2). Nous avons aussi noté que AC3rm était plus robuste que AC3r sur les instances non binaires composées de contraintes de dureté élevée.

		$MGAC$ *embarquant*				
Instances		$GAC3$	$GAC3r$	$GAC3rm$	$GAC2001$	$GAC3.2$
Instances aléatoires (résultats moyens pour 10 instances)						
⟨6-20-6-32-0.55⟩	*cpu*	0.75	0.50	0.46	0.58	0.49
	ccks	$676K$	$357K$	$278K$	$364K$	$235K$
⟨6-20-6-36-0.55⟩	*cpu*	13.1	8.7	8.0	10.2	8.5
	ccks	$12M$	$6481K$	$4997K$	$6825K$	$4324K$
⟨6-20-8-22-0.75⟩	*cpu*	2.5	1.5	1.3	1.6	1.3
	ccks	$2313K$	$1240K$	$971K$	$1232K$	$804K$
⟨6-20-8-24-0.75⟩	*cpu*	51.7	31.8	27.7	34.6	26.8
	ccks	$48M$	$26M$	$20M$	$26M$	$17M$
⟨6-20-10-13-0.95⟩	*cpu*	35.2	20.7	15.8	20.8	13.9
	ccks	$40M$	$23M$	$17M$	$22M$	$14M$
⟨6-20-10-14-0.95⟩	*cpu*	220	135	102	135	89
	ccks	$249M$	$151M$	$108M$	$149M$	$91M$
⟨6-20-20-10-0.99⟩	*cpu*	659	392	267	254	177
	ccks	$1653M$	$1037M$	$647M$	$662M$	$462M$
⟨6-20-20-15-0.99⟩	*cpu*	869	489	301	351	220
	ccks	$2255M$	$1289M$	$785M$	$887M$	$583M$
Instances structurées						
tsp-20-366	*cpu*	387	242	243	266	235
	ccks	$607M$	370	$364M$	$387M$	$333M$
gr-44-9-*a*3	*cpu*	73.1	37.2	38.4	56.3	43.6
	ccks	$166M$	$44M$	$41M$	$74M$	$33M$
gr-44-10-*a*3	*cpu*	2945	1401	1465	2129	1631
	ccks	$6819M$	1513	$1527M$	$2914M$	$1224M$
series-14	*cpu*	233	218	217	312	285
	ccks	$1135M$	531	$490M$	$618M$	$422M$
renault	*cpu*	25.0	25.4	16.2	25.2	15.2
	ccks	$68M$	$66M$	$42M$	$66M$	$42M$

TAB. 7 – Coût de MGAC

Finalement, nous tenons à souligner la facilité d'implantation de (G)AC3r(m), et de son incorporation au sein d'un algorithme MAC ou SAC, puisqu'aucune maintenance de structures de données n'est nécessaire lors de retours-arrières. On pourra reprocher l'absence de comparaison avec les algorithmes à grain fin mais, nous nous sommes concentrés ici sur les algorithmes à gros grain et, de plus, les éléments de comparaison donnés dans [3] montrent que AC2001 est tout à fait compétitif vis à vis de AC6

Références

[1] R. Bartak and R. Erben. A new algorithm for singleton arc consistency. In *Proc. of FLAIRS'04*, 2004.

[2] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.

[3] C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[4] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.

[5] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[6] J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.

[7] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

[8] C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.

[9] C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc consistency in MAC : a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.

[10] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[11] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25 :65–74, 1985.

[12] J.C. Régin. Ac-* : a configurable, generic and adaptive arc consistency algorithm. In *Proceedings of CP'05*, pages 505–519, 2005.

[13] J.C. Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *Proceedings of CP'05*, pages 520–533, 2005.

[14] M.R.C. van Dongen. Saving support-checks does not always save time. *AI Review*, 21(3-4) :317–334, 2004.