Algorithmes Optimaux et Sous-optimaux de Singleton Consistance d'Arc

Christian Bessiere, Romuald Debruyne

Algorithmes Optimaux et Sous-optimaux de Singleton Consistance d'Arc

Christian Bessiere[1] **Romuald Debruyne**[2]

[1] LIRMM (CNRS/Université de Montpellier),
161 Rue Ada, 34392 Montpellier Cedex 5, France.
[2] École des Mines de Nantes, LINA/FRE CNRS 2729,
4 rue Alfred Kastler, 44307 Nantes Cedex 3, France
bessiere@lirmm.fr romuald.debruyne@emn.fr

Résumé

La singleton consistance d'arc (SAC) permet de filtrer bien plus que la consistance d'arc en s'assurant qu'aucune affectation ne puisse rendre le réseau de contraintes arc inconsistant. Des algorithmes ont été proposés pour réaliser la fermeture par singleton consistance d'arc mais avec des complexités temporelles clairement non optimales. Dans cet article, nous donnons une borne inférieure de la complexité temporelle dans le pire des cas des algorithmes de singleton consistance d'arc. Nous proposons un algorithme possédant cette borne pour complexité temporelle. Cet algorithme optimal est cependant coûteux en espace. C'est pourquoi nous proposons également un algorithme troquant l'optimalité temporelle pour une meilleure complexité spatiale. Bien que non optimal, ce dernier possède une complexité temporelle dans le pire des cas inférieure à celle de tous les algorithmes de singleton consistance d'arc déjà publiés. Une étude expérimentale met en valeur les bonnes performances des algorithmes proposés.

Abstract

Singleton arc consistency (SAC) enhances the pruning capability of arc consistency by ensuring that the network cannot become arc inconsistent after the assignment of a value to a variable. Algorithms have already been proposed to enforce SAC, but they are far from optimal time complexity. We give a lower bound to the time complexity of enforcing SAC, and we propose an algorithm that achieves this complexity, thus being optimal. However, it can be costly in space on large problems. We then propose another SAC algorithm that trades time optimality for a better space complexity. Nevertheless, this last algorithm has a better worst-case time complexity than previously published SAC algorithms. An experimental study shows the good performance of the new algorithms.

1 Introduction

S'assurer qu'une certaine consistance locale ne va pas conduire à un échec si on la vérifie après l'affectation d'une variable est une idée diversement utilisée en raisonnement à base de contraintes. Elle a été utilisée (parfois sous le nom de 'shaving') dans les réseaux de contraintes à domaines continus en réduisant les affectations aux bornes des domaines et en s'assurant que la consistance de borne ne détecte pas l'inconsistance du problème [13]. Dans le cadre du problème SAT elle a été employée comme un moyen d'obtenir des heuristiques mieux informées pour l'algorithme DPLL [10, 14, 1]. Enfin, dans les problèmes de satisfaction de contraintes à domaines discrets (CSPs) elle a été introduite sous le nom de singleton consistance d'arc (SAC) dans [8] puis a été davantage étudiée, à la fois de façon théorique et expérimentale, dans [19, 9].

Quelques bonnes propriétés confèrent à la SAC un réel avantage sur les autres consistances locales améliorant l'incontournable consistance d'arc. Sa définition est plus simple que celles des autres consistances locales exotiques comme la consistance de chemin restreinte (RPC, [4]) ou la max-consistance de chemin restreinte (Max-RPC, [7]). Sa sémantique opérationnelle peut être comprise par un non-expert du domaine. De plus, réaliser la SAC ne supprime que des valeurs des domaines, et ne change donc pas la structure du problème, par opposition entre autres à la consistance de chemin (PC, [17]) et aux niveaux plus élevés de k-consistance [11]. Enfin, la SAC peut être implémentée en s'appuyant sur n'importe quel algorithme de consistance d'arc.

Des algorithmes de SAC ont déjà été proposés

(*SAC1* [8] et *SAC2* [3]) mais leur complexité temporelle est loin d'être optimale. Une étude de la complexité de la SAC fait d'ailleurs défaut. Dans cet article, nous étudions la complexité de la SAC (section 3) et nous proposons un algorithme de SAC, *SAC-Opt*, dont la complexité temporelle est optimale (section 4). Cependant, l'optimalité temporelle est atteinte au prix d'une importante complexité spatiale qui peut empêcher l'utilisation de *SAC-Opt* sur les problèmes de grande taille. C'est pourquoi nous proposons *SAC-SDS*, un algorithme de SAC moins coûteux en espace, en section 5. La complexité temporelle de cet algorithme n'est pas optimale mais elle demeure meilleure que celle des algorithmes de SAC précédemment proposés. L'étude expérimentale présentée en section 6 montre les bonnes performances à la fois de *SAC-Opt* et de *SAC-SDS* comparées à celles des précédents algorithmes de SAC.

2 Définitions

Un *réseau de contraintes* $P = (X, D, C)$ consiste en un ensemble fini de n variables $X = \{i, j, \ldots\}$, un ensemble de domaines $D = \{D(i), D(j), \ldots\}$, où le domaine $D(i)$ est l'ensemble fini d'au plus d valeurs que la variable i peut prendre, et un ensemble de e contraintes $C = \{c_1, \ldots, c_e\}$. Chaque contrainte c_k est définie par l'ensemble ordonné $var(c_k)$ des variables sur lesquelles elle porte et par l'ensemble $sol(c_k)$ des combinaisons de valeurs qui la vérifient. Une *solution* d'un réseau de contraintes est une affectation d'une valeur de son domaine à chaque variable telle que toutes les contraintes du réseau soient satisfaites. Quand $var(c) = (i, j)$, la contrainte est dite *binaire* et on désigne $sol(c)$ par c_{ij}.

Une valeur a dans le domaine d'une variable i (notée (i, a)) est arc consistante pour la contrainte c_k ssi il existe un *support* pour (i, a) sur c_k, c'est-à-dire une instanciation des autres variables de $var(c_k)$ à des valeurs de leurs domaines respectifs telle que prises conjointement, cette instanciation et l'affectation de a à i satisfassent c_k. Dans le cas contraire, (i, a) est dite *arc inconsistante*. Un réseau de contraintes $P = (X, D, C)$ est *arc consistant* ssi D ne contient aucun domaine vide et aucune valeur de D n'est arc inconsistante. On désigne par $AC(P)$ le réseau obtenu par suppression dans P de toutes les valeurs arc inconsistantes. Si $AC(P)$ comporte un domaine vide, P est dit arc inconsistant.

Définition 1 (Singleton consistance d'arc) *Un réseau de contraintes $P = (X, D, C)$ est **singleton arc consistant** ssi $\forall i \in X, \forall a \in D(i)$, le réseau $P|_{i=a} = (X, D|_{i=a}, C)$, obtenu en remplaçant $D(i)$ par le singleton $\{a\}$, n'est pas arc inconsistant. Si $P|_{i=a}$ est arc inconsistant, (i, a) est dite SAC inconsistante.*

3 Complexité de la SAC

SAC1 [8] et *SAC2* [3] ont tous deux une complexité temporelle dans le pire des cas en $O(en^2d^4)$ sur les réseaux de contraintes binaires. Mais cette complexité n'est pas optimale.

Theorème 1 *La meilleure complexité temporelle qu'on puisse espérer pour un algorithme réalisant la SAC sur des réseaux de contraintes binaires est en $O(end^3)$.*

Preuve. D'après [16], nous savons que déterminer si une valeur (i, a) est telle que l'AC ne vide pas de domaines dans $P|_{i=a}$ revient à vérifier si dans chaque domaine $D(j)$ il existe au moins une valeur b telle que $((i, a), (j, b))$ soit "chemin consistante sur (i, a)" (une paire de valeurs $((i, a), (j, b))$ est *chemin consistante sur* (i, a) ssi elle n'est pas supprimée lors de la réalisation de la chemin consistance sur les paires impliquant (i, a)). Pour chaque variable j, nous devons trouver une valeur $b \in D(j)$ telle que $((i, a), (j, b))$ soit chemin consistante sur (i, a). Dans le pire des cas, toutes les valeurs b de $D(j)$ doivent être considérées. Par conséquent, il peut être nécessaire de vérifier la chemin consistance de chaque paire $((i, a), (j, b))$ en recherchant une valeur (k, c) dans le domaine de chaque troisième variable k qui soit compatible à la fois avec (i, a) et (j, b). Cependant, il est inutile de considérer les variables k qui ne sont pas reliées à j puisque dans cette forme partielle de chemin consistance seules les paires impliquant (i, a) peuvent être supprimées. En utilisant un algorithme de chemin consistance optimal (mémorisant les supports), nous sommes assurés de ne considérer chaque valeur (k, c) qu'au plus une fois lors des différentes recherches d'une valeur compatible avec $((i, a), (j, b))$ dans $D(k)$. Prouver que la chemin consistance sur (i, a) ne conduit pas à un échec a donc une complexité dans le pire des cas en $\Sigma_{b \in D(j)} \Sigma_{c_{kj} \in C} \Sigma_{c \in D(k)} 1 = ed^2$. De plus, réaliser la chemin consistance sur (i, a) est complètement indépendant de la réalisation de la chemin consistance sur une autre valeur (j, b). En effet, une paire $((i, a), (j, b))$ peut être chemin consistante sur (i, a) mais devenir chemin inconsistante suite à la suppression de certaines paires $((k, c), (j, b))$ et donc être chemin inconsistante sur (j, b). Par conséquent, la complexité dans le pire des cas d'un algorithme de SAC est au moins en $O(\Sigma_{(i,a) \in D} ed^2) = O(end^3)$. □

4 Un Algorithme de SAC Optimal en Temps

SAC1 n'utilise aucune structure de données permettant de mémoriser quelles valeurs peuvent devenir SAC inconsistantes suite à la suppression d'une certaine valeur. En cas de suppression d'une valeur, *SAC1* doit par conséquent vérifier à nouveau la SAC consistance de toutes les valeurs encore présentes. *SAC2* a lui des structures de données qui utilisent le fait que si l'AC ne vide aucun domaine de $P|_{i=a}$ alors la SAC consistance de (i,a) est assurée tant que toutes les valeurs de $AC(P|_{i=a})$ demeurent présentes. Après la suppression d'une valeur (j,b), *SAC2* teste à nouveau la SAC consistance de toutes les valeurs (i,a) telles que (j,b) figurait dans $AC(P|_{i=a})$. Ceci conduit à une meilleure complexité temporelle que celle de *SAC1* mais les structures de données de *SAC2* ne sont pas suffisantes pour atteindre l'optimalité puisque *SAC2* peut perdre du temps à réaliser plusieurs fois l'AC dans un sous-problème $P|_{i=a}$ en redémarrant de zéro.

L'algorithme 1, appelé *SAC-Opt*, est un algorithme qui réalise la singleton consistance d'arc en $O(end^3)$, la meilleure complexité temporelle qu'on puisse espérer pour un tel algorithme (cf. Théorème 1).

L'idée principale de cet algorithme optimal est que nous ne voulons pas faire et refaire en démarrant de zéro (potentiellement nd fois) la consistance d'arc dans chaque sous-problème $P|_{j=b}$ chaque fois qu'on détecte la SAC inconsistance d'une valeur (i,a) (ce qui représente n^2d^2 réalisations de l'AC potentielles). Pour éviter ces coûteuses répétitions lors des appels successifs de l'algorithme d'AC, on duplique le problème nd fois, créant les structures de données pour chaque sous-problème $P|_{i=a}$, de telle sorte qu'on puisse bénéficier de l'incrémentalité de l'algorithme d'AC sur chaque sous-problème. Tous les algorithmes d'AC sont incrémentaux, en d'autres termes, leur complexité sur un problème P est la même qu'on ne fasse qu'un seul appel ou jusqu'à nd appels où deux appels consécutifs ne diffèrent que par la suppression d'un certain nombre de valeurs dans P.

Dans ce qui suit, P_{ia} est le sous-problème dans lequel *SAC-Opt* mémorise le domaine courant (noté D_{ia}) et la structure de données correspondant à la réalisation de l'AC dans $P|_{i=a}$. `propagAC`(P,Q) désigne la fonction qui propage incrémentalement la suppression de l'ensemble Q de valeurs dans le problème P alors qu'un appel initial à l'algorithme d'AC a déjà eu lieu, initialisant la structure de données de l'algorithme d'AC utilisé. Elle retourne faux ssi P est arc inconsistant.

SAC-Opt procède en deux étapes principales. Après avoir rendu le problème arc consistant (ligne 1), la boucle de la ligne 2 crée le problème P_{ia} pour chaque valeur a de chaque domaine $D(i)$ comme étant une copie de l'état courant de P. Puis, à la ligne 5, la suppression de chaque valeur différente de a pour i est propagée dans $P|_{i=a}$. Si un sous-problème P_{ia} est arc inconsistant, (i,a) est éliminée du problème "principal" P et cette suppression est propagée dans P (ligne 7). L'avantage de propager immédiatement dans le problème principal la SAC inconsistance de valeurs trouvée en ligne 5 est double. Premièrement, le sous-problème P_{kc} d'une valeur (k,c) retirée en ligne 7 avant sa sélection en ligne 2 ne sera jamais créé. Deuxièmement, les sous-problèmes P_{jb} créés après la suppression de (k,c) bénéficieront de cette propagation puisqu'ils sont créés par duplication de P (ligne 3). Pour chaque sous-problème P_{jb} déjà créé avec $a \in D_{jb}(i)$, (i,a) est placée dans Q_{jb} et P_{jb} est placé dans *PendingList* pour une future propagation (lignes 9–10).

Une fois la phase d'initialisation terminée, nous savons que *PendingList* contient toutes les valeurs (i,a) pour lesquelles des suppressions de valeurs SAC inconsistantes (mémorisées dans Q_{ia}) n'ont pas encore été propagées dans P_{ia}. La boucle de la ligne 12 propage ces suppressions (ligne 14). Quand la propagation échoue, cela signifie que (i,a) est elle-même SAC inconsistante. Elle est alors retirée du domaine D du problème principal à la ligne 16 et chaque sous-problème

Algorithm 1: `L'algorithme optimal de singleton arc consistance`

```
function SAC-Opt(inout P : Problem) : Boolean;
  /* init phase */;
1  if AC(P) then PendingList ← ∅ else return false;
2  foreach (i,a) ∈ D do
3      P_ia ← P /* copy only domains and data
       structures */;
4      D_ia(i) ← {a}; Q_ia ← D(i) \ {a};
5      if ¬propagAC(P_ia, Q_ia) then
6          D ← D \ {(i,a)};
7          if propagAC(P, {(i,a)}) then
8              foreach P_jb such that (i,a) ∈ D_jb do
9                  Q_jb ← Q_jb ∪ {(i,a)};
10                 PendingList ← PendingList ∪ {(j,b)};
11         else return false;

  /* propag phase */;
12 while PendingList ≠ ∅ do
13     pop (i,a) from PendingList;
14     if propagAC(P_ia, Q_ia) then Q_ia ← ∅;
15     else
16         D ← D \ {(i,a)};
17         if D(i) = ∅ then return false;
18         foreach (j,b) ∈ D such that (i,a) ∈ D_jb do
19             D_jb(i) ← D_jb(i) \ {a};
               Q_jb ← Q_jb ∪ {(i,a)};
20             PendingList ← PendingList ∪ {(j,b)};
21 return true;
```

P_{jb} contenant (i, a) est mis à jour ainsi que sa liste de propagation Q_{jb} pour une future propagation (lignes 18–20).

Une fois *PendingList* vidée, toutes les suppressions ont été propagées dans les sous-problèmes . Toutes les valeurs de D sont dès lors SAC consistantes.

Theorème 2 *SAC-Opt est un algorithme de singleton arc consistance ayant sur les réseaux de contraintes binaires une complexité temporelle optimale en $O(end^3)$ et une complexité spatiale en $O(end^2)$.*

Preuve. Correction. Supposons qu'une valeur a ait été injustement retirée de $D(i)$. (i, a) a été retirée soit en ligne 6 soit en ligne 16 parce que l'arc inconsistance de P_{ia} a été trouvée. Si P_{ia} a été identifié comme arc inconsistant pendant la phase d'initialisation (ligne 5) nous pouvons conclure immédiatement à la SAC inconsistance de (i, a). Procédons par induction pour l'autre cas. Supposons que toutes les valeurs précédemment supprimées en ligne 16 étaient SAC inconsistantes. Si l'arc inconsistance de P_{ia} est détectée en ligne 14 cela signifie que P_{ia} ne peut pas être rendu arc consistant sans la présence de certaines valeurs SAC inconsistantes dans son domaine. Par conséquent, (i, a) est SAC inconsistante et *SAC-Opt* est correct.

Complétude. Grâce à la façon dont sont mises à jour *PendingList* et Q_{ia} aux lignes 8–10 et 18–20, nous savons qu'à la fin de l'algorithme, tous les sous-problèmes P_{ia} sont arc consistants. Par conséquent, pour toute valeur (i, a) restant dans P à la fin de *SAC-Opt*, $P|_{i=a}$ n'est pas arc inconsistant et (i, a) est donc SAC consistante.

Complexité. L'étude de complexité porte sur des réseaux de contraintes binaires puisque la preuve d'optimalité a été donnée uniquement pour ces réseaux (mais *SAC-Opt* peut être employé quelle que soit l'arité des contraintes). Puisque n'importe quel algorithme de consistance d'arc peut être utilisé dans l'implémentation de notre algorithme de SAC, les complexités temporelles et spatiales vont bien évidemment dépendre de ce choix. Traitons d'abord le cas où un algorithme optimal en temps, comme *AC-6* [5] ou *AC2001* [6, 20] est utilisé. La ligne 3 nous indique que la complexité spatiale sera nd fois celle de l'algorithme d'AC, c'est-à-dire $O(end^2)$. En ce qui concerne la complexité temporelle, la première boucle copie les structures de données et propage la consistance d'arc dans chaque sous-problème (ligne 5), deux tâches qui sont respectivement en $nd \cdot ed$ et $nd \cdot ed^2$. Dans la boucle `while` (ligne 12), la propagation de la consistance d'arc peut être appelée nd fois pour chacun des nd sous-problème. Cependant, *AC-6* et *AC2001* étant tous deux incrémentaux, la complexité des nd restrictions sur le même problème est en ed^2 et non en $nd \cdot ed^2$. Le coût total de la propagation de la consistance d'arc sur les nd sous-problèmes est donc en $nd \cdot ed^2$. Nous devons ajouter à cela la mise à jour des listes en lignes 8 et 18. Dans le pire des cas, les valeurs sont retirées une à une, et donc nd valeurs sont placées dans les nd listes Q, conduisant à une mise à jour de *PendingList* en n^2d^2. Si $n < e$, la complexité temporelle totale est en $O(end^3)$, ce qui est optimal. □

Remarques. Nous avons choisi de présenter les listes Q_{ia} comme des listes de valeurs dont la suppression doit être propagée parce que l'algorithme d'arc consistance utilisé n'est pas spécifié ici et parce qu'une suppression de valeur est l'information la plus fine que nous puissions avoir. Si un algorithme d'AC à grain fin est utilisé, comme *AC-6*, les listes seront utilisées comme indiqué. Si c'est un algorithme à gros grain qui est utilisé, comme *AC-3* [15] ou *AC2001*, seule est utile la connaissance des variables dont le domaine a été modifié. L'adaptation est immédiate.

Nous devons souligner que si *AC-3* est utilisé, la complexité spatiale diminue à $O(n^2d^2)$, mais la complexité temporelle est quant à elle augmentée à $O(end^4)$ du fait de la non optimalité d'*AC-3*.

5 Optimalité Temporelle contre Complexité Spatiale

SAC-Opt ne peut pas être utilisé sur des réseaux de contraintes de grande taille du fait de sa complexité spatiale en $O(end^2)$. Il semble de plus difficile d'atteindre l'optimalité temporelle avec une complexité spatiale moindre. Un algorithme de SAC doit maintenir l'AC sur nd sous-problèmes $P|_{i=a}$, et pour garantir une complexité temporelle en $O(ed^2)$ sur ces sous-problèmes nous devons recourir à un algorithme d'arc consistance optimal. Or, il n'existe pas d'algorithme d'AC optimal requiérant moins de $O(ed)$ en espace, ce qui conduit à un espace en $nd \cdot ed$ pour réaliser la SAC.

Nous proposons dans cette section d'abandonner l'optimalité temporelle afin d'obtenir un compromis satisfaisant entre l'espace requis et la complexité temporelle. Afin de ne pas discuter en des termes trop généraux, nous présentons cette idée sur une implémentation basée sur *AC2001* et destinée aux réseaux de contraintes binaires. Cette idée peut cependant être implémentée en utilisant n'importe quel algorithme optimal d'AC, comme *AC-6*, et avec des contraintes d'arité quelconque. L'algorithme *SAC-SDS (Sharing Data Structures)* tente d'utiliser autant que possible l'incrémentalité des algorithmes d'AC afin d'éviter un travail redondant, mais sans recourir à la duplication pour chaque sous-problème $P|_{i=a}$ des structures de

données requises par l'algorithme d'AC utilisé. Cet algorithme requiert donc moins d'espace que *SAC-Opt* mais n'est pas optimal en temps.

Comme dans *SAC-Opt*, *SAC-SDS* mémorise pour chaque valeur (i, a) le domaine local D_{ia} du sous-problème $P|_{i=a}$ ainsi que sa liste de propagation Q_{ia}. Notons que cette présentation faisant l'hypothèse de l'emploi de l'algorithme *AC2001*, Q_{ia} est une liste de variables j dont le domaine a été modifié (dans *SAC-Opt* il s'agit d'une liste de valeurs (j, b) supprimées de D_{ia}). Comme dans *SAC-Opt*, grâce aux domaines locaux D_{ia} nous savons quelles valeurs (i, a) peuvent ne plus être SAC consistantes suite à la suppression d'une valeur (j, b) : il s'agit des valeurs (i, a) telles que (j, b) était dans D_{ia}. Ces domaines locaux sont également utilisés pour éviter de recommencer à zéro chaque nouvelle phase de propagation d'AC dans les sous-problèmes. En effet, D_{ia} est exactement le domaine à partir duquel il convient de poursuivre la réalisation de l'AC suite à la suppression d'une valeur. A titre de comparaison, chaque nouvelle phase de propagation de la consistance d'arc dans *SAC1* et *SAC2* recommence sur le domaine de P (refaisant un travail proche de celui réalisé lors de précédentes propagations) car ils ne mémorisent pas les domaines des différents sous-problèmes.

Mais l'idée principale de *SAC-SDS* consiste en ce que contrairement à *SAC-Opt*, il n'y a pas duplication pour chaque sous-problème P_{ia} des structures de données utilisées par l'algorithme optimal d'AC. La structure de données n'existe qu'au niveau du problème principal P. Dans le cas d'*AC2001*, la structure *Last* qui mémorise dans $Last(i, a, j)$ la plus petite valeur de $D(j)$ compatible avec (i, a) sur c_{ij} est créée et mise à jour uniquement dans P. Cependant, elle est utilisée par tous les sous-problèmes P_{ia} pour éviter de refaire des tests de consistance déjà réalisés dans P.

SAC-SDS (Algorithme 2) opère ainsi : après quelques initialisations (lignes 1–4), et tant que *PendingList* n'est pas vide, *SAC-SDS* retire une valeur (i, a) de *PendingList* et propage la consistance d'arc dans P_{ia} (lignes 6–9). Notons que "D_{ia}=*nil*" signifie qu'il s'agit de la première réalisation de l'AC dans P_{ia}, et D_{ia} doit donc être initialisé (ligne 8). Si P_{ia} est arc inconsistant, (i, a) est SAC inconsistante. Elle est par conséquent retirée de D (ligne 11) et cette suppression est propagée dans le problème principal P au moyen de la fonction `propagMainAC` (ligne 12). Chaque valeur (i, a) supprimée de P est placée dans l'ensemble *Deleted* (lignes 11 et 24). Cet ensemble est utilisé par `updateSubProblems` (ligne 13) pour que les valeurs retirées de P soient supprimées des sous-problèmes, et pour mettre à jour les listes Q_{jb} et *PendingList* pour une future propagation dans les sous-problèmes modifiés.

Algorithm 2: `L'algorithme` *SAC-SDS*

```
function SAC-SDS-2001(inout P : problem) :
    Boolean;
 1  if AC2001(P) then PendingList ← ∅ else return
    false;
 2  foreach (i, a) ∈ D do
 3      D_ia ← nil ; Q_ia ← {i};
 4      PendingList ← PendingList ∪ {(i, a)};
 5  while PendingList ≠ ∅ do
 6      pop (i, a) from PendingList ;
 7      if a ∈ D(i) then
 8          if D_ia = nil then
            D_ia ← (D \ D(i)) ∪ {(i, a)};
 9          if propagSubAC(D_ia, Q_ia) then Q_ia ← ∅;
10          else
11              D(i) ← D(i) \ {a} ; Deleted ← {(i, a)} ;
12              if propagMainAC(D, {i}, Deleted) then
13                  updateSubProblems(Deleted)
14              else return false;
15  return true;

function Propag Main/Sub AC(inout D : domain; in
    Q : set ;
                          inout Deleted : set ) : Boolean;
16  while Q ≠ ∅ do
17      pop j from Q ;
18      foreach i ∈ X such that ∃c_ij ∈ C do
19          foreach a ∈ D(i) such that
            Last(i, a, j) ∉ D(j) do
20              if ∃b ∈ D(j), b > Last(i, a, j) ∧ c_ij(a, b)
                then
21                  Last(i, a, j) ← b
22              else
23                  D(i) ← D(i) \ {a} ; Q ← Q ∪ {i} ;
24                  Deleted ← Deleted ∪ {(i, a)} ;
25          if D(i) = ∅ then return false;
26  return true;
    /* ··· in propagMainAC but not in propagSubAC */;

procedure updateSubProblems(in Deleted : set);
27  foreach (j, b) ∈ D | D_jb ∩ Deleted ≠ ∅ do
28      Q_jb ← Q_jb ∪ {i ∈ X | D_jb(i) ∩ Deleted ≠ ∅} ;
29      D_jb ← D_jb \ Deleted;
30      PendingList ← PendingList ∪ {(j, b)} ;
```

Il nous faut à présent insister sur la différence entre `propagMainAC`, qui propage les suppressions dans le problème principal P, et `propagSubAC`, qui propage les suppressions dans les sous-problèmes P_{ia}. Les parties placées dans des boites apparaissent dans `propagMainAC` mais pas dans `propagSubAC`. La fonction `propagSubAC`, utilisée pour propager la consistance d'arc dans les sous-problèmes, suit le même principe que l'algorithme de consistance d'arc. La seule différence tient au fait que la structure de données

de l'algorithme d'AC n'est pas modifiée. Si on utilise *AC2001*, cela signifie que *Last* n'est pas mise à jour en ligne 21. En effet, cette structure de données est utile pour réaliser l'AC plus rapidement dans les sous-problèmes (lignes 19–20) puisque nous savons qu'il n'y a pas de support pour (i, a) sur c_{ij} plus petit que $Last(i, a, j)$ dans P, et dans les sous-problèmes également par conséquent. Cependant, cette structure est partagée par l'ensemble des sous-problèmes. Elle ne doit donc pas être mise à jour par `propagSubAC`, sinon nous n'aurions plus la garantie que $Last(i, a, j)$ soit encore le plus petit support dans les autres sous-problèmes et dans P. La structure *Last* est par contre mise à jour lors de la réalisation de l'AC dans P par `propagMainAC`. La ligne 24, qui permet de mémoriser les valeurs retirées de D dans *Deleted*, est la seule différence avec *AC2001*. Ces suppressions effectuées dans P peuvent directement être répercutées dans les sous-problèmes au moyen de la fonction `updateSubProblems` sans qu'il ne soit utile de déterminer à nouveau leur inconsistance dans les sous-problèmes.

Theorème 3 *SAC-SDS est un algorithme de SAC ayant une complexité temporelle en $O(end^4)$ et une complexité spatiale en $O(n^2d^2)$ sur les réseaux de contraintes binaires.*

Preuve. Correction. Notons tout d'abord que la structure *Last* n'est mise à jour que lors de la réalisation de l'AC dans P de telle sorte que tout support de (i, a) dans $D(j)$ soit plus grand ou égal à $Last(i, a, j)$. Les domaines des sous-problèmes étant des sous-domaines de D, tout support d'une valeur (i, a) sur c_{ij} dans un sous-problème est également plus grand ou égal à $Last(i, a, j)$. Ceci justifie que `propagSubAC` puisse utiliser la structure *Last* sans crainte de rater un support (lignes 19–20). Une fois cette constatation faite, la correction découle des mêmes raisons que pour l'algorithme *SAC-Opt*.

Complétude. La complétude tient au fait que toutes les suppressions sont propagées. Après l'initialisation (lignes 2-4), $PendingList = D$ et par conséquent la boucle principale de *SAC-SDS* va traiter chaque sous-problème $P|_{i=a}$ au moins une fois. Chaque fois qu'une valeur (i, a) est trouvée SAC inconsistante dans P à cause de l'arc inconsistance de $P|_{i=a}$ (ligne 9) ou parce que la suppression de valeurs SAC inconsistantes rend (i, a) arc inconsistante dans P (ligne 23 de `propagMainAC` appelée en ligne 12), (i, a) est retirée des sous-problèmes (line 29), et *PendingList* et les listes de propagation locales sont mises à jour pour une future propagation (lignes 28 and 30). A la fin de la boucle principale, *PendingList* est vide. Toutes les suppressions ont donc été propagées et pour chaque valeur $(i, a) \in D$, D_{ia} est un sous-domaine arc consistant non vide de $P|_{i=a}$.

Nom de l'algorithme	Complexité temporelle	Complexité spatiale
SAC1	$O(en^2d^4)$	$O(ed)$
SAC2	$O(en^2d^4)$	$O(n^2d^2)$
SAC-Opt	$O(end^3)$	$O(end^2)$
SAC-SDS	$O(end^4)$	$O(n^2d^2)$

TAB. 1 – Les complexités dans le pire cas des algorithmes de SAC sur des contraintes binaires.

Complexité. La structure *Last* demande un espace en $O(ed)$. Chacun des nd domaines D_{ia} peut contenir nd valeurs et il y a au plus n variables dans les nd liste de propagation locales Q_{ia}. Puisque $e < n^2$, la complexité spatiale de *SAC-SDS-2001* est en $O(n^2d^2)$. Par conséquent, en termes d'espace requis, *SAC-SDS-2001* est comparable à *SAC2*.

Considérons à présent la complexité temporelle. *SAC-SDS-2001* commence par dupliquer les domaines et par propager la consistance d'arc dans chaque sous-problème (lignes 8 et 9), deux tâches qui sont respectivement en $nd \cdot nd$ et en $nd \cdot ed^2$. Chaque suppression de valeur est propagée à l'ensemble des sous-problèmes $P|_{i=a}$ au moyen d'une mise à jour de *PendingList*, Q_{ia} et D_{ia} (lignes 27–30). Ceci réclame $nd \cdot nd$ opérations. La propagation de la consistance d'arc peut être appelée au plus nd fois dans chacun des nd sous-problèmes. Les domaines de chaque sous-problème sont mémorisés de façon à ce que la propagation de l'AC dans les sous-problèmes commence toujours avec des domaines dans l'état où ils étaient à la fin de la précédente propagation. Ainsi, bien qu'il y ait plusieurs propagations d'AC sur un même sous-problème, une valeur ne sera toujours supprimée qu'au plus une fois, et grâce à l'incrémentalité de la consistance d'arc, la propagation de ces nd suppressions est en $O(ed^3)$ (La complexité optimale en ed^2 pour la consistance d'arc sur ces sous-problèmes ne peut pas être atteinte parce que la structure nécessaire pour garantir cette optimalité n'est pas dupliquée). Par conséquent, le coût total de la propagation de la consistance d'arc est en $nd \cdot ed^3$ et la complexité temporelle de *SAC-SDS-2001* est en $O(end^4)$. □

Tout comme *SAC2*, *SAC-SDS* améliore la propagation de *SAC1* puisque après la suppression d'une valeur (i, a) de D, *SAC-SDS* ne teste la consistance d'arc que des sous-problèmes $P|_{j=b}$ ayant (i, a) dans leurs domaines (et non tous les sous-problèmes comme le fait *SAC1*). Cette amélioration n'est cependant pas suffisante pour avoir une meilleure complexité temporelle que *SAC1*. *SAC2* a d'ailleurs la même complexité

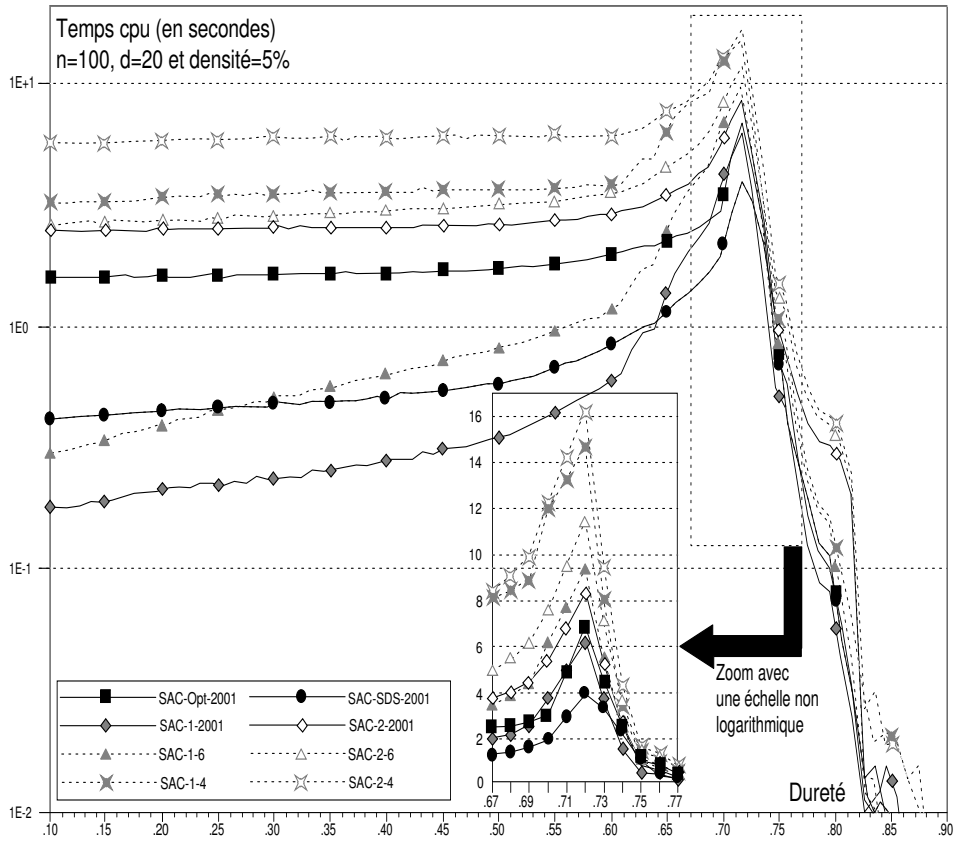

FIG. 1 – Temps cpu sur des problèmes avec $n = 100$, $d = 20$, et densité= .05.

temporelle en $O(en^2d^4)$ que *SAC1*. *SAC-SDS* améliore cette complexité parce qu'il mémorise le domaine *courant* de chaque sous-problème et peut ainsi commencer chaque propagation avec des domaines dans le même état qu'à la fin de la propagation précédente[1]. De plus, nous pouvons espérer une amélioration de la complexité temporelle en moyenne puisque la structure *Last* partagée permet de réduire le nombre de tests de contraintes requis, bien qu'elle ne suffise pas à obtenir une complexité temporelle dans le pire cas optimale. Ce gain potentiel ne peut être évalué qu'au moyen d'une comparaison expérimentale des différents algorithmes de SAC. Enfin, dans *SAC1* et *SAC2* chaque réalisation de l'AC dans un sous-problème doit être réalisée sur une nouvelle copie de D réalisée durant l'exécution (potentiellement $nd \cdot nd$ fois) alors qu'une telle duplication n'est réalisée qu'une seule fois pour chaque valeur dans *SAC-SDS* (en créant les sous-domaines D_{ia}).

Le tableau 1 récapitule les complexités des algorithmes de SAC en supposant qu'ils s'appuient sur un algorithme d'arc consistance optimal.

[1] Ceci est indépendant de l'algorithme d'AC utilisé et une version basée sur *AC-3* conserverait cette complexité en $O(end^4)$.

6 Résultats Expérimentaux

Nous avons comparé les performances de algorithmes de SAC sur des réseaux de contraintes binaires générés aléatoirement à l'aide du générateur de [12], lequel produit des instances du modèle B [18]. Tous les algorithmes ont été implémentés en C++ et exécutés sur un Pentium IV-1600 MHz disposant de 512 Mo de mémoire et ayant Windows XP pour OS. Différents algorithmes de consistance d'arc ont été utilisés. Dans ce qui suit, nous notons *SAC-1-X*, *SAC-2-X* et *SAC-Opt-X* les versions de *SAC1*, *SAC2* et *SAC-Opt* basées sur *AC-X*. L'implémentation de la liste de propagation de *SAC2* a été réalisée en tenant compte des recommandations faites dans [2]. Pour chaque combinaison de paramètres utilisée, 50 instances ont été générées et nous mentionnons ici la moyenne des temps cpu obtenus sur ces instances.

6.1 Réseaux de contraintes peu denses

La figure 1 montre les performances sur des réseaux de contraintes ayant 100 variables, des domaines comportant chacun 20 valeurs et une densité de 5% (248 contraintes). Ces réseaux sont relativement peu denses puisque les variables on chacune cinq voisins en

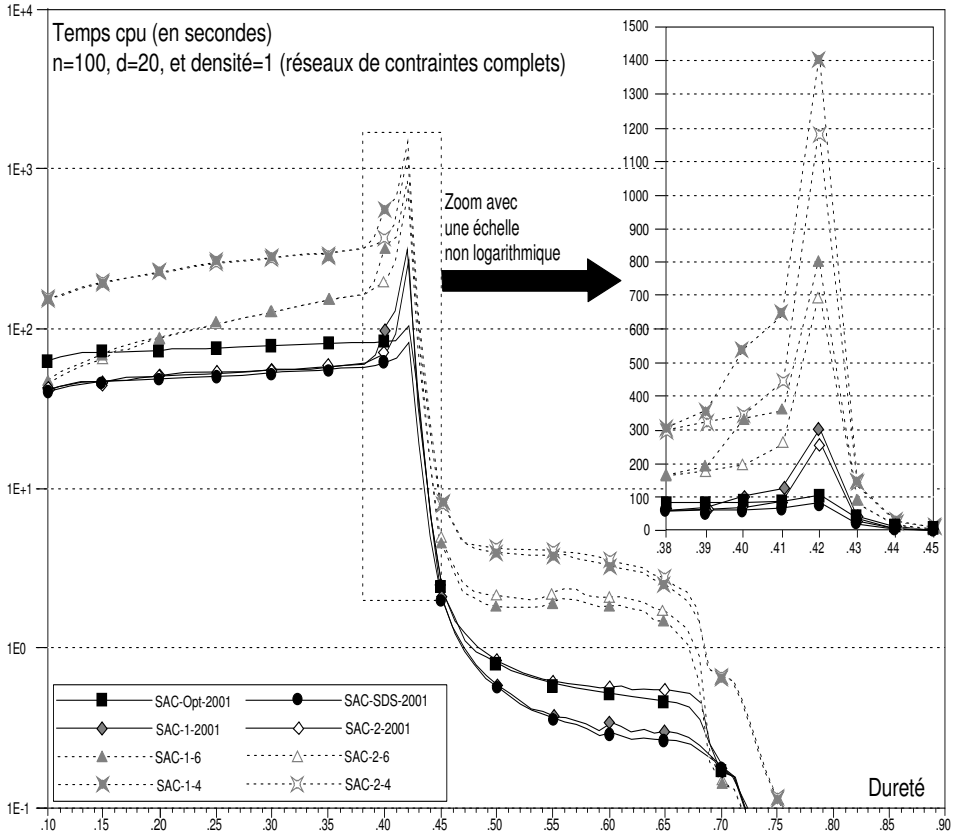

FIG. 2 – Temps cpu sur des problèmes avec $n = 100$, $d = 20$, et densité= 1.

moyenne.

Sur les problèmes dont la dureté ne dépasse pas 55% toutes les valeurs sont singleton arc consistantes. Sur ces réseaux sous-contraints les algorithmes de SAC vérifient la consistance d'arc de chaque sous-problème au plus une fois. Mémoriser des listes de supports (comme dans *SAC2*) ou des sous-domaines locaux (comme dans *SAC-Opt* et *SAC-SDS*) est inutile. Un algorithme brutal comme *SAC1* est suffisant et c'est *SAC-1-2001* qui montre les meilleures performances.

Sur des problèmes ayant des contraintes plus dures, quelques valeurs SAC inconsistantes sont supprimées. On peut observer un pic de complexité pour une dureté de 72%. Cependant, comme indiqué dans [2], la propagation améliorée de *SAC2* est inutile sur des réseaux de contraintes peu denses et *SAC2-X* (avec $X \in \{4, 6, 2001\}$) est toujours moins performant que *SAC1-X* sur les problèmes générés. Autour du pic de complexité, *SAC-SDS-2001* est clairement le plus performant. *SAC-Opt-2001* et *SAC1-2001* sont environ 1,7 fois moins rapides, et les autres algorithmes sont entre 2,1 et 10 fois plus lents.

6.2 Réseaux de contraintes denses

La figure 2 montre les performances sur des réseaux de contraintes binaires complets de 100 variables dont les domaines comportent chacun 20 valeurs.

SAC1 et *SAC2* ont des performances très proches. Quand toutes les valeurs sont SAC consistantes (dureté inférieure à 37%) la structure de données supplémentaire de *SAC2* est inutile puisqu'il n'y a aucune propagation. Cependant le coût de création de cette structure n'est pas important comparé au temps global et *SAC2* réclame donc à peu près le même temps que *SAC1*. Autour du pic de complexité, *SAC2-X* (avec X$\in \{4, 6, 2001\}$) est un peu plus rapide que *SAC1*. *SAC2* doit re-vérifier la consistance d'arc dans moins de sous-problèmes que *SAC1* mais chacune de ces vérification réclame autant de temps qu'avec *SAC1*. Sur des contraintes très dures, *SAC1* est plus rapide que *SAC2* puisque l'inconsistance du problème est trouvée avec très peu de propagation et la création de la structure de données de *SAC2* est inutile.

Contrairement à ce qui est supposé dans [3], il n'est pas préférable d'utiliser *AC-4* dans *SAC1* (ou dans *SAC2*) plutôt que *AC-6* ou *AC2001*. L'intuition consistait à penser que puisque la structure de données d'*AC-4* n'a pas à être mise à jour, le coût de sa

création serait faible comparé au profit qu'on pouvait en espérer. On peut cependant constater que *SAC1*-4 et *SAC2*-4 sont bien plus coûteux que leurs versions basées sur *AC-6* ou *AC2001*.

Les meilleurs résultats sont obtenus en utilisant *SAC-Opt-2001* et *SAC-SDS-2001* qui sont entre 2,6 et 17 fois plus rapides que les autres algorithmes au niveau du pic. Ces deux algorithmes ont une meilleure propagation que *SAC1* mais ils évitent également certaines tâches redondantes et réduisent ainsi le travail accompli sur chaque sous-problème.

7 Conclusion

Nous avons présenté *SAC-Opt*, le premier algorithme de SAC optimal en temps. Cependant, l'importante complexité spatiale de cet algorithme le rend inutilisable sur de grands réseaux de contraintes. C'est pourquoi nous avons proposé *SAC-SDS*, un algorithme de SAC dont la complexité temporelle n'est pas optimale, mais qui requiert moins d'espace que *SAC-Opt*. Une évaluation expérimentale montre les bonnes performances de ces nouveaux algorithmes comparées à celles des algorithmes précédemment proposés. Ceci conduit à envisager à nouveau l'utilisation de la SAC comme une alternative à la consistance d'arc pour filtrer les valeurs des réseaux de contraintes, ou du moins à l'utiliser dans les parties les plus "prometteuses" du réseau. La singleton consistance d'arc peut également être utilisée pour obtenir des heuristiques efficaces de choix des variables à instancier, de manière similaire à ce qui a été fait avec succès pour le problème SAT [1, 14].

Références

[1] Anbulagan. Extending unit propagation look-ahead of DPLL procedure. In *Proceedings PRICAI'04*, pages 173–182, Auckland, New Zealand, 2004.

[2] R. Barták and R. Erben. Singleton arc consistency revised. In *ITI Series 2003-153*, Prague, 2003.

[3] R. Barták and R. Erben. A new algorithm for singleton arc consistency. In *Proceedings FLAIRS'04*, Miami Beach FL, 2004. AAAI Press.

[4] P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE-CAIA'95*, 1995.

[5] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[6] C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.

[7] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings CP'97*, pages 312–326, Linz, Austria, 1997.

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

[9] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[10] J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.

[11] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11) :958–966, Nov 1978.

[12] D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform csp generators. URL : http ://www.ics.uci.edu/d̃frost/csp/generator.html, 1996.

[13] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI'93*, pages 232–238, Chambéry, France, 1993.

[14] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, 1997.

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

[16] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19 :229–250, 1979.

[17] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132, 1974.

[18] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81 :81–109, 1996.

[19] P. Prosser, K. Stergiou, and T Walsh. Singleton consistencies. In *Proceedings CP'00*, pages 353–368, Singapore, 2000.

[20] Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings IJCAI'01*, pages 316–321, Seattle WA, 2001.