
Les algorithmes d'arc-consistance dans les CSP dynamiques

Romuald Debruyne

*Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
161, rue Ada
34392 Montpellier Cedex 5*

RESUME. Les problèmes de satisfaction de contraintes (CSP) permettent de représenter sous une forme simple et agréable un grand nombre de problèmes [MES 89]. Leur résolution étant un problème NP-difficile, un grand intérêt fut porté aux algorithmes de filtrage qui simplifient le traitement en éliminant des inconsistances locales. Le plus utilisé de ces filtrages est celui réalisant l'arc-consistance [M&F 85].

Afin de représenter des problèmes exigeant un environnement dynamique comme on en rencontre en conception, en planification ou encore en ordonnancement, le formalisme fut étendu aux CSP dynamiques (DCSP). Des algorithmes d'arc-consistance incrémentaux sur les DCSP ont alors été proposés et ce sont ces algorithmes qui sont présentés dans cet article. Après de brefs rappels, DnAC-4 [BES 91], DnAC-6 [DEB 94] et AC|DC [N&B 94] sont étudiés en détail. Une étude comparative de leurs performances permet ensuite de mettre en valeur leurs avantages respectifs.

ABSTRACT. Constraint satisfaction problems (CSPs) provide a pleasant and simple representation for many problems [MES 89]. Since their solving is a NP-hard problem, filtering techniques which simplify the task by eliminating some local inconsistencies are particularly studied. Arc-consistency [M&F 85] is the most used filtering technique.

To represent some problems which require a dynamic environment, as we can find in design, planning or scheduling, the formalism was extended to dynamic CSPs (DCSPs). Then, some incremental arc-consistency algorithms were proposed and we present them in this paper. After some recalls, DnAC-4 [BES 91], DnAC-6 [DEB 94] and AC|DC [N&B 94] are studied in detail. A comparison of their performances shows their respective advantages.

MOTS-CLES : Problème de satisfaction de contraintes dynamique, arc-consistance.

KEY WORDS : Dynamic constraint satisfaction problem, arc-consistency.

1. Introduction

Le formalisme des problèmes de satisfaction de contraintes permet la représentation et le traitement d'un large spectre de problèmes. Leur attrayante simplicité a favorisé leur étude et tout un arsenal d'algorithmes traite le problème NP-difficile de leur résolution. Un grand intérêt a été porté aux algorithmes de filtrage qui simplifient les CSP avant ou durant la recherche de solutions. Les plus utilisés de ces algorithmes permettent d'obtenir un CSP équivalent en terme de solutions qui possède la propriété d'arc-consistance. Ils suppriment dans les domaines des valeurs qui de toute évidence ne peuvent pas faire partie d'une solution et réduisent ainsi l'espace de recherche.

Les CSP dynamiques furent introduits pour répondre à un réel besoin en intelligence artificielle. Bien que les CSP statiques couvrent une vaste gamme de problèmes, de nombreuses applications nécessitent un environnement dynamique. Un concepteur par exemple doit faire des restrictions pour compléter la définition de son problème mais doit aussi pouvoir retirer des contraintes quand il n'y a plus de solutions. Les procédures d'arc-consistance classiques peuvent très facilement être modifiées pour devenir incrémentales pour l'ajout de contraintes. Elles sont pourtant inefficaces sur les DCSP car elles ne peuvent pas déterminer l'ensemble des valeurs qui ont été retirées à cause d'une contrainte donnée. Par conséquent, pour maintenir la propriété d'arc-consistance lors d'une relaxation, elles doivent ajouter de nouveau toutes les contraintes du problème sur le CSP initial.

Cet article présente les algorithmes d'arc-consistance incrémentaux sur les DCSP à l'exception de celui figurant dans [PRO 92]. Ce dernier ne présente aucun véritable avantage sur ses concurrents. Notre étude commence par la présentation de DnAC-4 dont l'idée est de conserver une justification pour chaque valeur retirée afin de pouvoir déterminer les valeurs à remettre lors d'une relaxation. Cette adaptation de l'algorithme AC-4 [M&H 86] réalise peu de tests de consistances mais conserve une imposante complexité en espace. C'est pour pallier à cet inconvénient que fut créé DnAC-6. Nous verrons que cet algorithme requiert un espace moindre bien qu'il améliore les performances de DnAC-4 sur la plupart des DCSP. Nous étudierons également l'algorithme AC|DC qui fut développé parallèlement à DnAC-6. Cet algorithme n'utilise pas d'imposantes structures de données globale ce qui non seulement permet d'obtenir une excellente complexité en espace mais aussi de conserver une grande simplicité. Nous terminerons cet article par une étude des performances des algorithmes présentés.

2. Rappels et notations

Par souci de clarté, nous ne considérerons dans cet article que des DCSP binaires ayant des domaines discrets finis. Un CSP "statique" binaire $P=(X, dom, C)$ est défini par une séquence $X=(i, j, \dots)$ de n variables, chacune prenant ses valeurs dans un domaine discret figurant dans $dom=(dom[i], dom[j], \dots)$ (on notera $d=\max_{k \in X} |dom[k]|$) et par une séquence C de e contraintes binaires. Une contrainte binaire C_{ij} est un sous-ensemble du produit cartésien $dom[i] \times dom[j]$ qui stipule les couples de valeurs permis pour les variables i et j . On notera

$((i, a), (j, b)) \in C_{ij}$ pour spécifier que la contrainte autorise qu'il y ait à la fois la valeur a pour la variable i et b pour j . Le **taux de satisfiabilité** de la contrainte C_{ij} est le rapport entre le nombre de couples qu'elle autorise et $|dom[i] \times dom[j]|$. On dira qu'une contrainte est **dure** (resp. **molle**) si son taux de satisfiabilité est faible (resp. grand). A tout CSP est associé son **graphe des contraintes** qui possède les variables pour sommets et une arête entre i et j si et seulement s'il existe une contrainte C_{ij} . On se référera parfois à ce graphe en considérant le graphe orienté symétrique équivalent.

Une **solution** est une instantiation des variables qui ne viole aucune contrainte. $b \in dom[j]$ est un **support** pour la valeur $a \in dom[i]$ sur la contrainte C_{ij} ssi $((i, a), (j, b)) \in C_{ij}$. Une valeur $a \in dom[i]$ est dite **viable** si elle possède un support sur chaque contrainte portant sur i . Le domaine dom est **arc-consistant** si toutes ses valeurs sont viables. Le **domaine arc-consistant maximal** de $P = (X, dom, C)$ est l'union de tous les domaines arc-consistants pour P inclus dans dom . Etant donné un CSP $P = (X, dom, C)$ le problème du filtrage arc-consistant est de trouver le domaine arc-consistant maximal de P . Un **CSP dynamique** (DCSP) est une suite de CSP statiques $P_{(0)}, \dots, P_{(\alpha)}, P_{(\alpha+1)}, \dots$, chacun résultant du précédent par l'ajout ou le retrait d'une contrainte (ces opérations sont respectivement appelées restriction et relaxation). Par conséquent si on note $P_{(\alpha)} = (X, dom, C_{(\alpha)})$, on a $P_{(\alpha+1)} = (X, dom, C_{(\alpha+1)})$ avec $C_{(\alpha+1)} = C_{(\alpha)} \pm C_{ij}$ où C_{ij} est une contrainte. Au départ on a $P_{(0)} = (X, dom, \emptyset)$.

3. L'algorithme DnAC-4

3.1. Introduction

DnAC-4 fut créé pour pallier à l'inefficacité sur les DCSP des algorithmes d'arc-consistance existants. Il s'agit d'une adaptation de l'algorithme AC-4. Ce dernier réalise l'arc-consistance dans les CSP statiques en utilisant une structure de compteurs indiquant pour chaque paire arc-valeur $[(i, j), a]$ le nombre de supports de (i, a) sur C_{ij} dans le domaine courant. La détection des valeurs non viables se résume alors à tester la nullité de compteurs. Un ensemble de listes de valeurs supportées lui permet de propager les conséquences de chaque retrait de valeur (décrémentations de compteurs) et de n'effectuer jamais deux fois le même test de consistance. Facilement adaptable aux restrictions, AC-4 est toutefois inefficace pour le retrait de contraintes car il est dans l'incapacité de déterminer les valeurs à remettre lors d'une relaxation. Pour retirer une contrainte avec cet algorithme on est alors contraint de procéder à des restrictions sur le problème initial, refaisant un travail proche de celui déjà effectué.

L'idée principale de l'algorithme incrémental DnAC-4 est de conserver des informations permettant de connaître les valeurs à remettre lors du retrait d'une contrainte.

3.2. Les idées

On notera $dom[i]$ le domaine de la variable i dans $P_{(0)}$.

- A l'image d'AC-4, DnAC-4 utilise des compteurs qui indiquent pour chaque paire arc-valeur $[(i, j), \mathbf{a}]$ le nombre de supports de (i, \mathbf{a}) sur C_{ij} dans le domaine courant. Les valeurs non viables sont dès lors celles qui possèdent au moins un compteur nul. Pour maintenir cette structure on utilise des listes de valeurs : $S[(i, j), \mathbf{a}]$ est l'ensemble des valeurs supportées par (i, \mathbf{a}) sur C_{ij} . Ces listes permettent de connaître les valeurs dont le compteur doit être modifié quand une valeur est retirée ou remise. DnAC-4 réalise peu de tests de consistance car on utilise ces derniers uniquement pour la constitution des compteurs et des listes de valeurs supportées et on n'est donc jamais amené à effectuer deux fois le même test.

- Ces structures sont à la base de l'efficacité de AC-4 sur les CSP statiques mais elles ne sont pas suffisantes pour réaliser des relaxations de manière acceptable. DnAC-4 pallie à ce problème par l'utilisation d'un système de justification des retraits. Pour chaque valeur absente, la structure *Justif* conserve la cause de sa suppression, c'est à dire la première contrainte rencontrée sur laquelle la valeur n'a pas de support. Grâce à cette structure, lors du retrait d'une contrainte C_{ij} , on peut déterminer les valeurs dont la suppression est directement ou indirectement due à cette contrainte. Mais pour cela, les justifications doivent vérifier la propriété de bien-fondé.

◊ Les justifications sont **bien-fondées** ssi dans tout ensemble E de valeurs retirées, il existe une valeur (i, \mathbf{a}) telle que, sa justification étant la contrainte C_{ij} , aucune des valeurs de j compatible avec (i, \mathbf{a}) sur la contrainte C_{ij} n'appartient à E .

◊ Soit R l'ensemble des valeurs retirées du domaine. Les justifications sont **bien-fondées** ssi il existe un ordre φ sur R tel que : $\forall (i, \mathbf{a}) \in R$, soit C_{ij} la contrainte justifiant le retrait de (i, \mathbf{a}) , $\forall (j, \mathbf{b}) / ((i, \mathbf{a}), (j, \mathbf{b})) \in C_{ij} : (j, \mathbf{b}) <_{\varphi} (i, \mathbf{a})$.

Ces deux définitions équivalentes expriment que des justifications bien-fondées sont sans circuits. Regardons un exemple simple de justifications qui ne vérifient pas cette propriété. Considérons que nous avons un problème à trois variables i, j et k et qu'un algorithme d'arc-consistance ai retiré les valeurs (i, \mathbf{a}) , (j, \mathbf{a}) et (k, \mathbf{a}) en les justifiant par $Justif[i, \mathbf{a}] = C_{ij}$, $Justif[j, \mathbf{a}] = C_{jk}$ et $Justif[k, \mathbf{a}] = C_{ki}$. Si de plus les valeurs (i, \mathbf{a}) , (j, \mathbf{a}) et (k, \mathbf{a}) sont compatibles entre elles les justifications ne sont pas bien fondées. Dans ces conditions nous ne sommes pas assurés d'avoir obtenu le domaine arc-consistant maximal. Notamment, sur cet exemple les valeurs (i, \mathbf{a}) , (j, \mathbf{a}) et (k, \mathbf{a}) ont été retirées alors qu'elles font partie d'une solution puisqu'elles sont compatibles entre elles.

3.3. L'algorithme

DnAC-4 utilise les structures de données suivantes :

- Une table de booléens D représente le domaine courant. On notera indifféremment $(i, \mathbf{a}) \in D$ ou $D[i, \mathbf{a}] = \text{VRAI}$.
- Chaque paire arc-valeur $[(i, j), \mathbf{a}]$ a un compteur $Cpt[(i, j), \mathbf{a}]$ qui indique le nombre de supports dans D de (i, \mathbf{a}) sur C_{ij} .

- A chaque paire arc-valeur $[(i, j), \mathbf{a}]$ est associé l'ensemble de supports $S[(i, j), \mathbf{a}] = \{b \in \text{dom}[j] / \mathbf{a} \text{ sur } i \text{ supporte } b \text{ sur } j\}$.
- $\text{Justif}[i, \mathbf{a}] = j$ ssi $(i, \mathbf{a}) \notin D$ et la contrainte justifiant le retrait de (i, \mathbf{a}) est C_{ij} (ce qui signifie que (i, \mathbf{a}) a été enlevée parce que $\text{Cpt}[(i, j), \mathbf{a}] = 0$).
 $\forall (i, \mathbf{a}) \in D, \text{Justif}[i, \mathbf{a}] = \text{nil}$.
- Les listes SL et RL sont utilisées pour propager le long des contraintes, respectivement les suppressions et les rajouts de valeurs.

Avant de pratiquer des restrictions et des relaxations, les structures D et Justif doivent être initialisées en appliquant la procédure **Init** sur $P_{(0)}$. Le CSP initial n'ayant pas de contraintes, cette procédure initialise les structures de données pour qu'elles correspondent au cas où le domaine courant est le domaine initial.

3.3.1. Les restrictions

L'ajout d'une contrainte C_{ij} s'effectue par l'appel de la procédure **Add** qui opère en deux étapes.

- La première, réalisée par **Init-Add**, s'occupe de la mise à jour des listes de supports et des compteurs ainsi que de l'initialisation du processus de suppression des valeurs non viables. La contrainte C_{ij} étant ajoutée à C , il faut en effet pour respecter les spécifications des structures, créer $\text{Cpt}[(i, j), \mathbf{a}]$, $\text{Cpt}[(j, i), \mathbf{b}]$, $S[(i, j), \mathbf{a}]$ et $S[(j, i), \mathbf{b}]$ pour tout $\mathbf{a} \in \text{dom}[i]$ et tout $\mathbf{b} \in \text{dom}[j]$. Remarquons que pour cela on doit effectuer des tests de consistance non seulement pour les valeurs du domaine courant mais également pour celles qui ont été retirées car elles pourraient être remises dans D lors d'une relaxation. Lors de cette mise à jour des structures, les paires arc-valeur dont le compteur est nul sont placées dans SL . Il s'agit des paires arc-valeur sans support sur C_{ij} , c'est à dire celles pour lesquelles C_{ij} constitue une cause directe de suppression.

- La deuxième phase est réalisée par **Propag-Suppress**. Pour chaque paire arc-valeur $[(k, m), \mathbf{a}]$ figurant dans SL avec $(k, \mathbf{a}) \in D$, on retire (k, \mathbf{a}) du domaine courant avec m pour justification et on décrémente les compteurs des valeurs qu'elle supporte. Les conséquences de chaque retrait sont propagées en plaçant dans SL les paires arc-valeur dont le compteur est devenu nul à la suite d'une telle décrémentation.

Remarque. La partie du test de la ligne 3 de **Propag-Suppress** qui contrôle la valeur du compteur n'est pas utile lors d'une restriction. En effet, lors de l'ajout d'une contrainte, le compteur est nul quand la paire arc-valeur est placée dans SL et le restera car il n'y a pas d'ajout de valeurs dans D . Ce test n'est donc utile que dans le cadre d'une relaxation comme nous allons le voir dans la prochaine section.

3.3.2. Les relaxations

La procédure **Relax** permet de retirer une contrainte C_{km} . Afin de conserver des justifications bien-fondées elle opère en deux phases.

- Durant la première phase on remet les valeurs dont la suppression est directement ou indirectement due à C_{km} . Pour cela, dans un premier temps **Init-**

Propag-Relax place dans la liste des valeurs à rajouter RL les valeurs dont le retrait est directement dû à C_{km} . Dans sa deuxième partie, elle propage les rajouts. Chaque valeur (i, a) prise dans RL est remise dans D et si cette valeur est un support pour une valeur absente (j, b) dont C_{ij} est la justification, alors (j, b) est placée dans RL . Durant cette phase on recherche également les éventuels compteurs nuls des valeurs remises et on place les paires arc-valeur correspondantes dans SL .

• Parmi les valeurs remises par Init-Propag-Relax il peut y avoir des valeurs non viables. En effet la justification n'était que la première des contraintes sur lesquelles la valeur n'avait pas de support et la valeur peut donc avoir conservé un compteur nul sur une autre contrainte. Propag-Suppress est dès lors exécutée pour éliminer les valeurs non viables remises à tort. Quand on traite une paire arc-valeur prise dans SL il est important de vérifier que le compteur est nul (ligne 3) car un support a pu être rajouté à D par Init-Propag-Relax après l'ajout de la paire arc-valeur dans SL .

<pre> procédure Init; pour tout $i \in X$ faire pour tout $a \in \text{dom}[i]$ faire $D[i, a] \leftarrow \text{VRAI}$; $\text{Justif}[i, a] \leftarrow \text{nil}$ procédure Add (in C_{ij} : contrainte); $C \leftarrow C \cup \{C_{ij}\}$; $SL \leftarrow \emptyset$; Init-Add((i, j), SL); Init-Add((j, i), SL); Propag-Suppress(SL); procédure Init-Add(in (i, j) : arc; in out SL : liste); 1 pour tout $b \in \text{dom}[j]$ faire $S[j, i, b] \leftarrow \emptyset$; 2 pour tout $a \in \text{dom}[i]$ faire 3 $\text{Total} \leftarrow 0$; 4 pour tout $b \in \text{dom}[j]$ faire 5 si $((i, a), (j, b)) \in C_{ij}$ alors 6 si $D[j, b]$ alors $\text{Total} \leftarrow \text{Total} + 1$; 7 Append($S[j, i, b]$, a); 8 $\text{Cpt}[i, j, a] \leftarrow \text{Total}$; 9 si $\text{Cpt}[i, j, a] = 0$ alors Append(SL, $[(i, j), a]$); procédure Propag-Suppress(in out SL : liste); 1 tant que $SL \neq \emptyset$ faire 2 prendre $[(i, m), a]$ dans SL; 3 si $D[i, a]$ et $\text{Cpt}[i, m, a] = 0$ alors 4 $\text{Justif}[i, a] \leftarrow m$; 5 $D[i, a] \leftarrow \text{FAUX}$; 6 pour tout $j' \in C_{ij}$ faire 7 pour tout $b \in S[(i, j), a]$ faire 8 $\text{Cpt}[j, i, b] \leftarrow \text{Cpt}[j, i, b] - 1$; 9 si $\text{Cpt}[j, i, b] = 0$ alors Append(SL, $[(j, i), b]$); </pre>	<pre> procédure Relax (in C_{km} : contrainte); $SL \leftarrow \emptyset$; Init-Propag-Relax(C_{km}, SL); Propag-Suppress(SL) procédure Init-Propag-Relax (in C_{km} : contrainte; in out SL : liste); { Partie 1 : Initialisation } 1 $RL \leftarrow \emptyset$; 2 pour tout $a \in \text{dom}[k]$ faire 3 si $\text{Justif}[k, a] = m$ alors 4 Append(RL, (k, a)); $\text{Justif}[k, a] \leftarrow \text{nil}$; 5 pour tout $b \in \text{dom}[m]$ faire 6 si $\text{Justif}[m, b] = k$ alors 7 Append(RL, (m, b)); $\text{Justif}[m, b] \leftarrow \text{nil}$; 8 Enlever C_{km} de C; { Partie 2 : Propagation } 9 tant que $RL \neq \emptyset$ faire 10 Prendre (i, a) dans RL; 11 $D[i, a] \leftarrow \text{VRAI}$; 12 pour tout $j' \in C_{ij}$ faire 13 pour tout $b \in S[(i, j), a]$ faire 14 $\text{Cpt}[j, i, b] \leftarrow \text{Cpt}[j, i, b] + 1$; 15 si $\text{Justif}[j, b] = i$ alors 16 Append(RL, (j, b)); $\text{Justif}[j, b] \leftarrow \text{nil}$; 17 si $\text{Cpt}[i, j, a] = 0$ alors Append(SL, $[(i, j), a]$); </pre>
---	---

Figure 1. L'algorithme DnAC-4

Les justifications sont bien-fondées car on a rajouté toutes les valeurs dont un support a été remis sur la contrainte justification. Ayant ensuite retiré les valeurs non viables on est assuré d'avoir obtenu le domaine arc-consistant maximal.

On peut trouver une preuve de DnAC-4 dans [BES 92].

3.4. Exemple

Afin d'illustrer les différentes méthodes présentées dans cet article, nous utiliserons l'exemple suivant, inspiré de celui de [BES 92] :

La firme PEUNAUT va sortir un nouveau modèle de voiture fabriquée dans toute l'Europe :

- Les portières et le capot sont faits à Lille où le constructeur dispose de peinture blanche, rose, rouge et noire.
- La carrosserie est faite à Hambourg où on a de la peinture blanche, rose, rouge et noire.
- Les pare-chocs faits à Palerme sont toujours blancs.
- La bâche du toit-ouvrant qui est faite à Madrid ne peut être que rouge.
- Les enjoliveurs sont faits à Athènes où l'on a de la peinture rose et de la peinture rouge.

Le concepteur de la voiture impose quelques-uns de ses désirs quant à l'agencement des couleurs pour cette voiture :

- La carrosserie doit être de la même couleur que le capot, lui même de la même couleur que les portières.
- Les enjoliveurs, les pare-chocs et le toit-ouvrant doivent être plus clairs que la carrosserie.

Un codage possible de ce problème peut être obtenu en représentant chaque composant de la voiture par une variable dont le domaine est constitué des couleurs disponibles. On dénotera par les lettres *a*, *b*, *c*, *d* les couleurs blanc, rose, rouge et noir respectivement. Les contraintes "plus clair que" se traduisent alors par "plus petit que" dans l'ordre lexicographique et sont représentées par des flèches dans la figure 2.

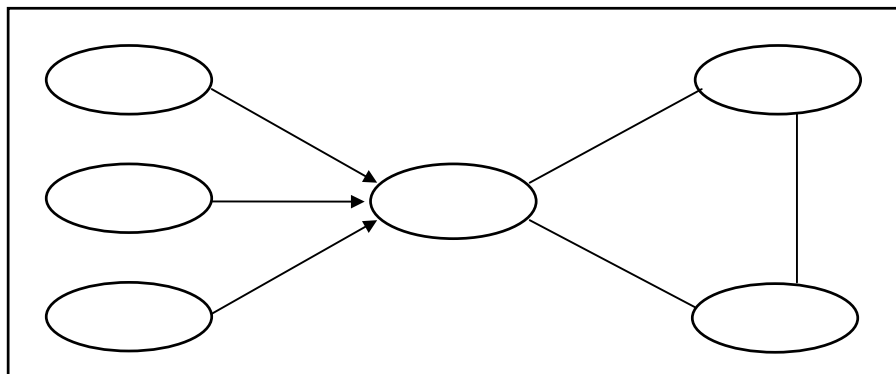


Figure 2. Une représentation du CSP pris en exemple

Supposons que le concepteur impose ses conditions dans l'ordre C_{23} , C_{34} , C_{24} , C_{12} , C_{62} , C_{52} et que les listes sont gérées de manière LIFO. On obtient alors l'exécution suivante :

Durant Add C_{23} , Add C_{34} et Add C_{24} aucune valeur n'est retirée car aucun des compteurs créés n'est nul.

Add C_{12} : Init-Add place $[(2, 1), a]$ dans SL .

$[(2, 1), a]$ prise dans $SL \rightarrow (2, a)$ est retirée de D avec C_{12} pour justification,
Ajout de $[(3, 2), a], [(4, 2), a]$ dans SL

$[(4, 2), a]$ prise dans $SL \rightarrow (4, a)$ est retirée de D avec C_{24} pour justification,
Ajout de $[(3, 4), a], [(2, 4), a]$ dans SL

$[(2, 4), a]$ prise dans $SL \rightarrow$ Rien car $(2, a) \notin D$

$[(3, 4), a]$ prise dans $SL \rightarrow (3, a)$ est retirée de D avec C_{34} pour justification,
Ajout de $[(2, 3), a], [(4, 3), a]$ dans SL

$[(4, 3), a]$ prise dans $SL \rightarrow$ Rien car $(4, a) \notin D$

$[(2, 3), a]$ prise dans $SL \rightarrow$ Rien car $(2, a) \notin D$

$[(3, 2), a]$ prise dans $SL \rightarrow$ Rien car $(3, a) \notin D$

De la même manière, l'ajout de C_{62} a pour conséquence le retrait de $(2, b)$ propagé aux retraits de $(4, b)$ et $(3, b)$. Enfin, la restriction correspondante à C_{52} supprime $(2, c)$ et ce retrait est propagé à celui de $(4, c)$, lequel entraîne la suppression de $(3, c)$.

Après l'ajout des six contraintes du problème, on obtient le DCSP de la figure 3.

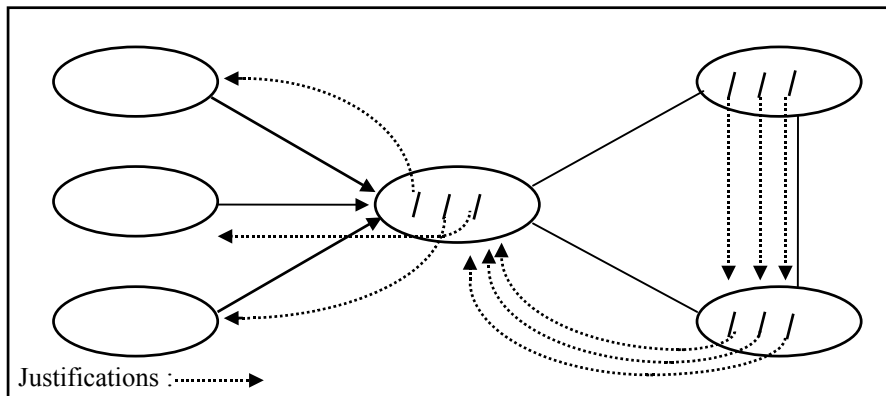


Figure 3. Le DCSP après l'ajout des six contraintes sur le CSP initial

Supposons qu'à présent le concepteur désire supprimer la contrainte C_{62} .

Relax C_{62} : La partie 1 de Init-Propag-Relax place $(2, b)$ dans RL .

$(2, b)$ prise dans $RL \rightarrow (2, b)$ est remise dans D , $(4, b)$ est placée dans RL
Ajout de $[(2, 3), b], [(2, 4), b], [(2, 5), b]$ dans SL

$(4, b)$ prise dans $RL \rightarrow (4, b)$ est remise dans D , $(3, b)$ est placée dans RL
Ajout de $[(4, 3), b]$ dans SL

$(3, b)$ prise dans $RL \rightarrow (3, b)$ est remise dans D , aucun ajout à RL ou SL

Au terme de **Init-Propag-Relax**, les valeurs $(2, b)$, $(4, b)$ et $(3, b)$ ont été remises mais des paires arc-valeur sans support ont été détectées et placées dans SL . **Propag-Suppress** va à présent utiliser la liste SL pour supprimer les éventuelles valeurs non viables.

- $[(4, 3), b]$ prise dans $SL \rightarrow$ Rien car $C_{pt}[(4, 3), b]=1$
- $[(2, 5), b]$ prise dans $SL \rightarrow (2, b)$ est retirée de D avec C_{25} pour justification
Ajout de $[(3, 2), b], [(4, 2), b]$ dans SL
- $[(4, 2), b]$ prise dans $SL \rightarrow (4, b)$ est retirée de D avec C_{24} pour justification
Ajout de $[(3, 4), b], [(2, 4), b]$ dans SL
- $[(2, 4), b]$ prise dans $SL \rightarrow$ Rien car $(2, b) \notin D$
- $[(3, 4), b]$ prise dans $SL \rightarrow (3, b)$ est retirée de D avec C_{34} pour justification
Ajout de $[(2, 3), b], [(4, 3), b]$ dans SL
- $[(4, 3), b]$ prise dans $SL \rightarrow$ Rien car $(4, b) \notin D$
- $[(2, 3), b]$ prise dans $SL \rightarrow$ Rien car $(2, b) \notin D$
- $[(3, 2), b]$ prise dans $SL \rightarrow$ Rien car $(3, b) \notin D$
- $[(2, 4), b]$ prise dans $SL \rightarrow$ Rien car $(2, b) \notin D$
- $[(2, 3), b]$ prise dans $SL \rightarrow$ Rien car $(2, b) \notin D$

La valeur $(2, b)$ qui avait été remise par **Init-Propag-Relax** du fait qu'un support avait été rajouté sur sa justification a due être retirée car elle ne possède pas de support sur C_{52} . L'exécution de **Relax** C_{62} n'a donc pas modifié les domaines mais la justification de $(2, b)$ est à présent C_{52} . On obtient le CSP de la figure 4.

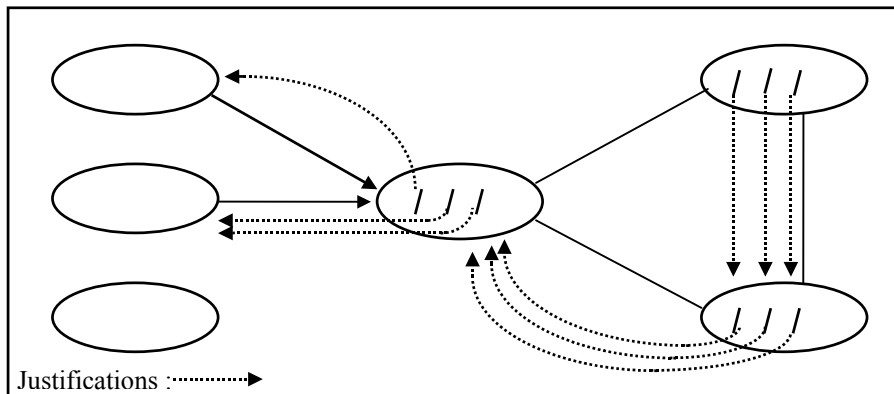


Figure 4. Le CSP de la figure 3 après le retrait de la contrainte C_{62}

3.5. Complexité

3.5.1. Complexité en espace

DnAC-4 utilise au maximum $2ed$ compteurs. Les tables D et $Justif$ ont une taille de nd . Les $2ed$ listes de supports peuvent chacune contenir au maximum d valeurs. L'espace qu'elles occupent est donc en $O(ed^2)$ et la complexité en espace de l'algorithme est en $O(ed^2+nd)$.

3.5.2. Complexité en temps

La complexité en temps des restrictions peut être déterminée en comptant le nombre d'opérations élémentaires effectuées. Ces dernières sont des incréments de compteurs dans **Init-Add** puis des décréments dans **Propag-Suppress**. Les compteurs ne pouvant pas être négatifs, la complexité est bornée par la somme de tous ces compteurs, laquelle ne peut excéder $2ed^2$. La complexité est donc en $O(ed^2)$ qui est la complexité optimale déjà atteinte par AC-4.

La complexité des relaxations est également déterminée par le nombre d'incréments et de décréments de compteurs et pour la même raison, elle est en $O(ed^2)$.

4. Un nouvel algorithme : DnAC-6

4.1. Introduction

L'algorithme DnAC-4 que nous venons de voir est performant sur les DCSP car il est incrémental à la fois pour les restrictions et les relaxations. Toutefois il conserve certains inconvénients de AC-4. Sa complexité en espace est mauvaise, à cause notamment de ses listes de valeurs supportées. Si ces dernières lui permettent de faire relativement peu de tests de consistance, elles sont très imposantes, particulièrement quand les contraintes sont molles. De plus, pour maintenir les propriétés intéressantes de cette structure de données il faut effectuer un lourd travail lors des restrictions. On peut remarquer que lors de l'ajout d'une contrainte C_{ij} la procédure **Init-Add**((i, j), SL) recherche **tous** les supports sur C_{ij} pour toutes les valeurs de i, y compris celles qui ne figurent plus dans le domaine courant.

DnAC-6 tire son nom du fait qu'il allie les idées de DnAC-4 à celles de AC-6 [BES 94], un algorithme d'arc-consistance sur les CSP statiques plus efficace et surtout moins coûteux en espace que AC-4.

4.2. Les idées

- L'idée de AC-6, c'est à dire rechercher le plus petit support selon un ordre pris sur les domaines afin de ne pas avoir à reconsidérer les valeurs inférieures si on est amené à chercher un nouveau support, n'est pas applicable formellement par un algorithme incrémental sur les DCSP. En effet, des valeurs peuvent non seulement être retirées du domaine mais aussi y être remises lors d'une relaxation. L'adaptation de cette idée aux CSP dynamiques est réalisée par DnAC-6 au moyen d'un domaine constitué de listes. A chaque variable i sont associées les listes *Présentes*[i] et *Absentes*[i] qui contiennent respectivement les valeurs présentes et absentes. De plus, chaque ajout d'une valeur s'effectue en fin de liste. Ainsi, la recherche d'un support se fera selon l'ordre des éléments dans les listes *Présentes* et si un nouveau support doit être trouvé, on sait qu'il ne peut figurer que parmi les successeurs du support courant.

- Pour chaque valeur du domaine courant, DnAC-6 tente de justifier qu'elle figure dans la fermeture arc-consistante en vérifiant qu'elle possède un support sur chaque contrainte portant sur sa variable. Ces supports sont mentionnés explicitement au moyen de deux structures : *SupportéPar* et *Supporte*. Si on trouve un support b pour une valeur (i, a) sur C_{ij} alors on affecte *SupportéPar*[(i, j), a] ← b et on ajoute a à *Supporte*[(j, i), b]. *Supporte* a la même fonction que les listes de supports de AC-6 : si une valeur est supprimée, cette structure permet de déterminer les valeurs dont la présence doit être remise en cause. *SupportéPar* a une double utilité. D'abord on peut la voir comme une justification de présence : Après l'exécution de *Add* ou de *Relax*, pour chaque valeur (i, a) figurant dans le domaine courant et pour toute variable j adjacente à i on a *SupportéPar*[(i, j), a] qui est un support de (i, a) sur C_{ij} . De plus cette structure permet de supprimer des listes *Supporte* les valeurs absentes. Ainsi *Supporte*[(i, j), a] ne contient que les valeurs du domaine courant de j pour lesquelles (i, a) est le support courant.

- Un système de justification pour les valeurs absentes similaire à celui de DnAC-4 permet à l'algorithme d'être incrémental. On retrouve également une liste *RL* pour la propagation des rajouts de valeurs. Par contre, pour la propagation des suppressions on doit obligatoirement avoir une gestion LIFO. C'est pourquoi on utilise une pile notée *SL*.

- Une structure nommée *DernièreTestée* est utilisée afin de ne pas refaire de tests inutilement. Plus précisément, lors de la relaxation de C_{km} on procède en deux phases : On remet les valeurs dont le retrait est dû directement ou indirectement à C_{km} puis on supprime les valeurs non viables remises à tort. Si en remettant une valeur (i, a) contrainte par C_{ij} on ne trouve pas de support lors du parcours de valeurs présentes sur j , on le stipule à la deuxième phase en plaçant [(i, j), a , VRAI] dans *SL* et en affectant à *DernièreTestée*[(i, j), a] la dernière valeur de *Présentes*[j]. Ainsi, si un support de (i, a) sur C_{ij} est remis ultérieurement il sera un successeur de *DernièreTestée*[(i, j), a] dans *Présentes*[j] et donc la seconde phase n'a pas à rechercher parmi les prédécesseurs.

4.3. L'algorithme

On désignera par $dom[i]$ le domaine de la variable i dans $P_{(0)}$.

Les structures de données utilisées et leurs propriétés essentielles sont les suivantes.

- D représente les domaines courants. Chaque domaine $D[i]$ est constitué d'une table, afin de déterminer la présence ou l'absence d'une valeur en $O(1)$, mais également de deux listes $Présentes[i]$ et $Absentes[i]$ contenant respectivement les valeurs présentes et absentes. Ces listes sont exploitées au moyen des fonctions suivantes qui s'exécutent toutes en temps constant :

◊ $Suivant(i, a)$ fournit la valeur qui suit (i, a) si (i, a) n'est pas la dernière valeur de sa liste, et renvoie nil sinon.

◊ $PremièrePrésente(i)$ et $DernièrePrésente(i)$ renvoient respectivement la première et la dernière valeur de la liste $Présentes[i]$ si cette liste n'est pas vide et renvoient nil sinon.

◊ On notera $Mettre((i, a), D)$ l'opération qui consiste à rendre présente la valeur (i, a) absente du domaine courant de i en le stipulant dans la table mais aussi en enlevant a de $Absentes[i]$ pour l'ajouter à la fin de $Présentes[i]$. De manière similaire $Enlever((i, a), D)$ enlève la valeur a de $Présentes[i]$ pour la placer à la fin de $Absentes[i]$ en plus de mentionner l'absence de (i, a) dans la table.

- Pour chaque valeur (i, a) de D et pour toute variable j voisine de i dans le graphe associé, $SupportéPar[(i, j), a]$ est un support dans D pour (i, a) sur C_{ij} (pour les autres paires arc-valeur $[(i, j), a]$, $SupportéPar[(i, j), a]$ vaut nil).

• A chaque paire arc-valeur $[(i, j), a]$ est associée une liste de supports $Supporte[(i, j), a]$ telle que $b \in Supporte[(i, j), a] \Leftrightarrow SupportéPar[(j, i), b] = a$.

- $Justif[i, a] = j$ ssi $(i, a) \notin D$ et la contrainte justifiant le retrait de (i, a) est C_{ij} .

- Une liste RL est utilisée pour la propagation des rajouts de valeurs.

- Pour propager les suppressions on utilise une pile de triplets arc-valeur-booléen notée SL . Un élément $[(i, j), a, SupportPossible]$ figurant dans SL signifie que lorsqu'il fut empilé, (i, a) n'avait pas de support sur C_{ij} dans D . Le booléen **SupportPossible** spécifie si un support a pu être remis après l'ajout du triplet dans SL . Cette liste est gérée au moyen des opérations suivantes :

◊ $Empiler(SL, [(i, j), a, SupportPossible])$ empile le triplet sur SL .

◊ $Dépiler(SL)$ renvoie le sommet après l'avoir retiré de SL .

Si on propageait les retraits au moyen d'une liste FIFO on pourrait supprimer une valeur (j, b) telle qu'il existe (i, a) avec $DernièreTestée[(i, j), a] = b$ ce qui empêcherait de poursuivre la recherche d'un éventuel support de (i, a) sur C_{ij} car b ne serait plus dans $Présentes[j]$. Dans [DEB 94] on trouve une preuve qu'un tel cas ne peut pas se produire quand on utilise une pile.

- $DernièreTestée[(i, j), a] = b$ si $[(i, j), a, VRAI]$ a été placé dans SL et b est la dernière valeur testée par **Init-Propag-Relax** lors de la recherche d'un support de (i, a) sur C_{ij} .

Avant de procéder à des restrictions et à des relaxations les structures doivent être initialisées en appliquant la procédure **Init** sur $P_{(0)}$.

La procédure **ProchainSupport**($i, j, a, b, SupportVide$) est utilisée pour trouver un support de (i, a) sur C_{ij} qui soit b ou un successeur de b dans la liste *Présentes*[jj]. Elle est appelée avec $a \in D[i]$ et $b \in D[j] \cup \{\text{nil}\}$. *SupportVide* est **VRAI** ssi $b = \text{nil}$ ou s'il n'existe pas de tel support pour (i, a) .

4.3.1. Les restrictions

La procédure **Add** ajoute une contrainte en deux phases.

- Dans un premier temps **Init-Add** recherche un support sur la nouvelle contrainte C_{ij} pour toute valeur présente dans le domaine courant de i et de j . Si elle en trouve un, elle complète les structures *SupportéPar* et *Supporte* et dans le cas contraire la paire arc-valeur sans support sur C_{ij} est empilée sur *SL*.

- Grâce à *SL*, **Propag-Suppress** peut ensuite retirer de D toutes les valeurs sans support sur une contrainte et en spécifier la justification. La propagation des retraits s'effectue par le biais de *SL* et des listes *Supporte*[(i, j), a] qui permettent de connaître pour chaque valeur (i, a) supprimée l'ensemble des valeurs dont la présence est remise en cause par le retrait de (i, a) . Le troisième élément des triplets ajoutés à *SL* est la constante **FAUX**. On spécifie par cette valeur qu'aucun support n'a pu être remis dans le domaine depuis que le triplet figure dans *SL*.

4.3.2. Les relaxations

Le retrait d'une contrainte est effectué au moyen de la procédure **Relax** qui procède en deux étapes pour conserver des justifications bien fondées :

- Dans la première phase **Init-Propag-Relax** commence par ajouter dans la liste *RL* les valeurs pour lesquelles C_{km} justifie le retrait. Puis les valeurs de *RL* sont remises dans D et on propage ces rajouts le long des justifications. Pour cela on remet dans le domaine courant les valeurs dont un support sur la contrainte justification est de nouveau présent. De plus, pour vérifier la viabilité de chaque valeur (i, a) remise on recherche un support sur chaque contrainte adjacente C_{ij} . Si cette recherche est infructueuse on empile [(i, j), a , **VRAI**] sur *SL*.

- Dans la seconde phase, **Propag-Suppress** est appelée pour supprimer les valeurs qui ont été ajoutées à tort dans le domaine courant. Grâce à *SL* cette procédure va supprimer les valeurs non viables. Contrairement au cas des restrictions, des triplets dont le troisième élément est **VRAI** peuvent figurer dans *SL*. Un triplet [(i, j), a , **VRAI**] figurant dans *SL* ne signifie pas que (i, a) doit obligatoirement être supprimée de D puisque **Init-Propag-Relax** peut avoir ajouté un support sur C_{ij} par la suite. C'est pourquoi la procédure **RechercherSupport** modifie la valeur *SupportéPar* de la paire arc-valeur si un support a été ajouté au domaine.

Une preuve de cet algorithme figure dans [DEB 94].

```

procédure Init;
  pour tout  $i \in X$  faire
    pour tout  $a \in \text{dom}[i]$  faire
      Mettre(  $(i, a), D$  );
      Justif[i, a] ← nil

procédure ProchainSupport( in  $(i, j)$  : arc; in  $a$  : valeur;
  in out  $b$  : valeur; out SupportVide : booléen)
1  tant que  $b \neq \text{nil}$  faire
2    si  $((i, a), (j, b)) \in C_{ij}$  alors
3      SupportVide ← FAUX;
4      Retour;
5       $b \leftarrow \text{Suivant}(j, b)$ ;
6  SupportVide ← VRAI;

procédure Add ( in  $C_{ij}$  : contrainte )
   $SL \leftarrow \emptyset$ ;
   $C \leftarrow C \cup \{ C_{ij} \}$ ;
  Init-Add(  $(i, j), SL$  ); Init-Add(  $(j, i), SL$  );
  Propag-Suppress(  $SL$  );

procédure Init-Add( in  $(i, j)$  : arc; in out  $SL$  : pile );
1  pour tout  $b \in \text{Présentes}[j]$  faire Supporte[( $j, i$ ),  $b$ ] ←  $\emptyset$ ;
2  pour tout  $a \in \text{Présentes}[i]$  faire
3     $b \leftarrow \text{PremièrePrésente}(j)$ ;
4    ProchainSupport(  $i, j, a, b, \text{SupportVide}$  );
5    si (SupportVide) alors
6      Empiler(  $SL, [(i, j), a, \text{FAUX}]$  );
7      SupportéPar[( $i, j$ ),  $a$ ] ← nil;
8    sinon
9      Append(Supporte[( $j, i$ ),  $b$ ],  $a$  );
10     SupportéPar[( $i, j$ ),  $a$ ] ←  $b$ ;

procédure Propag-Suppress( in  $SL$  : pile );
1  tant que (  $SL \neq \emptyset$  )
2    [( $i, j$ ),  $a, \text{SupportPossible}$ ] ← Dépiler(  $SL$  );
3    si (  $(i, a) \in D$  ) alors
4      si (SupportPossible = VRAI) alors
5        RechercherSupport(  $(i, j), a$  );
6      si (SupportéPar[( $i, j$ ),  $a$ ] = nil) alors
7         $p \leftarrow \text{Suivant}(i, a)$ ;
8        Enlever(  $(i, a), D$  );
9        Justif[i, a] ←  $j$ ;
10     pour tout  $k / C_{ik} \in C$ 
11       si ( SupportéPar[( $i, k$ ),  $a$ ]  $\neq$  nil ) alors
12          $c \leftarrow \text{SupportéPar}[(i, k), a]$ ;
13         Supprimer( SupportéPar[( $k, i$ ),  $c$ ],  $a$  );
14         SupportéPar[( $i, k$ ),  $a$ ] ← nil;
15         DernièreTestée[( $i, k$ ),  $a$ ] ← nil;
16       pour tout  $b \in \text{Supporte}[(i, k), a]$  faire
17         Supprimer( Supporte[( $i, k$ ),  $a$ ],  $b$  );
18          $c \leftarrow p$ ;
19         ProchainSupport(  $k, i, b, c, \text{SupportVide}$ 
20 );
21     si ( SupportVide ) alors
22       Empiler(  $SL, [(k, i), b, \text{FAUX}]$  );
23       SupportéPar[( $k, i$ ),  $b$ ] ← nil;
24     sinon
25       Append( Supporte[( $i, k$ ),  $c$ ],  $b$  );
26       SupportéPar[( $k, i$ ),  $b$ ] ←  $c$ ;

procédure Relax( in  $C_{km}$  : contrainte;
   $SL \leftarrow \emptyset$ ;
  Init-Propag-Relax(  $C_{km}, SL$  );
  Propag-Suppress(  $SL$  );

procédure RechercherSupport( in  $(i, j)$  : arc ;
  in  $a$  : valeur );
1  si (DernièreTestée[( $i, j$ ),  $a$ ]  $\neq$  nil) alors
2     $b = \text{DernièreTestée}[(i, j), a]$ ;
3    DernièreTestée[( $i, j$ ),  $a$ ] ← nil;
4     $b = \text{Suivant}(j, b)$ ;
5    ProchainSupport(  $i, j, a, b, \text{SupportVide}$  );
6    si (non (SupportVide)) alors
7      Append( Supporte[( $j, i$ ),  $b$ ],  $a$  );
8      SupportéPar[( $i, j$ ),  $a$ ] ←  $b$ ;
9    sinon
10      $b \leftarrow \text{PremièrePrésente}(j)$ ;
11     ProchainSupport(  $i, j, a, b, \text{SupportVide}$  );
12     si (non (SupportVide)) alors
13       Append( Supporte[( $j, i$ ),  $b$ ],  $a$  );
14       SupportéPar[( $i, j$ ),  $a$ ] ←  $b$ ;

procédure Init-Propag-Relax( in  $C_{km}$  : contrainte;
  in out  $SL$  : pile);
  {Partie 1 : Initialisation }
1   $RL \leftarrow \emptyset$ ;
2  pour tout  $a \in \text{Absentes}[k]$  faire
3    si (Justif[ $k, a$ ] =  $m$ ) alors
4      Append( $RL, (k, a)$ ); Justif[ $k, a$ ] ← nil;
5  pour tout  $b \in \text{Absentes}[m]$  faire
6    si (Justif[ $m, b$ ] =  $k$ ) alors
7      Append( $RL, (m, b)$ ); Justif[ $m, b$ ] ← nil;
8  Enlever  $C_{km}$  de  $C$ ;
9  Détruire les structures Supporte, SupportéPar et
  DernièreTestée entre  $k$  et  $m$ 

  {Partie 2 : Propagation }
10 tant que ( $RL \neq \emptyset$ ) faire
11  prendre ( $i, a$ ) dans  $RL$ ;
12  Mettre(  $(i, a), D$  );
13  pour tout  $j / C_{ij} \in C$  faire
14     $b \leftarrow \text{PremièrePrésente}(j)$ ;
15    ProchainSupport(  $i, j, a, b, \text{SupportVide}$  );
16    si ( SupportVide ) alors
17      DernièreTestée[( $i, j$ ),  $a$ ] ← DernièrePré-
18        sente( $j$ );
19    Empiler(  $SL, [(i, j), a, \text{VRAI}]$  );
20    sinon
21      Append( Supporte[( $j, i$ ),  $b$ ],  $a$  );
22      SupportéPar[( $i, j$ ),  $a$ ] ←  $b$ ;
23    pour tout  $c \in \text{Absentes}[i]$  faire
24      si (Justif[ $j, c$ ] =  $i$ ) alors
25        si (  $(i, a), (j, c) \in C_{ij}$  ) alors
26          Append(  $RL, (j, c)$  );
          Justif[j, c] ← nil;

```

Figure 5. L'algorithme DnAC-6

4.4. Exemple

Regardons le comportement de DnAC-6 sur l'exemple vu au paragraphe 3.4. Durant Add C_{23} , Add C_{34} et Add C_{24} aucune valeur n'est retirée car chaque valeur a un support sur chaque contrainte adjacente.

Add C_{12} Init-Add place $[(2, 1), a, \text{FAUX}]$ dans SL .

$[(2, 1), a, \text{FAUX}]$ pris dans $SL \rightarrow (2, a)$ est retirée de D avec C_{12} pour justification,

Ajout de $[(3, 2), a, \text{FAUX}]$, $[(4, 2), a, \text{FAUX}]$ dans SL

$[(4, 2), a, \text{FAUX}]$ pris dans $SL \rightarrow (4, a)$ est retirée de D avec C_{24} pour justification,

Ajout de $[(3, 4), a, \text{FAUX}]$ dans SL

$[(3, 4), a, \text{FAUX}]$ pris dans $SL \rightarrow (3, a)$ est retirée de D avec C_{34} pour justification.

$[(3, 2), a, \text{FAUX}]$ pris dans $SL \rightarrow$ Rien car $(3, a) \notin D$

De la même manière, l'ajout de C_{62} a pour conséquence le retrait de $(2, b)$ propagé aux retraits de $(4, b)$ et $(3, b)$. Enfin, la restriction correspondante à C_{52} supprime $(2, c)$ et ce retrait est propagé à celui de $(4, c)$, lequel entraîne la suppression de $(3, c)$.

Après l'ajout des six contraintes du problème, on obtient le DCSP de la figure 6.

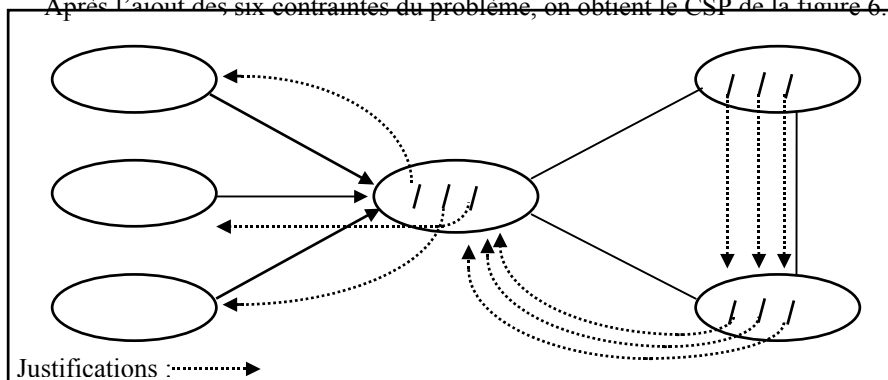


Figure 6. Le DCSP après l'ajout des six contraintes sur le CSP initial

Retirons à présent la contrainte C_{62} .

Relax C_{62} La partie 1 de Init-Propag-Relax place $(2, b)$ dans RL .

$(2, b)$ prise dans $RL \rightarrow (2, b)$ est remise dans D , $(4, b)$ est placée dans RL

Ajout de $[(2, 3), b, \text{VRAI}]$, $[(2, 4), b, \text{VRAI}]$ et $[(2, 5), b, \text{VRAI}]$ dans SL

$(4, b)$ prise dans $RL \rightarrow (4, b)$ est remise dans D , $(3, b)$ est placée dans RL

Ajout de $[(4, 3), b, \text{VRAI}]$ dans SL

$(3, b)$ prise dans $RL \rightarrow (3, b)$ est remise dans D , aucun ajout dans RL ou SL

On a donc remis les valeurs $(2, b)$, $(4, b)$ et $(3, b)$ mais il reste à exécuter Propag-Suppress pour retirer les éventuelles valeurs non viables.

$[(4, 3), b, \text{VRAI}]$ pris dans $SL \rightarrow$ Rien car $(4, b)$ a $(3, b)$ pour support sur C_{43} dans D

$[(2, 5), b, \text{VRAI}]$ pris dans $SL \rightarrow (2, b)$ supprimée avec C_{25} pour justification
Ajout de $[(3, 2), b, \text{FAUX}], [(4, 2), b, \text{FAUX}]$ dans SL

$[(4, 2), b, \text{FAUX}]$ pris dans $SL \rightarrow (4, b)$ supprimée avec C_{24} pour justification
Ajout de $[(3, 4), b, \text{FAUX}]$ dans SL

$[(3, 4), b, \text{FAUX}]$ pris dans $SL \rightarrow (3, b)$ supprimée avec C_{34} pour justification

$[(3, 2), b, \text{FAUX}]$ pris dans $SL \rightarrow$ Rien car $(3, b) \notin D$

$[(2, 4), b, \text{VRAI}]$ pris dans $SL \rightarrow$ Rien car $(2, b) \notin D$

$[(2, 3), b, \text{VRAI}]$ pris dans $SL \rightarrow$ Rien car $(2, b) \notin D$

Bien évidemment comme avec DnAC-4 la valeur $(2, b)$ a été retirée. Bien qu'ayant des principes de fonctionnement très différents, DnAC-6 qui tente de justifier l'appartenance à la fermeture arc-consistante en recherchant des supports dans les listes *Présentes* et DnAC-4 qui propage des incréments et de décréments de compteurs, présentent des suites d'ajouts et de retraites de valeurs presque similaires lors de Relax C_{62} .

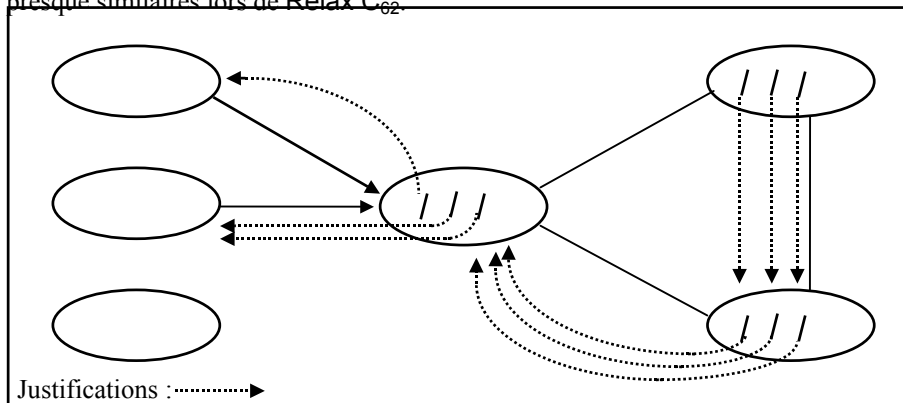


Figure 7. Le CSP de la figure 6 après le retrait de la contrainte C_{62}

4.5. Complexité

4.5.1. Complexité en espace

Les structures D et $Justif$ ont une taille de nd . Le nombre de valeurs *SupportéPar* utilisées est tout comme pour les valeurs *DernièreTestée* d'au plus $2ed$. Comme on a $\mathbf{a} \in \text{Supporte}[j, i], \mathbf{b}] \Leftrightarrow \text{SupportéPar}[i, j], \mathbf{a}] = \mathbf{b}$ on a au plus $2ed$ éléments dans les listes *Supporte*. La complexité en espace de DnAC-6 est donc en $O(ed + nd)$.

4.5.2. Complexité en temps.

Remarquons d'abord que le retrait d'un élément d'une liste doublement chaînée *Supporte* s'effectue en $O(1)$ car on a la propriété $b \in \text{Supporte}[(i, j), \mathbf{a}] \Leftrightarrow \text{SupportéPar}[(j, i), \mathbf{b}] = \mathbf{a}$ et dans une implémentation de l'algorithme la structure *SupportéPar* peut donc être utilisée pour conserver un pointeur sur la cellule concernée de la liste.

Les instructions élémentaires que nous pouvons utiliser pour le calcul de la complexité sont les tests de consistance réalisés par *ProchainSupport* (ligne 2). Ces tests forment la partie la plus interne des boucles majeures (la ligne 4 de *Init-Add*, la ligne 19 de *Propag-Suppress*, les lignes 4 et 10 de *RechercherSupport* et la ligne 15 de *Init-Propag-Relax*). Quand on appelle *ProchainSupport*($i, j, \mathbf{a}, \mathbf{b}, \text{SupportVide}$) c'est avec pour valeur \mathbf{b} la valeur qui suit *SupportéPar*[(i, j), \mathbf{a}] dans *Présentes*[j] si cette dernière ne vaut pas nil. Ainsi, lors d'une même application d'une procédure de DnAC-6 on ne refait pas deux fois le même test de consistance.

De plus, *ProchainSupport*($i, j, \mathbf{a}, \mathbf{b}, \text{SupportVide}$) n'est appelée que si (i, \mathbf{a}) n'a pas de support courant sur C_{ij} (Dans *RechercherSupport* : du fait que [(i, j), \mathbf{a}, VRAI] est retiré de *SL* et donc *SupportéPar*[(i, j), \mathbf{a}] = nil. Dans *Propag-Suppress* : voir les lignes 8 et 16. Dans *Init-Propag-Relax* : car (i, \mathbf{a}) vient d'être remise dans *D* et on a encore *SupportéPar*[(i, j), \mathbf{a}] = nil). Quand *ProchainSupport* a atteint la fin du domaine, *ProchainSupport* ne sera appelée avec les mêmes valeurs i, j et \mathbf{a} que si des valeurs sont rajoutées à *D*[j]. Ainsi, *SupportéPar*[(i, j), \mathbf{a}] changera au plus d fois de valeurs et d tests de consistance au plus auront été effectués avec la paire arc-valeur [(i, j), \mathbf{a}]. Par conséquent, la complexité est en $O(ed^2)$ qu'il s'agisse d'une restriction ou d'une relaxation.

5. Une approche sans justifications des retraits : AC | DC

5.1. Introduction

Parallèlement au développement de DnAC-6, B. Neveu et P. Berlandier créèrent un algorithme qui, contrairement aux deux approches que nous venons de voir, ne s'appuie pas sur un système de justification des retraits. L'objectif était non seulement de réduire la complexité en espace mais également d'avoir un algorithme le plus simple possible.

5.2. Les idées

- L'idée principale est de conserver une représentation dénuée d'imposantes structures de données globales. Notamment, contrairement aux deux approches précédentes, il n'est pas nécessaire de stocker des informations durant les

restrictions pour faciliter d'éventuelles relaxations. Par conséquent, n'importe quelle procédure réalisant l'arc-consistance sans utiliser de structure de données globales peut être utilisée pour pratiquer une restriction. Pour utiliser AC-4 ou AC-6 il faudrait lors des relaxations effectuer un travail supplémentaire afin de maintenir les propriétés de leurs structures de données. On perdrait donc à la fois en simplicité, en complexité en espace et en efficacité lors des relaxations. Comme dans [N&B 94] la version de AC|DC que nous présentons s'appuie sur AC-3 [MAC 77].

- Les relaxations s'opèrent selon un protocole sensiblement identique à celui déjà vu : On remplace dans le domaine courant un sur-ensemble des valeurs dont le retrait est dû à la contrainte supprimée, puis on retire les valeurs qui demeurent sans support sur une contrainte. Le retrait des valeurs non viables est réalisé par une procédure de filtrage dérivée de AC-3. L'essentiel de AC|DC est donc la détermination du sur-ensemble des valeurs à remettre lors de la relaxation d'une contrainte C_{km} . Pour cela on commence par effectuer une surestimation des valeurs dont le retrait est directement dû à C_{km} en plaçant dans un tableau d'ensembles nommé *Propagable* les valeurs absentes de k et m qui n'ont pas de support sur C_{km} dans le domaine courant. En effet, une valeur absente (k, a) ou (m, a) qui a un support dans D sur C_{km} a forcément été retirée à cause d'une autre contrainte. On propage ensuite chaque ajout d'une valeur (i, a) dans *Propagable* en recherchant parmi les valeurs absentes des variables adjacentes celles pour lesquelles (i, a) constitue un nouveau support. Pour ne pas propager plusieurs fois un même ensemble, quand *Propagable*[i] a été propagé à toutes les variables voisines de i , on le transfère dans un autre tableau d'ensembles nommé *Restorable*.

On montre facilement que l'ensemble des valeurs réintroduites par DnAC-4 ou DnAC-6 est toujours inclus dans l'ensemble des valeurs remises par AC|DC.

5.3. L'algorithme

Les structures de données utilisées sont particulièrement simples car mises à part celles utilisées par AC-3, elles se limitent aux deux tableaux d'ensembles *Propagable* et *Restorable*. *Propagable* sert, comme son nom l'indique, à la propagation de rajouts de valeurs alors que *Restorable* est utilisé pour contenir le sur-ensemble des valeurs à remettre tout en assurant la terminaison du processus de propagation.

5.3.1. Les restrictions

Comme nous l'avons précisé, les restrictions sont effectuées par AC-3. Le lecteur intéressé par le détail de ce classique de l'arc-consistance pourra consulter [MAC 77].

5.3.2. Les relaxations

Pour initialiser les rajouts de valeurs, la procédure **Propose** détermine un sur-ensemble des valeurs de k et m dont le retrait est dû à C_{km} . Pour cela elle sélectionne les valeurs de $dom[k] \setminus D[k]$ et de $dom[m] \setminus D[m]$ qui n'ont pas de support sur C_{km} dans D . Ces valeurs sont placées dans la table *Propagable* afin de servir de base à l'étape suivante. Dans la seconde phase, tant qu'il existe un ensemble *Propagate*[i] non vide, la procédure **Propagate** le propage à chaque variable j voisine de i en créant l'ensemble S_j des valeurs de $dom[j] \setminus D[j]$ ne figurant ni dans *Propagable* ni dans *Restorable* et qui possèdent un support dans *Propagate*[i]. Les ensembles S_j créés seront placés dans *Propagable* pour être à leur tour propagés. Pour ne propager qu'une seule fois chaque ensemble, quand *Propagate*[i] a été propagé il est retiré de *Propagable* puis placé dans *Restorable*.

Au terme de cette propagation, la table *Restorable* contient les valeurs à remettre dans le domaine mais également des valeurs non viables par rapport au problème relaxé ayant $D \cup Restorable$ pour domaine. La troisième phase réalisée par la procédure **Filter** consiste à retirer de *Restorable* les valeurs qui ne possèdent pas de support dans $D \cup Restorable$ sur chaque contrainte adjacente. Elle peut donc être réalisée à partir d'une procédure d'arc-consistance classique (comme AC-3) modifiée pour ne filtrer que les valeurs figurant dans *Restorable* et pour rechercher les supports à la fois dans le domaine courant et dans *Restorable*.

Il ne reste plus ensuite qu'à remettre dans D les valeurs de *Restorable*.

Dans l'algorithme qui suit, on note V l'ensemble des variables contraintes.

```

procédure Retract-constrained( in  $C_{km}$  : contrainte );
1  initialiser Propagable à  $\emptyset$ ;
2  supprimer  $C_{km}$  de  $C$ ;
3  Propose( $k, m, Propagable$ );
4  Propose( $m, k, Propagable$ );
5  initialiser Restorable à  $\emptyset$ ;
6  Propagate( $C_{km}, Propagable, Restorable$ );
7  Filter(Restorable);
8  pour tout  $i \in V$  faire
9    ajouter Restorable[ $i$ ] à  $D$ .

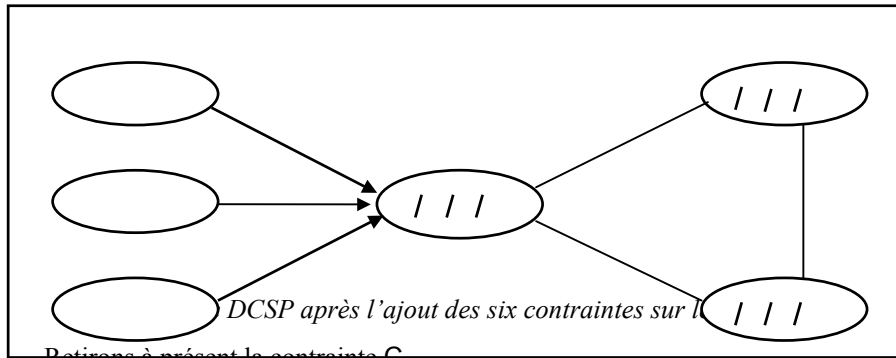
procédure Propagate( in  $C_{km}$  : contrainte ;
  in out Propagable, Restorable: tableau d'ensembles);
1   $L \leftarrow \{k, m\}$ ;
2  tant que  $L \neq \emptyset$  faire
3    prendre  $i$  dans  $L$ ;
4    pour tout  $j / C_{ij} \in C$  faire
5       $S \leftarrow \emptyset$ ;
6      pour tout  $b$  appartenant à
           $dom[j] \setminus (D[j] \cup Restorable[j] \cup Propagable[j])$ 
7        pour tout  $a \in Propagable[i]$  faire
8          si  $((i, a), (j, b)) \in C_{ij}$  alors
9            ajouter  $b$  à  $S$ ;
10         break;
11       si  $S \neq \emptyset$  faire
12         ajouter  $j$  à  $L$ ;
13         joindre  $S$  à Propagable[ $j$ ];
14       joindre Propagable[ $i$ ] à Restorable[ $i$ ];
15       propagable[ $i$ ]  $\leftarrow \emptyset$ ;

```

Figure 8. L'algorithme AC/DC

5.4. Exemple

Regardons le déroulement de l'algorithme AC|DC sur l'exemple du paragraphe 3.4. L'ajout des différentes contraintes n'est que de peu d'intérêt car il est réalisé par le célèbre AC-3. Seul compte ici le fait qu'il aboutit bien évidemment au CSP de la figure 9.



Retirons à présent la contrainte C_{62} .

Relax C_{62}

Propose place a et b dans $Propagable[2]$ puis on initialise L avec $\{2, 6\}$.

2 pris dans $L \rightarrow a$ et b sont placées dans $Propagable[3]$, 3 est ajoutée à L .

a et b sont placées dans $Propagable[4]$, 4 est ajoutée à L .

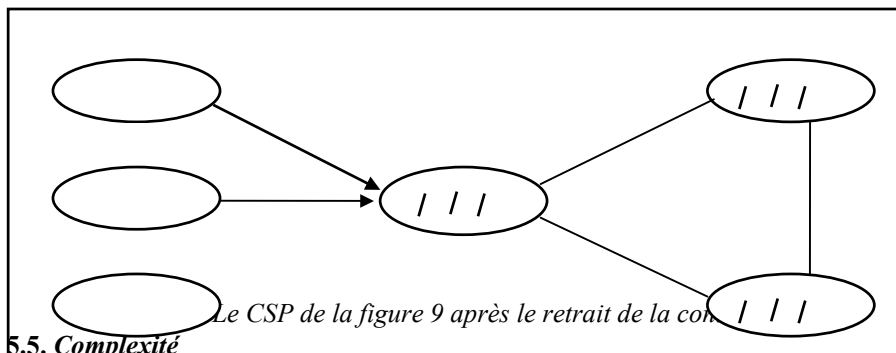
transfert des valeurs de $Propagable[2]$ dans $Restorable[2]$

4 pris dans $L \rightarrow$ transfert des valeurs de $Propagable[4]$ dans $Restorable[4]$

3 pris dans $L \rightarrow$ transfert des valeurs de $Propagable[3]$ dans $Restorable[3]$

6 pris dans $L \rightarrow$ rien car $Propagable[6]$ est vide

N'utilisant pas de système de justification des retraits, l'estimation des valeurs à remettre réalisée par AC|DC est plus grossière que celle des deux approches précédentes. Les six valeurs placées dans $Restorable$ seront ensuite retirées par Filter et le CSP résultant correspond à celui de la figure 10.



5.5. Complexité

5.5.1. Complexité en espace

Les tableaux *Propagable* et *Restorable* ont une taille en $O(nd)$. AC3 requérant un espace en $O(e+nd)$, la complexité de AC|DC est en $O(e+nd)$.

5.5.2. Complexité en temps

La complexité d'une restriction est bien évidemment celle de AC-3 c'est à dire $O(ed^3)$.

Propose a trivialement un coût en $O(d^2)$. Regardons la complexité de *Propagate* de manière globale. Pour chaque variable au plus d valeurs seront restaurées. Pour chacune de ces valeurs on regarde toutes les contraintes adjacentes et pour chaque contrainte on fait au plus d tests de consistance pour trouver une nouvelle valeur supportée. On réalise donc au plus $2ed^2$ tests de consistance et la complexité de *Propagate* est en $O(ed^2)$. La procédure *Filter* est une version modifiée de AC-3 et comme elle se contente de filtrer les valeurs figurant dans *Restorable* sa complexité est en $O(edd_a^2)$ où d_a est le nombre maximal de valeurs restorables pour une variable. Dans le pire des cas toutes les valeurs figurent dans *Restorable* et $d_a=d$. La complexité des relaxations est donc en $O(d^2 + ed^2 + ed^3)$ c'est à dire en $O(ed^3)$.

6. Analyse des performances

Les résultats portant sur AC|DC ont été obtenus à partir d'une implémentation basée sur AC-3 légèrement améliorée en ce sens qu'on vide D et qu'on arrête le processus dès qu'un domaine vide est rencontré. Toutefois, aucune autre amélioration n'a été apportée aux implémentations de DnAC-4, DnAC-6 et AC|DC. Notamment on ne tire aucun profit de l'appartenance des contraintes à une classe particulière.

6.1. Résultats expérimentaux sur le problème du zèbre

Dans le domaine des CSP, le problème du zèbre est devenu un test incontournable pour les algorithmes. Il est composé à la fois de contraintes molles et de contraintes dures ce qui le rend relativement proche des problèmes réels.

Sous sa forme originelle il n'est toutefois pas exploitable par les algorithmes traitant les DCSP. C'est pourquoi les résultats présentés dans cette section sont ceux obtenus sur une version dynamique du problème que nous allons à présent détailler.

6.1.1. L'énoncé du problème

Présentons d'abord le problème sous sa forme classique;

1. Il y a 5 maisons, chacune d'une couleur différente et habitées par des hommes de nationalités différentes ayant des boissons, des marques de cigarettes et des animaux différents.
 2. L'anglais vit dans la maison rouge.
 3. L'espagnol possède un chien.
 4. On boit du café dans la maison verte.
 5. L'ukrainien bois du thé.
 6. La maison verte est immédiatement à droite de la maison ivoire.
 7. Le fumeur de Old-Gold possède des escargots.
 8. On fume des Kools dans la maison jaune.
 9. On boit du lait dans la maison centrale.
 10. Le norvégien vit dans la première maison à gauche.
 11. Le fumeur de Chesterfield vit à coté du propriétaire du renard.
 12. On fume des Kools à côté de la maison du propriétaire du cheval.
 13. Le fumeur de Lucky-Strike boit du jus d'orange.
 14. Le japonais fume des Parliament.
 15. Le norvégien vit à coté de la maison bleue.
- Question : Qui boit de l'eau ? et qui possède le zèbre ?

Classiquement ce problème est représenté par un réseau de contraintes à 25 variables, une pour chacune des cinq couleurs, des cinq nationalités, des cinq animaux, des cinq boissons et des cinq marques de cigarettes. Chaque variable prend ses valeurs dans $\{1, 2, 3, 4, 5\}$ pour désigner la position d'une maison.

Pour rendre dynamique ce problème nous nous sommes inspirés des deux constatations suivantes. Habituellement, l'utilisateur ajoute des contraintes tant que le nombre de solutions est élevé. Le bon sens veut également que l'on supprime des contraintes dès que le problème devient inconsistant et donc a fortiori quand l'un des domaines est vide. Le protocole suivi a eu pour but de calquer ces principes dans la version dynamique du problème du zèbre. Pour cela nous avons ajouté les six contraintes suivantes :

16. Le buveur de lait habite la maison rouge.
17. Le buveur de jus d'orange habite la maison ivoire.
18. L'ukrainien possède un cheval.
19. Le buveur de jus d'orange habite la maison bleue.
20. Le propriétaire du zèbre ne boit pas de café.
21. L'espagnol fume des Kools.

Chacune des trois dernières contraintes rend inconsistant le problème classique du zèbre. Les contraintes qui correspondent à la ligne 1 sont l'essence du problème. Elles sont placées dans le CSP initial et ne seront jamais retirées. Le DCSP généré commence par une phase de restrictions. A chaque étape nous choisissons une contrainte qui correspond à une des lignes entre 2 et 21 non encore traduite dans le problème et on l'ajoute au CSP. Cette phase prend fin quand on a atteint le "seuil". Ce dernier est l'instant à partir duquel il n'est plus possible d'ajouter des contraintes. Il correspond donc ici au fait qu'il existe un domaine vide. Dans un CSP

auquel on ajouterait des contraintes plus molles il pourrait également être l'instant auquel le graphe des contraintes devient complet. Après avoir atteint ce seuil, on réalise 40 opérations dont la nature est une restriction si aucun des domaines n'est vide, et une relaxation sinon.

6.1.2. Les indicateurs de performance

Pour évaluer les performances de leurs algorithmes, les auteurs ont des critères différents. Afin que chacun trouve des mesures qui lui sont familières, les résultats mentionnés regroupent la plupart des méthodes.

- La première mesure est le nombre de tests de consistance effectués. Bien que ce nombre soit relativement déconnecté du temps d'exécution, il est intéressant car le test de consistance est l'opération de base dans les algorithmes relatifs aux CSP. De plus, si dans les cas courants un test se résume à la consultation d'une table, dans certains cas il peut s'agir d'une opération beaucoup plus coûteuse dont on veut avant tout minimiser le nombre.

Grâce à ses listes de supports, DnAC-4 n'effectue pas de tests de consistance lors des relaxations. C'est pourquoi nous comparons le nombre global de tests de consistances effectués à la fois durant les restrictions (y compris celles avant le seuil) et les relaxations.

- La seconde mesure que nous appellerons "nombre d'opérations atomiques" (N.O.A.) correspond à celle qu'utilisa C. Bessière pour l'algorithme AC-6. On compte le nombre d'opérations atomiques et de tests effectués ; plus précisément, on compte 1 pour tout ensemble d'instructions en $O(1)$. Le nombre ainsi obtenu est plus significatif que le simple nombre de tests car il est plus proche du temps d'exécution tout en restant indépendant de l'implémentation. Nous présenterons indépendamment le nombre moyen obtenu pour un ajout de contrainte, calculé sur l'ensemble des restrictions effectuées y compris celles avant le seuil, et le nombre moyen obtenu sur les relaxations.

- Le dernier critère est le temps d'exécution. Pour qu'il soit significatif, les mêmes structures de données et primitives ont été utilisées pour l'implémentation des trois algorithmes et toutes les mesures ont été réalisées sur le même ordinateur. La durée considérée est le temps mis pour réaliser à la fois les 40 opérations et les restrictions qui précèdent le seuil. Il rend donc compte à la fois des restrictions et des relaxations.

Les résultats obtenus sur ces trois critères seront présentés sous forme de gains en pourcentages. Nous considérerons d'une part les gains de DnAC-6 sur DnAC-4 et d'autre part ceux de AC|DC sur DnAC-4. Le signe du gain indique à qui va l'avantage. Ainsi un gain de 40 % de DnAC-6 sur DnAC-4 sur le temps d'exécution, signifie que DnAC-6 a gagné 40 % sur le temps d'exécution de DnAC-4. Par contre, si le gain est de -40 %, c'est que DnAC-4 a gagné 40 % sur DnAC-6.

6.1.3. Les résultats obtenus

Afin que les performances exprimées soient significatives, les résultats présentés par la figure 11 sont une moyenne sur 50 problèmes générés de la manière énoncée en 6.1.1.

	N.O.A. pour une restriction	N.O.A. pour une relaxation	Nombre de tests de consistance global	Temps d'exécution global
Gains de DnAC-6 sur DnAC-4	68,90 %	34,73 %	-90,28 %	38,08 %
Gains de AC DC sur DnAC-4	81,62 %	-7,83 %	-92,80 %	36,41 %

Figure 11. Les résultats obtenus sur le problème du zèbre

On peut noter que le critère du nombre d'opérations élémentaires est significatif. En effet, sachant que le temps passé sur les relaxations est bien plus imposant que celui des restrictions (On remet de nombreuses valeurs car on ne retire une contrainte que lorsqu'on a trouvé un domaine vide), les résultats des deux premières colonnes sont en accord avec ceux sur le temps d'exécution.

Le résultat le plus frappant sur le problème du zèbre est que DnAC-4 réalise dix fois moins de tests de consistance que ses concurrents. Pour maintenir ses listes de supports, cet algorithme doit lors d'une restriction procéder à tous les tests de consistance sur la nouvelle contrainte, y compris ceux concernant les valeurs ne figurant plus dans D . Mais ce surcroît de travail n'est pas vain car pour réaliser une relaxation on n'a plus qu'à consulter les compteurs et les listes de valeurs supportées. Cette structure permet donc de limiter le nombre de tests de consistance réalisés mais exige un travail considérable sur les restrictions. C'est pourquoi AC|DC comparé à DnAC-4 réalise un gain conséquent en temps d'exécution malgré ses pertes sur les relaxations. Bien qu'ayant stocké moins d'informations que DnAC-4, DnAC-6 est le plus performant sur les relaxations et enregistre lui aussi des gains sur DnAC-4 en temps d'exécution.

6.2. Résultats expérimentaux sur les problèmes aléatoires

6.2.1. La génération

Les paramètres d'appel du générateur sont n , d , $pumin$ et $pumax$. Le générateur déclare n variables ayant chacune un domaine de taille d . Ensuite il crée un ensemble de $(n \times (n-1)/4)$ relations noté R dont la moitié sont des relations arithmétiques ($=$, \neq , \leq , $>$, ...) et dont les autres ont été choisies de manière équiprobable dans l'ensemble des relations qui possèdent un taux de satisfiabilité compris entre $pumin$ et $pumax$. La taille de cet ensemble, bien qu'arbitraire, n'est

pas fortuite. En effet il est rare qu'un CSP soit composé de contraintes toutes différentes entre elles et en prenant un ensemble de cette taille on reproduit cette constatation. La constitution de cet ensemble n'est bien évidemment pas prise en compte dans la durée d'exécution. La première phase des DCSP générés consiste à réaliser des restrictions correspondant à des relations choisies de manière équiprobable dans R jusqu'à ce qu'il existe un domaine vide ou que le graphe des contraintes soit complet. Au terme de cette phase, on dit qu'on a atteint le seuil et à partir de cet instant on va réaliser quarante opérations. Ces dernières sont des relaxations s'il existe un domaine vide ou si le graphe des contraintes est complet et des restrictions dans le cas contraire.

6.2.2. Le protocole de test

L'échantillon de réseaux sur lequel portèrent les tests est défini par $n \in \{10, 14, 18, 22, 26, 30, 34, 38\}$, $d \in \{2, 6, 10, 14, 18, 22, 26, 30\}$ et $n+d \leq 40$ avec quatre intervalles de probabilité pour le taux de satisfiabilité des relations aléatoirement générées : $[5, 35]$, $[35, 60]$, $[60, 95]$ et $[5, 95]$. Pour ne pas accabler le lecteur de résultats nous ne présentons que la tranche $[5, 95]$ qui est assez représentative. Afin d'obtenir des résultats significatifs, une moyenne a été faite sur 25 instances de chacun des 144 réseaux ci-dessus.

Les critères utilisés sont les mêmes qu'en 6.1.2.

Pour faciliter leur lecture, les pourcentages présentés par les figures 12 et 13 sont des valeurs arrondies à l'entier le plus proche et une représentation graphique est réalisée.

6.2.3. Les résultats obtenus

D'après la figure 12 on constate que comme sur le problème du zèbre, DnAC-6 comparé à DnAC-4 réalise des gains importants à la fois sur les restrictions et les relaxations et donc également sur le temps d'exécution. Ces gains sont d'autant plus importants que les domaines sont grands mais ils varient peu selon n . Il n'y a que sur le critère du nombre de tests de consistance que DnAC-4 est plus performant que DnAC-6 avec des gains de l'ordre de 50 %.

L'évolution des gains de AC|DC sur DnAC-4 présentée par la figure 13 est très différente. Il n'y a que sur les problèmes ayant de petits domaines que AC-3 est plus performant que DnAC-6 lors des restrictions car la complexité de AC-3 est en $O(ed^3)$ et ses performances diminuent fortement quand d augmente. Le prix de la simplicité de AC|DC est ses faibles performances sur les relaxations. Non seulement l'estimation qu'il fait des valeurs à remettre lors du retrait d'une contrainte est plus grossière, mais la phase de retrait des valeurs non viables remises à tort est réalisée par une procédure de filtrage dérivée de l'algorithme non optimal AC-3.

En temps d'exécution DnAC-4 et AC|DC sont comparables. Par contre AC|DC est des trois algorithmes celui qui réalise le plus de tests de consistance et l'écart avec ses concurrents est d'autant plus marqué que d est grand.

Sur les autres intervalles de taux de satisfiabilité pour les contraintes aléatoires, les résultats sont proches de ceux constatés sur l'intervalle $[5, 95]$ que nous venons de voir. Les courbes ont une allure similaire mais avec des gains plus ou moins élevés selon la dureté des contraintes. La principale constatation que l'on peut faire est que DnAC-4 est d'autant plus efficace que les contraintes sont dures car alors ses listes de valeurs supportées sont petites et donc avantageuses comparées à la nécessité pour les autres méthodes de scruter de nombreuses valeurs pour trouver un support. DnAC-4 est par contre désavantagé quand on réalise une opération alors qu'il reste peu de valeurs dans D comparé à dom car dans ses listes figurent tous les supports y compris ceux retirés de D .

La comparaison des trois algorithmes ne serait pas complète si nous ne parlions pas de l'espace mémoire occupé par chacun d'eux pour résoudre un problème. C'est pourquoi nous rappelons dans la figure 14 les complexités dans le pire des cas en espace et en temps de chacun des algorithmes présentés.

	DnAC-4	DnAC-6	AC DC
Complexité en espace	$O(ed^2 + nd)$	$O(ed + nd)$	$O(e + nd)$
Complexité en temps	$O(ed^2)$	$O(ed^2)$	$O(ed^3)$

Figure 14. Les complexités dans le pire des cas des algorithmes présentés.

7. Conclusion

Dans cet article nous avons étudié les algorithmes incrémentaux d'arc-consistance sur les DCSP. L'étude comparative de ces algorithmes sur une version dynamique du problème du zèbre et sur des DCSP aléatoirement générés nous a permis de mettre en valeur les qualités et les défauts de chacun d'eux.

Les listes de valeurs supportées de DnAC-4 sont très coûteuses en espace mais elles permettent d'effectuer relativement peu de tests de consistance. On peut recommander l'emploi de DnAC-4 dans les applications pour lesquelles réaliser un test de consistance est une opération coûteuse et pour filtrer des DCSP constitués de contraintes très dures car ses listes de supports sont dès lors courtes et donc efficaces.

AC|DC a les moins bons résultats sur les relaxations car pour posséder une bonne complexité en espace cet algorithme ne possède aucune structure de données globale imposante qui pourrait lui permettre de déterminer plus finement l'ensemble des valeurs à remettre. En utilisant AC-3 pour réaliser les restrictions il parvient

toutefois à être comparable à DnAC-4 en temps d'exécution sur les problèmes ayant des domaines de taille moyenne. Il est regrettable que la complexité non optimale de cet algorithme fasse très vite sentir ses effets quand d augmente car sa bonne complexité en espace aurait pu favoriser son utilisation sur les problèmes qui ont des domaines imposants. Ses principaux atouts résident dans sa simplicité et dans sa facilité d'extension.

Du point de vue de la complexité en espace, DnAC-6 apparaît comme un compromis en utilisant des structures de données moins volumineuses que celles de DnAC-4. Ces dernières lui permettent non seulement d'effectuer des restrictions de manière efficace mais aussi d'avoir les meilleures performances sur les relaxations

8. Références

- [BES 91] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In Proc. AAAI-91, Anaheim CA, (1991) 221-226
- [BES 92] C. Bessière. Systèmes à contraintes évolutifs en intelligence artificielle. Thèse de doctorat, université de Montpellier II (1992) 53-89
- [BES 94] C. Bessière. Arc-consistency and arc-consistency again. Artificial Intelligence 65 (1994) 179-190
- [DEB 94] R. Debruyne. DnAC-6. Technical Report 94054, LIRMM (1994)
- [MAC 77] A.K. Mackworth. Consistency in networks of relations. Artificial Intelligence 8 (1977) 99-118
- [M&F 85] A.K. Mackworth & E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial Intelligence 25 (1985) 65-74
- [MES 89] P. Meseguer. Constraint satisfaction problems : An overview. AICOM Vol.2 March 1989
- [M&H 86] R. Mohr & T.C.Henderson. Arc and path consistency revisited. Artificial Intelligence 28 (1986) 225-233
- [N&B 94] B. Neveu & P. Berlandier. Arc-consistency for dynamic constraint satisfaction problems : An RMS free approach. In Proc. ECAI-94, Workshop on "Constraint satisfaction issues raised by practical applications", Amsterdam, The Netherlands (1994).
- [PRO 92] P. Prosser, C. Conway, C. Muller. A distributed constraint maintenance system. In Proc. "Les systèmes experts et leurs applications", Avignon, France (1992).