Raisonnement par contraintes

Christian Bessiere
CNRS, Université de Montpellier

21 juin 2010

Préambule

Dans ce chapitre je vais présenter succinctement le raisonnement par contraintes. Ce domaine de recherche de l'intelligence artificielle est plus connu aujourd'hui sous le nom de programmation par contraintes, notamment depuis qu'il est utilisé à grande échelle pour résoudre des problèmes combinatoires dans des applications industrielles et surtout depuis qu'il s'est enrichi des apports de la programmation logique pour les aspects langage et de la recherche opérationnelle pour la propagation de contraintes complexes. Mais vu le thème de ce livre, je vais volontairement rester sur une présentation très IA du domaine. On pourra trouver que ce chapitre souffre d'un tropisme français. Si c'est vrai que je n'ai rien fait pour l'éviter, on doit quand même garder en tête que la communauté française de programmation par contraintes a toujours été à la pointe et est aujourd'hui sans conteste celle qui a la plus grosse production scientifique.

0.1 Historique

La notion de contrainte comme restriction des combinaisons de valeurs que peuvent prendre un ensemble de variables est suffisamment naturelle pour être apparue assez tôt dans l'histoire de l'intelligence artificielle. On peut citer notamment les travaux de Fikes sur le système REF-ART en 1970, ou ceux de Waltz sur l'interprétation de contours en 1972. On a cependant l'habitude d'associer la naissance du raisonnement par contraintes au papier fondateur de Montanari, qui en 1974, définit formellement pour la première fois ce qu'est un réseau de contraintes. Ce papier sera suivi d'articles tout aussi importants par Mackworth et surtout Freuder, qui donneront aux recherches leur orientation vers les consistances locales et la propagation de contraintes, notions propres au raisonnement par contraintes.

La communauté française a eu pour pionniers Laurière et son système ALICE (Laurière, 1978) ainsi que Mohr avec l'algorithme de propagation AC4 (Mohr and Henderson, 1986). Mais la place importante que la communauté française a aujourd'hui dans ce domaine doit sans doute beaucoup aux deux projets BAHIA et CSPFlex, du programme de recherches coordonnées IA (PRC IA) au début des années 90. Ils ont permis de structurer la communauté et ont donné un élan indéniable aux chercheurs qui ont eu la chance de participer à ce bouillon d'idées sur un domaine à ses débuts.

0.2 Introduction

Un réseau de contraintes est composé de variables qui prennent chacune leur valeur dans leur domaine respectif, et de contraintes qui restreignent les combinaisons de valeurs possibles entre certaines variables. Un problème de satisfaction de contraintes (CSP pour *Constraint Satisfaction Problem*) consiste à savoir si un réseau de contraintes donné accepte ou non une solution, c'est à dire une assignation de valeurs aux variables qui satisfasse toutes les contraintes. Par exemple, on peut représenter le problème du coloriage de carte présenté en figure 1 par un réseau de contraintes où chaque pays est représenté par une variable, son domaine de valeurs étant l'ensemble des couleurs disponibles pour ce pays (voir figure 2). Chaque paire de variables représentant deux pays limitrophes est reliée par une contrainte spécifiant que les va-

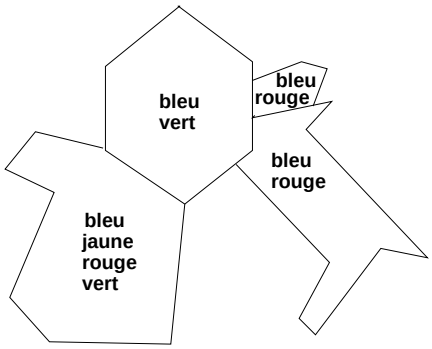

FIG. 1: Problème de coloriage de carte où chaque pays n'a qu'un ensemble restreint de couleurs autorisées.

leurs prises par ces deux variables doivent être différentes. Il est aisé de se convaincre qu'une solution à ce réseau de contraintes correspondra à un coloriage satisfaisant de notre carte.

Le premier intérêt des CSP, déjà mis en avant par Freuder en 1993 lors d'un tutorial à IJCAI, est de fournir un formalisme stable sur lequel des algorithmes de résolution travaillent. Si vous arrivez à représenter votre problème en réseau de contraintes tel qu'une solution du réseau soit la représentation d'une solution de votre problème, vous pourrez utiliser n'importe quel algorithme de satisfaction de contraintes pour trouver votre solution.

Les utilisations du raisonnement par contraintes sont nombreuses. Parmi les premiers succès enregistrés, on peut citer l'optimisation du parcours des excavatrices à charbon dans des mines ou le planning des infirmières dans des hôpitaux. Depuis, on a vu le raisonnement par contraintes utilisé pour affecter des fréquences à des liaisons radio ou téléphone, affecter des trains à des quais, optimiser des chaînes de montage de voiture ou chercher des structures dans des séquences d'ARN.

Il existe d'autres formalismes qui définissent un standard de modélisation à partir duquel divers algorithmes peuvent être utilisés. SAT ou la programmation linéaire en nombres entiers en sont des exemples. Un avantage des CSP par rapport à ces formalismes est l'expressivité disponible. Certains 'motifs' se retrouvent dans beaucoup d'applications réelles, comme par exemple la contrainte 'tous différents' qui impose à un ensemble de variables de prendre des valeurs différentes. Ces motifs peuvent être encapsulés dans une seule contrainte qui permettra à l'utilisateur de modéliser facilement tout un morceau de son problème et aux algorithmes de raisonner globalement sur ces motifs, qui autrement auraient dû être décomposés en sous-parties. Raisonner à partir du motif de départ permet de coller à la définition du problème et dans certains cas d'en déduire plus de choses que sur n'importe quelle décomposition en sous-problèmes de taille polynomiale.

0.3 Définitions

Nous donnons ici les définitions nécessaires à la compréhension de la suite de ce chapitre. Nous avons autant que possible favorisé la simplicité quitte à parfois perdre un peu de rigueur

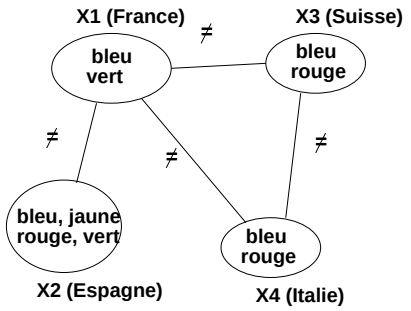

FIG. 2: Réseau de contraintes modélisant le problème de coloriage de carte de la figure 1.

quand il n'y a pas d'ambiguïté possible sur l'interprétation.

Définition 1 (Réseau de contraintes) *Un réseau de contraintes $P = (X, D, C)$ est composé de :*

- *une séquence finie $X = \{x_1, x_2, \ldots, x_n\}$ de* variables *entières,*
- *un* domaine *pour X, c'est à dire un ensemble $D = D(x_1) \times \ldots \times D(x_n)$, où $D(x_i) \subset \mathbb{Z}$ est fini et donné en extension, et*
- *un ensemble $C = \{c_1, \ldots, c_e\}$ de* contraintes*. Une contrainte $c_j \in C$ est une relation définie sur une séquence de variables $X(c_j) = (x_{j_1}, \ldots, x_{j_{|X(c_j)|}})$, appelée la* portée *de c_j. c_j est un sous-ensemble de $\mathbb{Z}^{|X(c_j)|}$.*

Les valeurs listées dans $D(x_i)$ pourraient être de n'importe quel type, mais cela simplifie grandement le discours de considérer qu'elles sont toutes des entiers, ce qui n'est pas une restriction vu que $D(x_i)$ est fini. Un élément de $\mathbb{Z}^{|X(c_j)|}$ est appelé *tuple*. Un tuple de $\mathbb{Z}^{|X(c_j)|}$ est dit *valide* si et seulement s'il appartient à $D^{X(c_j)} = D(x_{j_1}) \times \ldots \times D(x_{j_{|X(c_j)|}})$. Les tuples de c_j sont appelés *autorisés* par c_j, les autres tuples de $\mathbb{Z}^{|X(c_j)|}$ sont *interdits* par c_j.

Exemple 1 L'exemple de coloriage de carte de la figure 1 peut être modélisé par le réseau de la figure 2. Les variables $x_1, \ldots, x_4$ correspondent respectivement aux pays France, Espagne, Suisse, Italie. Les domaines sont $D(x_1) = \{B, V\}, D(x_2) = \{B, J, R, V\}, D(x_3) = D(x_4) = \{B, R\}$, où B, J, R, V sont des entiers qui représentent les couleurs. La contrainte entre France et Espagne, appelons-la c_1, qui garantit que France (variable x_1) et Espagne (variable x_2) prendront des couleurs différentes, peut se définir par $X(c_1) = (x_1, x_2)$ et $c_1 = \{(B, J), (B, R), (B, V), (V, B), (V, J), (V, R)\}$ ou plus simplement par $x_1 \neq x_2$. $\diamondsuit$

Le concept d'instanciation est essentiel à la suite de ce chapitre.

Définition 2 (Instanciation) *Une* instanciation I sur *un ensemble $Y \subseteq X$ de variables est une fonction qui affecte à chaque variable x_i de Y une valeur de son domaine $D(x_i)$. Si W est un sous-ensemble de Y, on note $I[W]$ la restriction (ou projection) de I aux variables de W. On dit d'une instanciation qu'elle* viole *une contrainte c_j si et seulement si $X(c_j) \subseteq Y$ et $I[X(c_j)] \notin c_j$. On dit d'une instanciation qu'elle est* localement consistante *si et seulement si aucune des contraintes de C n'est violée.*

Algorithm 1: Backtrack chronologique

function BT (**(**P : problem ; I : instanciation**) : Boolean** ;
1 **if** $|I| = n$ **then** imprimer I ; return **true**;
2 choisir une variable x_i pas encore dans I;
3 **foreach** $v_i \in D(X_i)$ **do**
4 **if** $I \cup \{(X_i, v_i)\}$ *est localement consistante* **then**
5 **if** $BT(P, I \cup \{(X_i, v_i)\})$ **then** return **true**;
6 return **false**;

Exemple 2 Sur notre exemple de coloriage de carte, $((x_1, B); (x_2, J); (x_4, B))$ est une instanciation sur (x_1, x_2, x_4) (c'est à dire sur France, Espagne et Suisse) qui n'est pas localement consistante car elle viole la contrainte entre x_1 et x_4. $((x_1, B); (x_2, J); (x_4, R))$ est une instanciation sur (x_1, x_2, x_4) localement consistante. ♢

Définition 3 (Solution) *Une* solution S *d'un réseau de contraintes* $P = (X, D, C)$ *est une instanciation sur* X *qui ne viole aucune contrainte de* C*. On note* $Sol(P)$ *l'ensemble de toutes les solutions de* P*.*

Définition 4 (CSP) *On appelle* problème de satisfaction de contraintes *(noté CSP) le problème défini par :*

Instance : *Un réseau de contraintes* $P = (X, D, C)$

Question : $Sol(P) \neq \emptyset$ *?*

Théorème 1 *CSP est NP-complet.*

Maintenant que nous avons cerné le modèle et la question centrale que l'on se pose, le chapitre suivant va recenser les techniques de base qui permettent de traiter cette question.

0.4 Backtracking

Comme dans tout problème NP-complet, l'énumération des possibilités est une technique naturelle pour résoudre le problème, c'est à dire trouver une solution. La méthode du "générer et tester" étant quand même un peu bête, nous allons directement présenter sa version améliorée qu'est le *backtrack chronologique (BT)* (Golomb and Baumert, 1965).

La fonction BT présentée en figure 1 prend en paramètre un réseau de contraintes P et une instanciation partielle localement consistante I. L'appel $BT(P, \emptyset)$ imprime la première solution trouvée et renvoie **true** si P a des solutions, sinon renvoie **false**. Après avoir testé si I n'est pas une instanciation complète (ligne 1), la fonction sélectionne une variable x_i non instanciée (ligne 2) puis essaie tour à tour toutes ses valeurs jusqu'à en trouver une qui s'étende en une solution. Une fois une valeur v_i choisie, et si l'instanciation I augmentée de l'affectation $x_i \leftarrow v_i$ est localement consistante (ligne 4), la fonction est rappelée récursivement avec la nouvelle instanciation (ligne 5). Si toutes les valeurs de $D(x_i)$ ont été essayées sans succès, la fonction renvoie **false** (ligne 6). L'espace exploré par BT est appelé *arbre de recherche*. Les affectations de valeurs en sont les noeuds et ont un arc entrant provenant de la dernière affectation localement consistante. La racine de l'arbre est l'instanciation vide (voir figure 3).

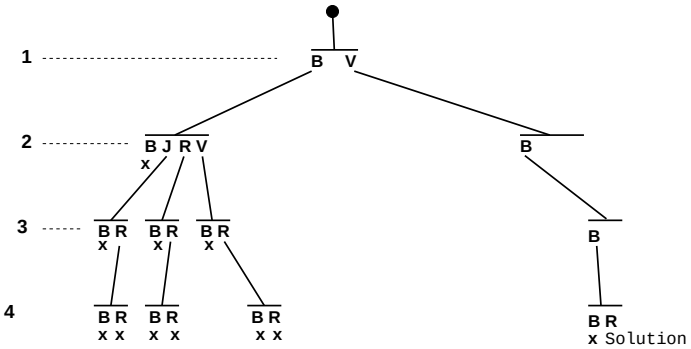

FIG. 3: Arbre de recherche de BT sur le réseau de contraintes de l'exemple 1.

0.5 Propagation de contraintes

Le raisonnement par contraintes utilise plusieurs moyens pour réduire la taille de l'espace de recherche exploré par la procédure BT. La propagation de contraintes est le principal de ces moyens. La propagation de contraintes consiste à explicitement interdire des valeurs ou combinaisons de valeurs pour des variables parce qu'autrement, un ensemble de contraintes ne pourra pas être satisfait. Par exemple, dans un problème contenant les deux variables x_1 et x_2 de domaines $1..10$, et une contrainte spécifiant que $|x_1 - x_2| > 5$, si on propage cette contrainte on peut interdire les valeurs 5 et 6 pour x_1 et pour x_2. Supprimer ces valeurs des domaines de x_1 et x_2 est un moyen de réduire l'espace des combinaisons qu'une procédure de recherche doit explorer.

La propagation de contraintes peut être présentée selon deux axes : *les consistances locales* et *l'itération de règles de réduction*. Une consistance locale définit une propriété que le problème doit satisfaire *après* la phase de propagation. Le côté opérationnel n'est pas spécifié. A l'inverse, l'approche par règles de réductions décrit le mécanisme de propagation lui-même. Les règles sont des conditions sur le type d'opérations de réduction qui peuvent être appliquées au problème. Je présenterai la propagation de contraintes principalement au travers des consistances locales.

0.5.1 Consistance sur une contrainte à la fois

Arc consistance

L'arc consistance est la plus ancienne et la plus connue des consistances locales. Elle garantit que chaque valeur de chaque variable est compatible avec chaque contrainte prise séparément. L'article fondateur sur l'arc consistance est dû à Mackworth, qui le premier a clairement défini ce concept (Mackworth, 1977).

Exemple 3 Soit un réseau P contenant les trois variables x_1, x_2 et x_3, de domaines $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$, et les contraintes $x_1 = x_2$ et $x_2 < x_3$. P n'est pas arc consistant parce que certaines valeurs sont incompatibles avec certaines contraintes. En examinant la

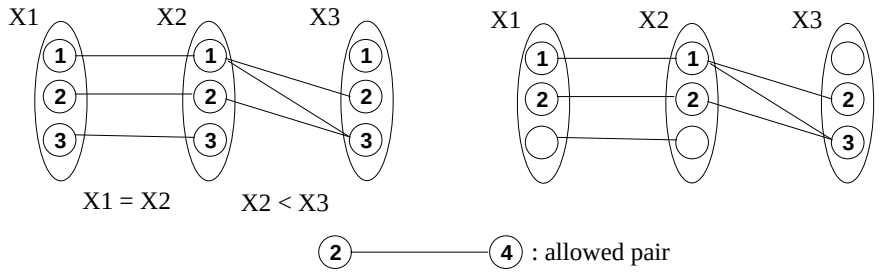

FIG. 4: Réseau de l'exemple 3 avant l'arc consistance (gauche) et après (droite).

contrainte $x_2 < x_3$ on voit que la valeur 3 pour x_2 doit être supprimée parce qu'il n'y a pas de valeur plus grande que 3 dans $D(x_3)$. On peut aussi supprimer la valeur 1 de $D(x_3)$ à cause de cette même contrainte $x_2 < x_3$. Supprimer 3 de $D(x_2)$ provoque la suppression de la valeur 3 pour x_1 à cause de $x_1 = x_2$. Maintenant, toutes les valeurs restantes sont compatibles avec toutes les contraintes. ♢

Je donne la définition d'arc consistance dans sa forme la plus générale, c'est à dire pour les contraintes d'arité quelconque. Dans ce cas, elle est souvent appelée arc consistance généralisée.

Définition 5 (Arc consistance (AC)) *Etant donnés un réseau $P = (X, D, C)$, une contrainte $c \in C$, et une variable $x_i \in X(c)$,*

- *Une valeur $v_i \in D(x_i)$ est* consistante avec *c dans D si et seulement si il existe un tuple valide τ satisfaisant c tel que $v_i = \tau[\{x_i\}]$. Un tel tuple est appelé* support *pour (x_i, v_i) sur c.*
- *La contrainte c est arc consistante sur D si et seulement si toutes les valeurs de toutes les variables dans $X(c)$ ont un support sur c.*
- *Le réseau P est* arc consistant *si et seulement si toutes les contraintes de C sont arc consistantes sur D.*

Appliquer l'arc consistance sur le réseau $P = (X, D, C)$ signifie calculer la *fermeture arc consistante* $AC(P)$ de P. $AC(P)$ est le réseau (X, D_{AC}, C) où $D_{AC} = \cup\{D' \subseteq D \mid (X, D', C)$ est arc consistant$\}$. $AC(P)$ est arc consistant et est *unique*. Il a les mêmes solutions que P. $AC(P)$ se calcule en supprimant itérativement les valeurs qui n'ont pas de support sur une contrainte jusqu'à atteindre un point fixe où toutes les valeurs ont des supports sur toutes les contraintes.

On verra par la suite qu'appliquer l'arc consistance dans un réseau de contraintes est une tâche essentielle à la recherche de solutions. Proposer des algorithmes efficaces d'arc consistance a donc toujours été une préoccupation importante de la communauté. Les idées utilisées pour améliorer les algorithmes d'arc consistance sont d'ailleurs souvent utilisées pour améliorer les algorithmes de propagation pour d'autres types de consistances que nous verrons plus loin.

AC3 est l'algorithme d'arc consistance le plus connu et le plus simple, même si ce n'est pas le plus performant. Il a été proposé dans (Mackworth, 1977) pour les contraintes binaires.

Algorithm 2: AC3 (aussi noté GAC3 sur les contraintes non-binaires)

```
function Revise3(in x_i : variable ; c : constraint) : Boolean ;
 1 CHANGE ← false;
 2 foreach v_i ∈ D(x_i) do
 3     if ∄ un tuple valide τ ∈ c avec τ[x_i] = v_i then
 4         supprimer v_i de D(x_i);
 5         CHANGE ← true;
 6 return CHANGE ;

function AC3(in X : set) : Boolean ;
 7 Q ← {(x_i, c) | c ∈ C, x_i ∈ X(c)};
 8 while Q ≠ ∅ do
 9     prendre (x_i, c) de Q;
10     if Revise(x_i, c) then
11         if D(x_i) = ∅ then return false ;
12         else Q ← Q ∪ {(x_j, c') | c' ∈ C ∧ c' ≠ c ∧ x_i, x_j ∈ X(c') ∧ j ≠ i};
13 return true ;
```

L'algorithme 2 le décrit dans sa forme pour contraintes d'arité quelconque.

Le composant principal d'AC3 est la révision d'un arc, c'est à dire la suppression des valeurs d'une variable inconsistantes sur une contrainte. Le nom 'arc' provient du cas binaire. La fonction `Revise`(x_i, c) passe chaque valeur v_i de $D(x_i)$ une par une (ligne 2), et explore $D^{X(c)\setminus\{x_i\}}$ pour trouver un support sur c pour v_i (ligne 3). Si un tel support n'est pas trouvé, v_i est supprimé de $D(x_i)$ et le fait que $D(x_i)$ ait été modifié est mémorisé (lignes 4–5). La fonction renvoie vrai si le domaine $D(x_i)$ a été modifié, faux sinon (ligne 6).

L'algorithme AC3 est une simple boucle qui révise les arcs jusqu'à ce que plus aucune modification ne soit faite. Les domaines sont alors tous arc consistants sur toutes les contraintes. Pour éviter trop d'appels inutiles à `Revise`, AC3 maintient une liste Q de toutes les paires (x_i, c) pour lesquelles on n'est pas sûrs que $D(x_i)$ soit arc consistant sur c. En ligne 7, Q est initialisée avec toutes les paires (x_i, c) possibles telles que $x_i \in X(c)$. La boucle principale (ligne 8) prend les paires (x_i, c) une par une dans Q (ligne 9) et appelle `Revise`(x_i, c) (ligne 10). Si $D(x_i)$ est vidé, AC3 renvoie faux (ligne 11). Sinon, si $D(x_i)$ est modifié, il est possible qu'une valeur d'une autre variable x_j ait perdu ses supports sur une contrainte c' impliquant à la fois x_i et x_j. Donc, toutes les paires (x_j, c') telles que $x_i, x_j \in X(c')$ doivent être mises à nouveau dans Q (ligne 12). Lorsque Q est vide, AC3 renvoie vrai (ligne 13) puisque tous les arcs ont été révisés et toutes les valeurs restantes de toutes les variables sont consistantes avec toutes les contraintes.

AC3 est polynomial en l'arité des contraintes. Il est en $O(er^3d^{r+1})$, où r est l'arité la plus grande des contraintes de C. Sur des réseaux de contraintes binaires cela donne $O(ed^3)$. La complexité d'AC3 n'est pas optimale car `Revise` ne mémorise rien sur les calculs faits pour trouver les supports et doit donc faire et refaire les mêmes tests de contraintes à chaque appel.

De nombreux travaux ont été publiés pour améliorer la complexité d'AC3. Mohr et Henderson (1986) ont proposé AC4, le premier algorithme optimal d'arc consistance ($O(ed^2)$ sur les contraintes binaires et $O(erd^r)$ en général), au prix d'une structure de données très coûteuse. Bessiere et al. (2005) ont proposé AC2001, qui a l'avantage d'être basé sur le même

schéma qu'AC3 tout en ayant une complexité optimale grâce à une structure de pointeurs facile à implémenter.

Consistance aux bornes

Quand une contrainte a une arité trop grande ou quand les domaines sont très grands, l'arc consistance peut s'avérer trop coûteuse. Une alternative est d'utiliser la *borne consistance (BC)*, une consistance plus faible qui tire parti du fait que les domaines sont composés d'entiers et héritent donc de l'ordre total sur $\mathbb{Z}$. On peut définir la plus petite et la plus grande valeur d'un domaine $D(x_i)$, notées $min_D(x_i)$ and $max_D(x_i)$ respectivement, et appelées les *bornes* de $D(x_i)$.

Définition 6 (Borne consistance) *Etant donnés un réseau $P = (X, D, C)$ et une contrainte c, un* support aux bornes τ *sur c est un tuple qui satisfait c et tel que pour tout $x_i \in X(c), min_D(x_i) \leq \tau[x_i] \leq max_D(x_i)$. Une contrainte c est* borne consistante *si et seulement si pour tout $x_i \in X(c)$, $(x_i, min_D(x_i))$ et $(x_i, max_D(x_i))$ appartiennent à un support aux bornes sur c. P est borne consistant si et seulement si toutes ses contraintes sont borne consistantes.*

Exemple 4 Soient les variables x, y, z, les domaines $D(x) = \{0, 5\}$, $D(y) = \{0, 2, 5, 12\}$, $D(z) = \{4, 5\}$, et la contrainte $|x - y| = z$. BC supprimera seulement la valeur 12 de $D(y)$ parce que c'est la seule borne qui n'a pas de support aux bornes. La valeur 2 pour y n'a pas de support aux bornes mais n'est pas une borne, et la valeur 4 de z est une borne mais a des supports aux bornes, comme $((x, 5), (y, 1), (z, 4))$ par exemple. $\diamondsuit$

Règles de réduction

Une règle de réduction spécifie une condition suffisante pour supprimer des valeurs. L'algorithme de propagation itère sur les règles jusqu'à ce qu'aucun domaine ne soit modifié, c'est à dire que l'on ait atteint un point fixe. L'approche par règles de réduction est utile principalement quand on a des informations sur la sémantique de la contrainte à propager car cela permet de spécifier simplement et efficacement un moyen de la propager. Il est en effet souvent inefficace d'utiliser un algorithme générique comme AC3 quand on a des informations sur la contrainte. Cette approche est très utilisée dans les résolveurs car ils contiennent en général beaucoup de contraintes de base prédéfinies, notamment des contraintes arithmétiques. Sur ces contraintes, une modification dans un domaine n'a souvent pas le même effet sur les autres variables de la contrainte selon que c'est la suppression d'une valeur au milieu du domaine, la suppression de la valeur minimum, la suppression de la valeur maximum ou une instanciation. Les résolveurs sont donc souvent munis de la capacité à différencier ces types de réductions de domaines, appelées *événements*. Les événements reconnus par la majorité des résolveurs sont :

- `RemValue`(x_i) : quand une valeur v est supprimée de $D(x_i)$
- `IncMin`(x_i) : quand la valeur minimum de $D(x_i)$ est supprimée
- `DecMax`(x_i) : quand la valeur maximum de $D(x_i)$ est supprimée
- `Instantiate`(x_i) : quand $D(x_i)$ devient un singleton

A l'aide de ces quatre types d'événements on peut spécifier des règles de réduction pour de nombreuses contraintes simples.

Exemple 5 Soit la contrainte $\texttt{alldifferent}(x, y, z)$ avec $D(x) = D(y) = D(z) = \{1, 2, 3, 4\}$. Plutôt que d'appeler la fonction `Revise` pour propager cette contrainte, ce qui coûterait $O(d^3)$, nous pouvons construire l'ensemble de règles suivant :

R1 : **if** $\texttt{Instantiate}(x)$ **then** $D(y) \leftarrow D(y) \setminus D(x)$; $D(z) \leftarrow D(z) \setminus D(x)$
R2 : **if** $\texttt{Instantiate}(y)$ **then** $D(x) \leftarrow D(x) \setminus D(y)$; $D(z) \leftarrow D(z) \setminus D(y)$
R3 : **if** $\texttt{Instantiate}(z)$ **then** $D(x) \leftarrow D(x) \setminus D(z)$; $D(y) \leftarrow D(y) \setminus D(z)$

Appliquer ces règles est très efficace car chacune des règles s'exécute en temps constant. On remarque que la différenciation des types d'événements permet de ne réveiller une règle sur une contrainte que si elle risque d'entraîner d'autres suppressions. Par exemple, si 1 et 3 sont supprimés de $D(z)$, seulement $\texttt{RemValue}(z)$ et $\texttt{IncMin}(z)$ sont vrais, donc aucune règle de la contrainte $\texttt{alldifferent}(x, y, z)$ n'a besoin d'être réveillée. Si 1,2 et 3 sont supprimés de $D(z)$, $\texttt{Instantiate}(z)$ est vrai. Donc la règle **R3** est appliquée.

Notons que l'ensemble de règles **R1**, **R2**, **R3** n'est pas suffisant pour garantir l'arc consistance. Supposons que 3 et 4 soient supprimés de $D(x)$ et $D(y)$ alors que $D(z) = \{1, 2, 3, 4\}$. Les règles **R1**, **R2**, **R3** ne suppriment aucune valeur alors que 1 et 2 dans $D(z)$ ne sont pas arc consistantes. ♢

0.5.2 Consistances fortes

L'arc consistance n'est pas le seul moyen de détecter des inconsistances dans un réseau. Dès les années 70, plusieurs auteurs ont proposé des propriétés qui permettent de découvrir plus d'inconsistances que l'arc consistance. Freuder (1978) a proposé les k-consistances, toute une classe de consistances locales plus fortes que l'arc consistance.

Définition 7 (k-consistance) *Soit $P = (X, D, C)$ un réseau.*

- *Etant donné un ensemble de variables $Y \subseteq X$ avec $|Y| = k - 1$, une instanciation localement consistante I sur Y est k-consistante si et seulement si pour toute k^{eme} variable $x_{i_k} \in X \setminus Y$ il existe une valeur $v_{i_k} \in D(x_{i_k})$ telle que $I \cup \{(x_{i_k}, v_{i_k})\}$ est localement consistante.*
- *Le réseau P est* k-consistant *si et seulement si pour tout ensemble Y de $k - 1$ variables, toute instanciation localement consistante sur Y est k-consistante.*

Avant que Freuder ne propose la k-consistance, Montanari (1974) avait proposé la chemin consistance. Elle était définie pour les réseaux de contraintes binaires dans lesquels chaque paire de variables est impliquée dans au plus une contrainte. Sur de tels réseaux, Mackworth (1977) a montré que la chemin consistance est équivalente à la 3-consistance.

Exemple 6 Soit le réseau P avec les variables x_1, x_2, x_3, les domaines $D(x_1) = D(x_2) = D(x_3) = \{1, 2\}$, et $C = \{x_1 \neq x_2, x_2 \neq x_3\}$. P n'est pas chemin/3-consistant parce que ni $((x_1, 1), (x_3, 2))$, ni $((x_1, 2), (x_3, 1))$ ne peuvent s'étendre à une valeur de x_2 satisfaisant à la fois c_{12} et c_{23}. Le réseau $P' = (X, D, C \cup \{x_1 = x_3\})$ est chemin/3-consistant. ♢

Freuder a aussi défini la k-consistance forte. Cette consistance locale garantit que le réseau est j-consistant pour $1 \leq j \leq k$.

Définition 8 (k-consistance forte) *Un réseau est* fortement k-consistant *si et seulement si il est j-consistant pour tout $j \leq k$.*

Le maximum de réduction que l'on peut appliquer sur un réseau de contraintes sans perdre de solutions est d'atteindre un réseau sur lequel *toutes* les instanciations localement consistantes peuvent s'étendre à une solution. La n-consistance forte garantit cela, avec $|X| = n$.

Freuder (1985) a aussi proposé les (i, j)-consistances, des consistances locales basées sur les variables. La (i, j)-consistance garantit que toute instanciation de taille i localement consistante peut être étendue à n'importe quelles j variables supplémentaires.

Janssen et al. (1989) et al. ont proposé la première consistance locale basée sur le nombre de contraintes impliquées et non pas sur le nombre de variables.

Définition 9 (Paire consistance) *Etant donné un réseau $P = (X, D, C)$, une paire de contraintes c_1 et c_2 dans C est paire consistante si et seulement si toute instanciation sur $X(c_1)$ (resp. sur $X(c_2)$) satisfaisant c_1 (resp. c_2) peut être étendu en une instanciation sur $X(c_1) \cup X(c_2)$ satisfaisant c_2 (resp. c_1). P est* paire consistant *si et seulement si toute paire de contraintes dans C est paire consistante.*

Exemple 7 Soit le réseau avec les variables x_1, x_2, x_3, x_4, les domaines $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2\}$ et les contraintes $c_1(x_1, x_2, x_3) = \{(121), (211), (222)\}$ et $c_2(x_2, x_3, x_4) = \{(111), (222)\}$. Ce réseau est arc consistant. Cependant il n'est pas paire consistant parce que le tuple (121) de c_1 n'est compatible avec aucun tuple dans c_2. $\diamondsuit$

D'autres auteurs ont proposé des consistances locales, qui sont paramétrées par le nombre de contraintes impliquées dans le raisonnement. Jégou (1993) a proposé l'hyper k-consistance, qui peut être vue comme le dual de la k-consistance basée sur les contraintes. Dechter et van Beek (1997) ont proposé la famille des consistances (i, m)-*relationnelles*. une forme de consistance qui se concentre sur les inconsistances à l'intérieur des portées des contraintes existantes.

Les consistances citées ci-dessus ont en commun un défaut majeur pour une intégration facile dans un résolveur. Elles créent de nouvelles contraintes qui n'étaient pas dans le réseau initial ou modifient des contraintes existantes (voir exemples 6 et 7). Dans les deux cas, ces contraintes sont coûteuses à stocker et à propager.

Certaines consistances locales permettent de supprimer plus de valeurs que l'arc consistance tout en laissant les contraintes inchangées. La *singleton arc consistance (SAC)* proposée par Debruyne et Bessiere (2001) est la plus connue, à la fois parce qu'elle permet un bon compromis entre niveau de propagation et coût en temps, et parce qu'elle peut s'intégrer assez facilement dans l'architecture des résolveurs actuels.

Définition 10 (Singleton arc consistency) *Un réseau $P = (X, D, C)$ est* singleton arc consistant *si et seulement si pour tout $x_i \in X$, pour tout $v_i \in D(x_i)$, le sous-problème $P|_{x_i=v_i}$, où le domaine de x_i dans P a été réduit au singleton $\{v_i\}$, peut être rendu arc consistant sans vider de domaine.*

Appliquer la singleton arc consistance revient donc à essayer à tour de rôle toutes les instanciations $x_i = v_i$ de taille 1 et à tester si elles conduisent à un échec quand on propage l'arc consistance sur le réseau obtenu. Un échec veut dire que l'instanciation $x_i = v_i$ n'appartient à aucune solution et donc que v_i peut être supprimée du domaine de x_i. Une approche de ce type limitée au test des bornes d'intervalles continus en ne propageant que la borne consistance au lieu de l'arc consistance a d'abord été utilisée sur les CSP numériques sous le nom de 3B-consistance (Lhomme, 1993) puis en ordonnancement sous le nom de 'shaving'.

0.6 Cas polynomiaux

Le CSP étant NP-complet, il est naturel de se poser la question de l'existence de classes particulières de réseaux de contraintes pour lesquelles CSP deviendrait polynomial.

Les premiers travaux dans ce sens ont été conduits par Freuder et utilisent la *structure* du réseau de contraintes, que l'on représente par son *graphe associé*. Le graphe associé à un réseau de contraintes $P = (X, D, C)$ est le graphe $G_P = (X, E)$ qui a un sommet par variable de P et une arête (x_i, x_j) dans E si et seulement si il existe une contrainte $c \in C$ avec $x_i \in X(c), x_j \in X(c)$. Freuder (1982) définit la *largeur*[1] du graphe associé à un réseau de contraintes. La largeur est l'entier k le plus petit tel qu'une procédure gloutonne supprimant un sommet chaque fois qu'il est de degré inférieur ou égal à k pourra supprimer tous les sommets du graphe. L'ordre inverse de l'ordre dans lequel la procédure a supprimé les sommets est un ordre de largeur k.

Théorème 2 ((Freuder, 1982)) *Si un réseau de contraintes P a un graphe associé de largeur k et est fortement $(k+1)$-consistant, alors appliquer la procédure BT sur un ordre de ses variables de largeur k est sans retour-arrière.*

Dans le même article, Freuder dénonce les limites de ce théorème. En effet, appliquer la $(k+1)$-consistance avec $k > 1$ crée de nouvelles contraintes et donc modifie la largeur du graphe associé au réseau, ce qui invalide la précondition du théorème. Freuder caractérise le cas où la structure n'est pas modifiée.

Corollaire 1 ((Freuder, 1982)) *Un réseau ayant un graphe associé de largeur 1 (arbre) peut être résolu en $O(nd^2)$ (arc consistance puis BT sans retour arrière).*

De nombreux travaux ont essayé d'étendre les résultats de Freuder à des structures de réseaux de contraintes plus générales que les arbres. L'idée sous-jacente est que si la *tree-width* du graphe associé est bornée, alors le réseau est polynomial à résoudre (Freuder, 1990).

Mais il y a une autre approche à la recherche de classes traitables, c'est celle qui au lieu de poser des restrictions sur la structure du réseau, pose les restrictions sur le type de contraintes utilisées dans le réseau. Cooper a proposé les contraintes ZOA, un des premiers exemples de classe traitable de contraintes. Une contrainte binaire c_j avec $X(c) = (x_i, x_j)$ est ZOA (pour zero, one, all) si et seulement si chaque valeur $v_i \in D(x_i)$ a soit aucun, soit un, soit $|D(x_j)|$ supports dans $D(x_j)$.

Théorème 3 ((Cooper et al., 1994)) *Si toutes les contraintes d'un réseau de contraintes binaires sont ZOA, alors la chemin consistance est suffisante pour que la procédure BT soit sans retour-arrière.*

De nombreuses autres classes de contraintes ont été proposées, qui offrent la garantie que le réseau est polynomial à résoudre. On peut citer les *connected row convex* (van Beek and Dechter, 1995; Deville et al., 1999) ou les nombreux travaux de Cohen et al. décrits dans (Cohen and Jeavons, 2006).

[1] La largeur de Freuder est une borne supérieure à la *tree-width* d'un graphe (Arnborg, 1985).

0.7 Synthèse de solutions et décompositions

Dechter et Pearl (1988) ont proposé *Adaptive consistency (AdC)*, une technique qui permet d'utiliser le théorème 2 tout en tenant compte des limites soulevées par Freuder (voir ci-dessus). Plutôt que d'appliquer uniformément la k-consistance à tout le réseau, on n'y applique qu'une forme réduite, unidirectionnelle et adaptée au nombre de voisins de la variable traitée. Soit un ordre total $<_o$ sur les variables. Notons $parents(x_i)$ l'ensemble des parents de x_i dans le graphe associé G_P. Les variables sont rendues AdC une par une en partant de la dernière selon $<_o$ jusqu'à la première. Une variable x_i est rendue AdC en créant une contrainte de portée $parents(x_i)$ qui interdira toute instanciation localement consistante de $parents(x_i)$ qui ne s'étend pas en une instanciation localement consistante sur $parents(x_i) \cup \{x_i\}$. L'ensemble des arêtes de G_P est alors augmenté de toutes les arêtes possibles sur $parents(x_i)$. Si aucune des contraintes générées par AdC n'est la contrainte vide, la procédure BT appliquée selon l'ordre $<_o$ trouvera une solution sans retour-arrière. Le coût de AdC est en $O(nd^{w+1})$ où w est la largeur induite de l'ordre $<_o$, c'est à dire la largeur de $<_o$ après application de AdC.

AdC nécessite d'avoir fixé un ordre avant de lancer la phase de consistance locale. Si l'ordre est mauvais (largeur induite importante), le processus sera coûteux. Dechter et Pearl (1989) proposent le *tree clustering*, une technique de décomposition qui transforme un réseau quelconque en réseau structuré en arbre. L'idée est de regrouper les variables en paquets de sorte que si l'on considère chaque paquet comme une méta-variable reliée aux méta-variables dont les variables partagent des contraintes avec les siennes, le réseau résultant est structuré en arbre. Toutes les instanciations consistantes sur chaque paquet sont calculées et considérées comme les valeurs de la méta-variable représentant le paquet. L'avantage par rapport à AdC est qu'aucun ordre n'est fourni à l'avance, mais le calcul des instanciations consistantes de chaque paquet peut s'avérer très coûteux en espace. De nombreuses autres techniques de décomposition ont été proposées (Gottlob et al., 2000; Jégou and Terrioux, 2003).

On peut aussi utiliser non pas la structure du réseau de contraintes mais la structure de ses solutions. Amilhastre et al. (2002) associent un automate d'états finis à chaque contrainte et les combinent jusqu'à obtenir un automate représentant toutes les solutions du réseau. Si les solutions ne sont pas trop dispersées, l'automate peut être de taille raisonnable et de nombreuses requêtes, habituellement coûteuses, se font alors en temps linéaire ou constant.

0.8 Améliorer le backtrack chronologique

Les techniques de synthèse ou de décomposition ne fonctionnent que dans des cas très particuliers où le réseau a une structure adéquate (par exemple proche d'un arbre pour le tree clustering). La méthode standard de résolution de CSP reste donc la procédure BT, et de nombreux travaux ont consisté à apporter des améliorations à ce backtrack pour le rendre moins inefficace. Suivant Dechter et Pearl (1988), on peut grossièrement classer les types d'amélioration en quatre catégories indépendantes qui peuvent se combiner.

Les méthodes rétrospectives tirent parti des informations implicites contenues dans un échec pour éviter plus tard des branches inutiles. Les *backjumping* sélectionnent un meilleur point de retour arrière (Gaschnig, 1979; Dechter, 1990; Prosser, 1993). Le *nogood recording* mémorise des instanciations menant à coup sûr à un échec (Dechter, 1990; Schiex and Verfaillie, 1993). Je ne vais pas décrire ces méthodes par manque de place et parce qu'elle sont

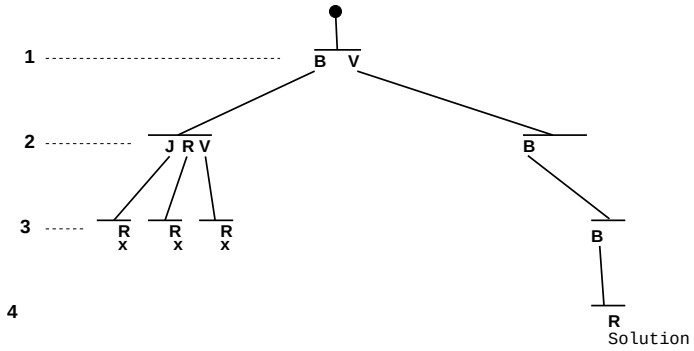

FIG. 5: Arbre de recherche de forward-checking sur le réseau de contraintes de l'exemple 1.

peu ou pas utilisées dans la génération actuelle de résolveurs à contraintes. On pourra se reporter au chapitre 22 (Logique propositionnelle et ses algorithmes) de ce livre pour de plus amples descriptions sur ces notions.

Les méthodes prospectives appliquent un certain niveau de propagation de contraintes à chaque noeud de l'arbre de recherche pour éliminer le plus tôt possible le maximum de branches infructueuses.

Enfin, les heuristiques d'ordonnancement des variables et des valeurs modifient l'ordre dans lequel les variables et les valeurs sont instanciées, ce qui a une incidence sur la taille de l'espace à explorer pour trouver une solution.

0.8.1 Méthodes prospectives

Les méthodes prospectives appliquent de la propagation de contraintes à chaque noeud de l'arbre de recherche exploré par la procédure BT. La motivation est qu'il vaut mieux découvrir une inconsistance une fois en haut de l'arbre de recherche qu'un nombre exponentiel de fois en bas de l'arbre. Les niveaux de propagation utilisés se contentent en général de supprimer des valeurs dans les domaines. Des niveaux de consistance plus élevés sont en effet considérés trop coûteux à maintenir à chaque noeud de l'arbre.

Le premier algorithme de cette catégorie est le *forward-checking*, esquissé par Golomb et Baumert (1965) et formellement décrit et étudié par Haralick et Elliott (1980) pour les contraintes binaires. Forward-checking est une procédure d'exploration d'instanciations partielles, tout comme BT. La différence est qu'après chaque instanciation d'une variable x_i, forward-checking supprime des domaines des variables x_k non encore instanciées et reliées à x_i par une contrainte c toutes les valeurs qui violent c. Dès qu'une de ces variables x_k a son domaine vide, la branche courante est coupée et les domaines restaurés comme avant l'instanciation de x_i. Le fait de supprimer des variables futures toutes les valeurs incompatibles avec les variables déjà instanciées rend inutile la vérification de la compatibilité des valeurs de x_i avec les variables déjà instanciées (voir ligne 4 dans l'algorithme 1). La figure 5 représente l'arbre de recherche développé par forward-checking sur le problème de l'exemple 1, à comparer à l'arbre développé par BT en figure 3.

Algorithm 3: Maintaining arc consistency (MAC)

function MAC**(in** P : réseau**) : Boolean** ;
1 appliquer AC sur P;
2 **if** P *contient un domaine vide* **then** return **false**;
3 **if** P *est complètement instancié* **then** imprimer D ; return **true**;
4 choisir une variable x_i non instanciée et une valeur v_i dans $D(x_i)$;
5 **if** *MAC*$(P|_{x_i=v_i})$ **then** return **true**;
6 return MAC$(P|_{x_i \neq v_i})$;

La question de la quantité de propagation à appliquer à chaque noeud de l'arbre de recherche a toujours fait débat. Plus de propagation signifie plus de travail à chaque noeud, mais moins de noeuds à explorer. Forward-checking a longtemps été considéré comme appliquant le bon niveau de propagation. On peut expliquer cela par le fait que les problèmes sur lesquels on testait les algorithmes dans les années 80/début 90 étaient souvent petits et pas très difficiles. Depuis le milieu des années 90, il est entendu que le niveau standard à appliquer pour résoudre des problèmes difficiles est l'arc consistance, et c'est ce que font majoritairement les résolveurs sous la forme de la procédure *MAC (Maintaining Arc Consistency)*, proposée par Sabin et Freuder (1994). MAC est un algorithme qui non seulement applique l'arc consistance après chaque instanciation, mais aussi après avoir réfuté le choix d'une valeur.

MAC est présenté dans l'algorithme 3. MAC applique d'abord l'arc consistance sur le problème donné en entrée (ligne 1). Puis, si aucun domaine n'a été vidé (ligne 2) et si on n'a pas encore atteint une solution (ligne 3), il sélectionne une variable x_i et une valeur v_i pour cette variable selon l'heuristique utilisée (ligne 4). L'appel récursif de la ligne 5 propagera le choix $x_i = v_i$ en appliquant l'arc consistance sur le réseau $P|_{x_i=v_i}$ où x_i a été instancié à v_i, avant de continuer la recherche dans ce sous-problème. Si ce choix se termine par un échec, l'appel récursif de la ligne 6 propagera la réfutation du choix précédent en appliquant l'arc consistance sur le réseau $P|_{x_i \neq v_i}$ avant de continuer la recherche. Lecoutre et Hemery (2007) ont proposé AC3rm, un algorithme très efficace pour maintenir l'arc consistance pendant la recherche. Cet algorithme stocke des supports comme AC2001, utilise la multidirectionnalité des supports comme AC7 (Bessiere et al., 1999), mais ne les restaure pas lors des retour-arrières. Il perd donc l'optimalité d'AC2001 sur un appel, mais économise le coût des restaurations, ce qui est au total est bénéfique. La plupart des résolveurs actuels sont basés sur ce principe.

0.8.2 Heuristiques

Jusqu'à maintenant nous avons considéré que la procédure de recherche de solutions sélectionnait les variables à instancier et les valeurs à leur donner dans l'ordre lexicographique. Cependant, rien ne nous oblige à respecter cet ordre arbitraire. Et il s'avère qu'en pratique, l'ordre de parcours peut avoir un impact énorme sur le temps de résolution.

Parmi les heuristiques d'ordonnancement des variables on peut distinguer les ordonnancements *statiques* et les ordonnancements *dynamiques*. Les ordonnancements statiques sont calculés avant la recherche de solutions et sont gardés pendant toute la recherche. Ils sont donc en général basés sur des critères de structure du réseau, puisque celle-ci ne change pas pendant la recherche. Par exemple, *maxdegree* ordonne les variables par ordre décroissant du nombre de contraintes portant sur elles. Cela favorise la détection d'échecs haut dans l'arbre

de recherche, ce qui coûte moins cher que de les détecter au fond de l'arbre. Les ordonnancements dynamiques vont tenir compte des modifications apparues dans le réseau au cours de la recherche, c'est à dire en général uniquement des suppressions de valeurs des domaines. Haralick et Elliott (1980) ont proposé l'heuristique *mindom*, qui prend toujours comme prochaine variable celle qui a le plus petit nombre de valeurs restant dans son domaine. L'idée est ici de minimiser le degré de branchement dans l'arbre, et non pas d'aller vers l'échec d'abord. Bessiere et Régin (1996) ont proposé *dom/deg*, qui combine les informations tirées de la taille des domaines avec des informations sur la structure du réseau. *dom/deg* sélectionne en priorité la variable qui a le plus petit ratio taille du domaine courant sur nombre de contraintes portant sur elle.

Boussemart et al. (2004) ont proposé de rendre encore plus dynamiques les heuristiques de choix de variable. Ils proposent *dom/wdeg*, une technique inspirée des résolveurs SAT qui non seulement se base sur l'état courant du réseau pendant la recherche, mais aussi sur les calculs que la procédure de recherche a déjà faits. Plus précisément, *dom/wdeg* mémorise par un compteur w_j le nombre de fois qu'une contrainte c_j a été la cause d'un échec, c'est à dire le nombre de fois où c'est la propagation de c_j qui a causé un domaine vide. Au départ, $w_j = 1$ pour toutes les contraintes. Pendant la recherche, chaque fois que `Revise`$(-, c_j)$ vide un domaine, w_j est incrémenté de 1. L'heuristique choisit la variable x_i qui minimise le ratio taille du domaine courant sur somme des w_j, où c_j porte sur x_i. On voit que cette heuristique démarre comme *dom/deg* mais va au cours de la recherche se focaliser de plus en plus sur les variables impliquant des contraintes difficiles à satisfaire.

Finalement, je citerai le très récent *last conflict reasoning* qui mime un backjumping simplement en modifiant le choix de la prochaine variable à instancier (Lecoutre et al., 2006).

Ordonnancement des valeurs

L'ordonnancement des valeurs a suscité beaucoup moins de travaux que l'ordonnancement des variables car l'effet sur les performances est bien moindre. Une des raisons est que dès qu'un problème est difficile, on a de grande chances de passer la plus grosse partie du temps sur de gros sous-problèmes inconsistants sur lesquels l'ordre de choix des valeurs a peu d'effet.

0.9 Symétries

Lors de la modélisation d'un problème en CSP il arrive souvent que l'on définisse plusieurs variables pour représenter plusieurs occurrences d'une même entité. Par exemple deux variables x_i et x_j peuvent représenter les deux cours d'histoire qu'une classe doit avoir dans la semaine. Que dans une instanciation x_i prenne la valeur Lundi-8h et x_j la valeur Jeudi-9h ou l'inverse ne changera rien au fait que l'instanciation soit solution ou pas. On dit alors que x_i et x_j sont symétriques. La même chose peut se produire pour les valeurs. Les méthodes de détection de symétries ont pour but d'éviter que la procédure de recherche explore deux sous-arbres qui sont symétriques. Ces méthodes vont en général couper des branches de l'arbre qui contiennent peut-être des solutions mais dont on est sûr qu'il existe une solution symétrique hors des branches coupées. Freuder (1991) a soulevé le problème de la symétrie des valeurs. De nombreux travaux ont depuis traité de toutes sortes de symétries (Benhamou, 1994; Gent et al., 2006).

0.10 Contraintes globales

Quand on modélise des problèmes réels en réseaux de contraintes, on s'aperçoit que certains 'motifs' se retrouvent. Par exemple, on a souvent besoin d'exprimer le fait que toutes les variables d'un ensemble doivent prendre des valeurs différentes. La taille du motif n'est pas la même dans tous les problèmes. La contrainte `alldifferent`, qui exprime ce motif, peut impliquer un nombre quelconque de variables. Elle représente donc toute une classe de contraintes. Toute contrainte spécifiant que ses variables doivent prendre des valeurs différentes, quel que soit le nombre de variables, est une contrainte `alldifferent`. On appelle contraintes globales ces classes de contraintes définies par une fonction booléenne qui peut prendre n'importe quel nombre de variables en paramètre. Beldiceanu et al. (2005) ont construit un catalogue recensant plus de 200 contraintes globales.

Exemple 8 La contrainte globale `alldifferent`$(x_1, \ldots, x_n)$ est la classe de contraintes qui sont définies sur n'importe quelle séquence de n variables, $n \geq 2$, telles que $x_i \neq x_j$ pour tout $i, j, 1 \leq i, j \leq n, i \neq j$. `NValue`$(y, x_1, \ldots, x_n)$ est la classe de toutes les contraintes définies sur une séquence de $n+1$ variables, $n \geq 1$, telles que $|\{x_i \mid 1 \leq i \leq n\}| = y$. $\diamondsuit$

Si une contrainte globale est disponible dans un résolveur, l'utilisateur peut exprimer facilement le motif associé, qui autrement pourrait être complexe à exprimer. Ces contraintes pouvant porter sur de grands nombres de variables, il est important d'avoir un moyen de les propager autre que leur appliquer un algorithme générique d'arc consistance comme AC3. Rappelons que les algorithmes génériques optimaux d'arc consistance sont en $O(erd^r)$ pour des contraintes portant sur r variables (voir Section 0.5.1.)

Une première solution pour propager une contrainte globale à moindre coût est de la décomposer en contraintes plus simples. Une décomposition d'une contrainte globale G est une transformation *polynomiale* δ_k (k étant un entier) qui pour tout réseau $P = (X(c), D, \{c\})$ où c est la contrainte de la classe G d'arité $|X(c)|$, retourne un réseau $\delta_k(P) = (X_{\delta_k(P)}, D_{\delta_k(P)}, C_{\delta_k(P)})$ tel que $X(c) \subseteq X_{\delta_k(P)}$, $D(x_i) = D_{\delta_k(P)}(x_i)$ pour tout $x_i \in X(c)$, $|X(c_j)| \leq k$ pour tout $c_j \in C_{\delta_k(P)}$, et $sol(P)$ est égal à la projection de $sol(\delta_k(P))$ sur $X(c)$. Transformer P en $\delta_k(P)$ veut dire remplacer c par des contraintes d'arité bornée (et de nouvelles variables si besoin) tout en préservant l'ensemble de tuples autorisés sur $X(c)$. Par définition, les domaines des variables additionnelles sont de taille polynomiale.

Exemple 9 La contrainte globale `atleast`$_{p,v}(x_1, \ldots, x_n)$ est satisfaite si et seulement si au plus p variables dans $x_1, \ldots, x_n$ valent v. Cette contrainte peut être décomposée avec $n+1$ variables additionnelles $y_0, \ldots, y_n$. La transformation contient la contrainte ternaire $(x_i = v \ \wedge \ y_i = y_{i-1} + 1) \vee (x_i \neq v \ \wedge \ y_i = y_{i-1})$ pour tout $i, 1 \leq i \leq n$, et les domaines $D(y_0) = \{0\}, D(y_n) = \{p, \ldots, n\}$ et $D(y_i) = \{0, \ldots, n\}$ pour tout $i, 1 \leq i \leq n$. $\diamondsuit$

Toute la question est bien sûr de trouver des décompositions qui *préservent* l'arc consistance sur la contrainte d'origine. C'est à dire, pour n'importe quelle instance c de la contrainte globale G sur n'importe quel domaine initial D sur $X(c)$, on veut que pour tout domaine D' inclus dans D, l'arc consistance appliquée sur la décomposition supprime de D' les mêmes valeurs que l'arc consistance sur c. La décomposition de `atleast`$_{p,v}$ donnée dans l'exemple 9 préserve l'arc consistance parce qu'elle a une structure Berge-acyclique.

Il n'est pas toujours possible de trouver une décomposition qui préserve l'arc consistance. Par exemple, Bessiere et al. (2007) ont montré que si le problème de l'existence d'un tuple valide satisfaisant la contrainte est NP-complet, alors il n'existe pas de décomposition préservant l'arc consistance, à moins que P=NP. C'est le cas de la contrainte globale `NValue`, pour laquelle il est donc inutile de chercher une décomposition préservant l'arc consistance. Bessiere et al. (2009) ont montré que les contraintes NP-difficiles ne sont pas les seules pour lesquelles il n'existe pas de décomposition préservant l'arc consistance. Toute contrainte globale qui peut représenter une fonction non représentable par un circuit booléen monotone de taille polynomiale n'admet pas de décomposition préservant l'arc consistance. Un exemple est la contrainte `alldifferent`. Pour de telles contraintes, la solution est d'implémenter un algorithme spécifique qui calcule l'arc consistance en temps polynomial. Par exemple, Régin (1994) a utilisé le problème du couplage maximum dans un graphe biparti pour produire un algorithme qui calcule l'arc consistance sur la contrainte `alldifferent`.

0.11 Extensions au modèle CSP

Dans ce chapitre nous avons présenté diverses techniques pour résoudre le CSP. Avec l'utilisation du raisonnement par contraintes pour résoudre des problèmes réels on s'est vite aperçu que répondre oui ou non ou juste donner une solution à un réseau spécifié une fois pour toutes n'est souvent pas suffisant pour satisfaire l'utilisateur. L'utilisateur peut très bien avoir oublié de spécifier certaines contraintes qui lui viennent à l'esprit en voyant une solution qui a des défauts (par exemple un emploi du temps avec quatre heures de maths d'affilée) ou au contraire avoir imposé trop de préférences qu'il faudra relâcher pour rendre le réseau satisfiable (par exemple enlever la contrainte de fin des cours à 16h pour les classes ayant plus de 32h hebdomadaires). Une première approche pour traiter ce problème a été les CSP dynamiques, où l'utilisateur interagit avec le résolveur en ajoutant ou enlevant des contraintes selon la réponse qu'il lui a fournie. Les modèles *maxCSP* et CSP valués peuvent aussi traiter ce problème puisqu'ils cherchent des solutions optimales selon certains critères. Ils seront développés au chapitre 24 (Contraintes valuées). Le problème à décrire peut aussi contenir des incertitudes sur les valeurs de certaines variables dont l'utilisateur n'est pas maître (par exemple liées à la météo). Une solution devra satisfaire les contraintes quelles que soient les valeurs prises par ces variables, ou devra maximiser une probabilité de satisfaction. Les CSP quantifiés ou les CSP stochastiques traitent ce cas. Le problème peut être distribué sur plusieurs sites distants communicant par messages et voulant garder privées certaines données du problème. On utilise alors les CSP distribués. Certains problèmes manipulent le concept d'ensemble. Représenter des ensembles par un ensemble de variables booléennes représentant leur fonction caractéristique est une solution parfois lourde et peu efficace. Des travaux se sont intéressés à leur représentation par des variables spéciales, les variables ensemblistes. On peut aussi avoir à représenter des informations qui ne se discrétisent pas facilement. Dans ce cas, on utilise les CSP numériques, dans lesquels les variables prennent leurs valeurs dans des domaines composés d'intervalles sur les réels. Finalement, sur certaines problèmes, on peut ne pas avoir toutes les valeurs des domaines dès le début de la résolution, ou bien la portée d'une contrainte va dépendre de la valeur d'autres variables. Les *open CSP* permettent de traiter ces cas. Il existe bien d'autres extensions qui sont plus marginales mais pourraient prendre de l'importance si leur intérêt se confirme.

Bibliographie

Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency restoration and explanations in dynamic csps -application to configuration. *Artificial Intelligence*, 135 :199–234.

Arnborg, S. (1985). Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25 :2–23.

Beldiceanu, N., Carlsson, M., and Rampon, J. (2005). Global constraint catalog. Technical Report T2005 :08, Swedish Institute of Computer Science, Kista, Sweden.

Benhamou, B. (1994). Study of symmetry in constraint satisfaction problems. In *Proceedings PPCP'94*, pages 249–257, Seattle WA.

Bessiere, C., Freuder, E., and Régin, J. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148.

Bessiere, C., Hebrard, E., Hnich, B., and Walsh, T. (2007). The complexity of reasoning with global constraints. *Constraints*, 12(2) :239–259.

Bessiere, C., Katsirelos, G., Narodytska, N., and Walsh, T. (2009). Circuit complexity and decompositions of global constraints. In *Proceedings IJCAI'09*, pages 412–418, Pasadena CA.

Bessiere, C. and Régin, J. (1996). MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings CP'96*, pages 61–75, Cambridge MA.

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

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings ECAI'04*, pages 146–150, Valencia, Spain.

Cohen, D. and Jeavons, P. (2006). The complexity of constraint languages. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 6. Elsevier.

Cooper, M., Cohen, D., and Jeavons, P. (1994). Characterising tractable constraints. *Artificial Intelligence*, 65 :347–361.

Debruyne, R. and Bessiere, C. (2001). Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230.

Dechter, R. (1990). Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41 :273–312.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34 :1–38.

Dechter, R. and Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, 38 :353–366.

Dechter, R. and van Beek, P. (1997). Local and global relational consistency. *Theoretical Computer Science*, 173(1) :283–308.

Deville, Y., Barette, O., and Van Hentenryck, P. (1999). Constraint satisfaction over connected row convex constraints. *Artificial Intelligence*, 109(1-2) :243–271.

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

Freuder, E. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32.

Freuder, E. (1985). A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4) :755–761.

Freuder, E. (1990). Complexity of k-tree structured constraint satisfaction problems. In *Proceedings AAAI'90*, pages 4–9, Boston MA.

Freuder, E. (1991). Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings AAAI'91*, pages 227–233, Anaheim CA.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical Report Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh PA.

Gent, I., Petrie, K., and Puget, J. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 10. Elsevier.

Golomb, S. and Baumert, L. (1965). Backtrack programming. *Journal of the ACM*, 12(4) :516–524.

Gottlob, G., Leone, N., and Scarcello, F. (2000). A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2) :243–282.

Haralick, R. and Elliott, G. (1980). Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313.

Janssen, P., Jégou, P., Nouguier, B., and Vilarem, M. C. (1989). A filtering process for general constraint-satisfaction problems : Achieving pairewise-consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, Fairfax VA.

Jégou, P. (1993). On the consistency of general constraint-satisfaction problems. In *Proceedings AAAI'93*, pages 114–119, Washington D.C.

Jégou, P. and Terrioux, C. (2003). Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75.

Laurière, J. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10 :29–127.

Lecoutre, C. and Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings IJCAI'07*, pages 125–130, Hyderabad, India.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2006). Last conflict based reasoning. In *Proceedings ECAI'06*, pages 133–137, Riva del Garda, Italy.

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

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

Mohr, R. and Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233.

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

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299.

Régin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA.

Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA.

Schiex, T. and Verfaillie, G. (1993). Nogood recording for static and dynamic csps. In *Proceedings IEEE ICTAI'93*, pages 48–55, Boston MA.

van Beek, P. and Dechter, R. (1995). On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3) :543–561.