Des symétries locales de variables aux symétries globales

Christophe Lecoutre, Sébastien Tabary

▶ **To cite this version:**

Christophe Lecoutre, Sébastien Tabary. Des symétries locales de variables aux symétries globales. JFPC 2008- Quatrièmes Journées Francophones de Programmation par Contraintes, LINA - Université de Nantes - Ecole des Mines de Nantes, Jun 2008, Nantes, France. pp.181-190. inria-00291567

HAL Id: inria-00291567

https://inria.hal.science/inria-00291567

Submitted on 27 Jun 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Des symétries locales de variables aux symétries globales

Christophe Lecoutre Sébastien Tabary

CRIL – CNRS UMR 8188,
Université Lille-Nord de France, Artois
rue de l'université, SP 16, F-62307 Lens
{lecoutre,tabary}@cril.fr

Résumé

Dans cet article, nous proposons de détecter automatiquement les symétries de variables pour les instances CSP en calculant au préalable pour chaque contrainte une partition mettant en valeur les variables dites localement symétriques. A partir de cette information qui peut être obtenue en temps polynomial, nous pouvons alors construire un graphe (appelé lsv-graphe) dont les automorphismes correspondent aux symétries de variables (globales). De manière intéressante, notre approche permet de nous abstraire de la représentation (extension, intention, globale) des contraintes, tandis que la taille des lsv-graphes reste linéaire en fonction de la somme des arités des contraintes. Pour éliminer les symétries de variables, une approche classique consiste à poster des contraintes d'ordre lexicographique. Nous proposons ici un nouvel algorithme qui établit GAC sur de telles contraintes. Celui-ci est simple à implanter, adapté aux solveurs génériques tout en étant capable de gérer des variables partagées. Les résultats expérimentaux obtenus montrent la robustesse de cette approche dans son ensemble : appliquée à de nombreuses séries de problèmes, un nombre plus important d'instances sont résolues tandis que le temps observé pour l'identification (et l'exploitation) des symétries est négligeable.

1 Introduction

L'élimination de symétries [4] est un domaine de recherche particulièrement dynamique en programmation par contraintes. En écartant les parties symétriques d'un réseau de contraintes, on peut espérer obtenir une réduction drastique de l'effort de recherche nécessaire pour trouver une solution ou prouver l'incohérence. Différentes approches ont été proposées pour exploiter les symétries : on peut a) ajouter des contraintes éliminant les symétries avant la recherche, b) utiliser des heuristiques éliminant les symétries, c) éliminer les symétries durant la recherche (SBDS), d) éliminer les symétries par détection de dominance (SBDD). Cependant, pour cette ligne de recherche, les symétries sont considérées comme étant données par l'utilisateur. En conséquence, pour construire des solveurs génériques ou boites noires (i.e. des solveurs qui peuvent être facilement pris en main par des non spécialistes), une question importante nécessite une réponse : comment détecter automatiquement les symétries.

Dans [8], un système, appelé CGRASS, permet d'analyser les instances CSP de manière à automatiquement identifier des symétries ainsi que des contraintes impliquées. Au coeur du système, une comparaison syntaxique est réalisée en utilisant le calcul de formes normales. Pour détecter les variables symétriques, chaque couple de variables est considéré à tour de rôle, et l'ensemble des contraintes obtenu après la permutation de deux variables est normalisé et comparé à la forme initiale. Malheureusement, comme indiqué dans [8], cette approche est pénalisante, du fait de la complexité de calculer des formes normales sur tous les couples de variables et sur l'ensemble des contraintes.

La détection automatique de symétries a aussi été abordée en réduisant le problème à celui du calcul d'automorphismes de graphes. Dans [14], les auteurs proposent de convertir les instances CSP en instances SAT tout en capturant leur structure symétrique (avant réduction). Un graphe d'analyse est construit à partir des expressions associées aux contraintes, et certaines transformations (e.g. éliminer les parenthèses, grouper les opérateurs) sont proposées pour favoriser la détection de symétries. Les automorphismes de ce graphe sont alors calculés avec un programme tel que Saucy [6]. Dans cette veine, une méthode

de détection de symétries automatique a été proposée dans [13] : elle permet de détecter des symétries de variables et de valeurs ainsi que des symétries non triviales impliquant à la fois des variables et des valeurs.

Dans cet article, nous proposons de détecter automatiquement des symétries de variables en calculant une partition de la portée de chaque contrainte pour exhiber les variables localement symétriques. A partir de cette information locale calculée en temps polynomial, nous construisons un graphe, nommé lsv-graphe, dont les automorphismes correspondent à des symétries de variables du réseau de contraintes. De manière intéressante, notre approche permet de nous abstraire de la représentation des contraintes : que la représentation soit donnée en extension, en intention ou sous la forme d'une contrainte globale, on introduit la même notion (type de noeuds) dans le lsv-graphe. En outre, à la fois le nombre de noeuds et d'arêtes du lsv-graphe est linéaire en fonction de la somme des arités des contraintes de l'instance CSP, rendant la détection de symétries efficace lorsqu'on utilise des outils disponibles tels que Nauty [11] et Saucy [6] sur de très grandes instances.

Les générateurs du groupe de symétries retournés par le programme d'identification des automorphismes de graphes peuvent impliquer des cycles de longueur strictement plus grands que 2. Cela signifie que, lorsque des contraintes d'ordre lexicographique sont introduites pour éliminer les symétries, des variables partagées peuvent apparaître au niveau des vecteurs de variables sur lesquels sont bâties ces contraintes. Pour gérer les variables partagées, nous proposons un nouvel algorithme pour établir la cohérence d'arc généralisée (GAC) sur les contraintes d'ordre lexicographique. Cet algorithme est plus simple que celui introduit dans [10], et ne nécessite pas une gestion à grain fin des événements tout en préservant la même complexité temporelle dans le pire des cas.

Cet article est organisé comme suit. Après quelques rappels, nous décrivons une nouvelle approche pour détecter automatiquement des symétries de variables. Ensuite, nous introduisons un algorithme simple pour établir GAC sur des contraintes d'ordre lexicographique avec variables partagées. Avant de conclure, nous présentons les résultats d'une expérimentation assez poussée.

2 Rappels techniques

Un réseau de contraintes (CN pour Constraint Network) P est un couple $(\mathscr{X}, \mathscr{C})$ où $\mathscr{X}$ est un ensemble fini de n variables et $\mathscr{C}$ un ensemble fini de e contraintes. A chaque variable $X \in \mathscr{X}$ on associe un domaine, noté $dom(X)$, qui contient l'ensemble des valeurs autorisées pour X. Chaque contrainte $C \in \mathscr{C}$ implique un sous-ensemble ordonné de variables de $\mathscr{X}$ et est définie par une relation, notée $rel(C)$, qui représente l'ensemble des tuples autorisés pour ces variables. Ce sous-ensemble de variables est appelé la *portée* de C et est noté $scp(C)$. L'arité d'une contrainte est le nombre de variables de sa portée. Une contrainte binaire est d'arité 2.

Une variable est *close* ssi son domaine est singleton. Sans perte de généralité, nous considérons ici que les domaines ne contiennent que des entiers. Nous noterons $min(X)$ et $max(X)$ la plus petite et la plus grande valeur de $dom(X)$. Un couple (X, a) avec $X \in \mathscr{X}$ et $a \in dom(X)$ sera appelé une *valeur* (de P). L'ensemble des valeurs de P qui peut être construit à partir d'une contrainte C est $values(C) = \{(X, a) \mid X \in scp(C) \wedge a \in dom(X)\}$.

Même si formellement une contrainte C est définie par une relation, en pratique, elle peut être représentée en extension par une table ou en intention par une expression (représentant un prédicat) notée $pre(C)$. La taille d'une table (resp. d'une expression) correspond au nombre de ses éléments (resp. de ses tokens : opérateurs, constantes et variables).

En règle générale, dans un processus de résolution d'un CN donné, on procède en réduisant le domaine des différentes variables par élimination de valeurs incohérentes i.e. des valeurs qui ne peuvent apparaître dans aucune solution. En particulier, il est possible de filtrer les domaines en considérant certaines propriétés des CNs : la cohérence d'arc généralisée (GAC pour Generalized Arc Consistency), simplement appelée AC, lorsque les contraintes sont binaires, reste la propriété centrale.

Avant de donner une définition technique de GAC, nous introduisons la notion de support. Étant donné un ensemble ordonné $\{X_1, \ldots, X_k\}$ de k variables et un k-tuple $\tau = (a_1, \ldots, a_k)$ de valeurs, pour tout $i \in 1..k$, la valeur a_i sera notée $\tau[i]$ et aussi $\tau[X_i]$ par abus de notation. $\tau_{X_i \leftrightarrow X_j}$ représente le tuple obtenu à partir de τ en permutant $\tau[X_i]$ et $\tau[X_j]$. Si C est une contrainte k-aire alors le k-tuple τ est dit autorisé par C ssi $\tau \in rel(C)$; un tuple valide de C ssi $\forall X \in scp(C)$, $\tau[X] \in dom(X)$; un support de C ssi τ est un tuple valide de C et autorisé par C. Un tuple τ est un support pour une valeur (X, a) dans C ssi $X \in scp(C)$ et τ est un support de C tel que $\tau[X] = a$. Déterminer si un tuple est autorisé est appelé un *test de cohérence*.

Définition 1. *Soit P un CN.*

- *Une valeur (X, a) de P est GAC ssi pour chaque contrainte C impliquant X, il existe un support pour (X, a) dans C ;*
- *une variable X de P est GAC ssi $dom(X) \neq \emptyset$ et $\forall a \in dom(X)$, (X, a) est GAC ;*
- *P est GAC ssi chaque variable de P est GAC.*

En satisfaction de contraintes, de nombreuses définitions (voir [4, 2]) de symétries ont été proposées. Une symétrie peut être perçue comme une permutation (bijection d'un ensemble sur lui-même) qui, lorsqu'elle est appliquée, laisse inchangée un objet donné (par exemple, un graphe).

Dans cet article, nous restreignons notre attention aux symétries de variables définies comme suit :

Définition 2. *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN avec $\mathscr{X} = \{X_1, \ldots, X_n\}$. Une symétrie de variables σ de P est une bijection de $\mathscr{X}$ vers $\mathscr{X}$ telle que $\{X_1 = a_1, \ldots, X_n = a_n\}$ est une solution de P ssi $\{\sigma(X_1) = a_1, \ldots, \sigma(X_n) = a_n\}$ est une solution de P.*

L'ensemble des symétries (de variables) d'un CN forme un groupe. L'inverse d'une symétrie ainsi que la composition de deux symétries est donc une symétrie. Il est possible de condenser la représentation de l'ensemble des symétries par un sous-ensemble d'entre elles appelées générateurs. Toute symétrie peut être obtenue par composition à partir des générateurs. Pour finir, toute symétrie peut être décrite par un ensemble de cycles. Par exemple, $\sigma = \{(X_1, X_3), ((X_2, X_4, X_5)\}$ contient deux cycles et représente $\sigma(X_1) = X_3, \sigma(X_2) = X_4, \sigma(X_3) = X_1, \sigma(X_4) = X_5, \sigma(X_5) = X_2$. Pour plus d'informations, voir [9].

3 Détecter les symétries de variables

Pour détecter automatiquement les symétries de variables, nous proposons une représentation originale des CNs sous la forme de graphes appelés lsv-graphes. Ces derniers permettent d'identifier un grand nombre de symétries de variables. Notre approche se décompose en 3 étapes. La première consiste en une analyse locale de chaque contrainte de manière à identifier les variables localement symétriques. La seconde correspond à la construction d'un lsv-graphe. Dans la troisième étape, des générateurs du groupe de symétries sont calculés en utilisant un algorithme de détection d'automorphisme, comme proposé dans [5, 14, 13].

3.1 Variables localement symétriques

Nous commençons par introduire le concept de variables localement symétriques (des définitions analogues peuvent être trouvées dans [12, 15]). Deux variables impliquées dans une contrainte C sont localement symétriques si et seulement si ces deux variables peuvent être permutées sans modifier l'ensemble des tuples autorisés.

Définition 3. *Deux variables X et Y sont localement symétriques pour une contrainte C si et seulement si $\{X, Y\} \subseteq scp(C)$ et $\forall \tau \in rel(C), \tau_{X_i \leftrightarrow X_j} \in rel(C)$.*

Par exemple, considérons l'inéquation $X_0 + X_1 + X_2 + 1 < X_3 + X_4$ où le domaine de chaque variable est $\{1, 2, 3\}$. Les variables de l'ensemble $\{X_0, X_1, X_2\}$ sont localement symétriques deux à deux puisque "+" est commutatif et associatif. Le même raisonnement est valable pour $\{X_3, X_4\}$.

Comme la symétrie est transitive, on peut calculer pour chaque contrainte une partition de sa portée, chaque élément de la partition étant un ensemble de variables symétriques deux à deux. La fonction $computeSymmetricalVariables$ (algorithme 1), identifie les variables localement symétriques pour une contrainte C exprimée en extension, en intention, ou par une contrainte globale[1]. Un ensemble S est tout d'abord initialisé avec toutes les variables impliquées dans C. A chaque tour de la boucle principale, l'algorithme sélectionne une variable X de S, et calcule l'ensemble T des variables symétriques avec X. Ensuite, T est soustrait à S et ajouté à la partition Δ en cours de construction.

L'appel à la fonction $isLocallySymmetrical$ (algorithme 2) se trouve au coeur de l'algorithme 1. Pour une contrainte C donnée et deux variables X et Y impliquées dans C, cette fonction détermine si X et Y sont localement symétriques pour C. De façon générale, trois cas peuvent être envisagés en fonction de la représentation de C :

1. Si la contrainte est définie en extension (lignes 1 à 5), l'algorithme construit pour chaque tuple de la table associée à C, un nouveau tuple en permutant les variables X et Y et en contrôlant ensuite si ce dernier appartient à $rel(C)$.
2. Si la contrainte est définie en intention (lignes 6 et 9), l'algorithme construit une représentation arborescente canonique du prédicat $pre(C)$ associé à la contrainte C en appelant la fonction $buildCanonicalTree$ (c.f. section 3.2). Une deuxième représentation arborescente canonique est construite après avoir permuté les variables X et Y dans $pre(C)$, noté $pre(C)_{X \leftrightarrow Y}$. Les deux représentations canoniques sont alors comparées, et X et Y sont identifiées comme étant localement symétriques pour C si et seulement si ces deux représentations sont identiques.
3. Si la contrainte correspond à une contrainte globale (lignes 10 à 15), un traitement spécifique doit être effectué. Par exemple, tous couples de variables impliquées dans une contrainte "AllDifferent" sont localement symétriques. Deux variables impliquées dans une contrainte "WeightedSum" sont localement symétriques si leur coefficient est identique.

Une partition de la portée d'une contrainte (définie en extension ou en intention) peut être calculée en temps polynomial. En effet, nous avons les deux résultats de complexité suivants :

Théorème 1. *La complexité temporelle dans le pire des cas de $computeSymmetricalVariables$ pour une contrainte définie en extension est $O(r^2 t \gamma)$ où r correspond à l'arité de la contrainte C, t la taille de la table*

[1] Nous illustrons notre propos avec deux types de contraintes globales : "AllDifferent" et "WeightedSum".

Algorithm 1: computeSymmetricalVariables(C) : Partition

```
1  Δ ← ∅ ; S ← scp(C)
2  while S ≠ ∅ do
3      pick X from S
4      T ← {X}
5      foreach Y ≠ X ∈ S do
6          if isLocallySymmetrical(C, X, Y) then
7              T ← T ∪ {Y}
8      S ← S \ T ; Δ ← Δ ∪ T
9  return Δ
```

Algorithm 2: isLocallySymmetrical(C, X, Y) : Boolean

```
1  if C is defined in extension then
2      foreach τ ∈ rel(C) do
3          if τ_{X↔Y} ∉ rel(C) then
4              return false
5      return true
6  if C is defined in intension then
7      G ← buildCanonicalTree(pre(C))
8      G' ← buildCanonicalTree(pre(C)_{X↔Y})
9      return G = G'
10 if C is a global constraint then
11     if C is AllDifferent then
12         return true
13     if C is WeightedSum then
14         return coefficient(X) = coefficient(Y)
15     ...
```

associée à C et γ la complexité d'effectuer un test de cohérence.

Démonstration. Le nombre d'appels à $isLocallySymmetrical$ est borné par $O(r^2)$. Dans le pire des cas, nous devons itérer tous les tuples de la table associée à C et effectuer un test de cohérence dont la complexité est γ. On obtient donc $O(r^2 t\gamma)$. □

Nous considérons ici que les contraintes sont définies en extension par une table positive, c'est à dire une table de tuples autorisés. Cependant, on peut aisément adapter le code et les complexités indiqués ici pour des tables négatives.

Théorème 2. *La complexité temporelle dans le pire des cas de $computeSymmetricalVariables$ pour une contrainte C définie en intension est $O(r^2p^2log(p))$ où r représente l'arité de C et p la taille de $pre(C)$.*

Démonstration. Le nombre d'appels à $isLocallySymmetrical$ (et donc à $buildCanonicalTree$) est borné par $O(r^2)$. Comme montré dans le théorème 3 de la section 3.2, $buildCanonicalTree$ est en $O(p^2log(p))$. Donc, on obtient $O(r^2p^2log(p))$. □

En pratique, on peut espérer une meilleure complexité que celle indiquée dans le pire des cas. Tout d'abord, le nombre d'appels à $isLocallySymmetrical$ est seulement $r-1$ lorsque la contrainte est totalement symétrique (i.e. toutes les variables sont localement symétriques), à cause de la transitivité de la symétrie. De plus, pour une contrainte exprimée en intention, $buildCanonicalTree$ est réduit à $O(p^2)$ quand les opérateurs sont binaires. Par ailleurs, pour les contraintes exprimées en extension, lorsque deux variables ne sont pas symétriques, on peut espérer quitter rapidement la boucle $foreach$ de l'algorithme 2. Finalement, on peut noter que la fonction $computeSymmetricalVariables$ ne doit pas nécessairement être appelée pour chaque contrainte. En effet, on peut associer une clef (chaîne de caractères) à chaque contrainte, en utilisant les références des domaines des variables comme préfixe. On concatène à cette clef la référence de la relation pour une contrainte exprimée en extension, et une représentation de la forme canonique de l'expression du prédicat (faisant référence aux variables par leurs positions dans la portée) pour une contrainte définie en intention. Deux contraintes ayant la même clef peuvent alors partager la même partition. Cela arrive très souvent sur les instances structurées.

3.2 Calcul des formes normales des prédicats

Comme mentionné précédemment, l'utilisation de formes normales a déjà été proposée [8] pour identifier les variables symétriques d'un CN. Cependant, cette approche est appliquée globalement à l'ensemble des contraintes, la rendant inutilisable en pratique (sauf pour de petites instances). Ici, nous proposons d'appliquer une approche similaire, mais sur chaque contrainte prise individuellement. Par conséquent, cette approche peut être appliquée de manière efficace, sauf pour des cas très spécifiques, lorsque l'expression ou l'arité des contraintes est très grande.

Pour rendre notre présentation concrète, nous considérons ici la grammaire, introduite dans le contexte des compétitions de solveurs CSP (`http://cpai.ucc.ie/08/XCSP2_1.pdf`) pour construire les expressions. La plupart des opérateurs à partir desquels les expressions sont construites sont à la fois commutatifs et associatifs : add (+), mul (*), min, max, and, or, xor, iff, eq (=), ne ($\neq$). Nous introduisons quelques règles de réécritures simples :

- Regrouper les opérateurs associatifs en utilisant un opérateur n-aire équivalent [14]. Par exemple, remplacer $add(X, add(Y, Z))$ par $add(X, Y, Z)$.
- Supprimer toutes les occurrences de "not" (opérateur logique unaire de négation). Par exemple, remplacer $not(and(X, Y))$ par $or(X, Y)$.
- Remplacer toutes les occurrences de ge ($\geq$) et gt ($>$) par le ($\leq$) et lt ($<$) [8]. Par exemple, remplacer $ge(X, Y))$ par $le(Y, X)$.
- Remplacer la séquence 'abs sub' avec un nouvel opérateur commutatif 'abssub' combinant ces deux opé-

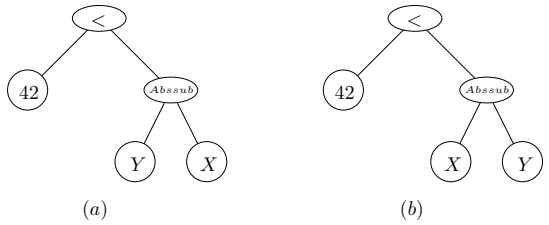

FIG. 1 – Représentation initiale (a) et canonique (b) du prédicat $|Y - X| > 42$ construite avec la fonction $buildCanonicalTree$

rateurs. Par exemple, remplacer $abs(sub(X, Y))$ par $abssub(X, Y)$.

Il est possible de construire un arbre syntaxique (chaque noeud est associé avec un token de l'expression) en une passe tout en prenant en compte l'ensemble des règles indiquées précédemment. Même si d'autres règles supplémentaires, plus sophistiquées, peuvent être imaginées (e.g. on peut adopter des règles spécifiques pour les équations linéaires et non-linéaires), nous pensons que cet ensemble simple de règles est suffisant pour capturer les symétries de variables de la plupart des contraintes. Remarquons également l'importance du nouvel opérateur 'abssub' qui apparaît dans de nombreuses séries d'instances (e.g. problèmes d'assignation de fréquences). Ce nouvel opérateur étant commutatif, on peut identifier des symétries non-détectées lorsque l'opérateur (non-commutatif) sub est présent.

Pour obtenir une forme canonique à partir de l'arbre syntaxique initial, il suffit de rendre canonique la racine de l'arbre. Un noeud est rendu canonique de la manière suivante : tous les fils (s'il y en a plusieurs) sont rendus canoniques, et triés si l'intitulé associé avec le noeud correspond à un opérateur commutatif. Pour obtenir une forme normale, il est nécessaire de définir un ordre total sur l'ensemble des opérateurs, entiers et variables. Cet ordre est construit naturellement [8].

Par exemple, la figure 1(a) illustre la représentation d'un arbre syntaxique initial (i.e. avant normalisation) du prédicat $|Y - X| > 42$. Comme le nouvel opérateur "abssub" est commutatif, on obtient la représentation canonique de la figure 1(b), en considérant ici que les variables sont triées par leur nom selon l'ordre lexicographique.

Théorème 3. *La complexité temporelle dans le pire des cas pour construire un arbre dans sa forme canonique, à partir de l'expression $expr$, est $O(p^2 log(p))$ où p représente la taille de $expr$.*

Démonstration. En $O(p)$ opérations, on peut construire un arbre contenant p noeuds, tout en prenant en considération l'ensemble des règles de réécriture introduites précédemment. Dans le pire des cas, le nombre cumulé de comparaisons nécessaires pour trier tous les noeuds est borné par $O(p.log(p))$, et chaque comparaison implique la visite d'au plus p noeuds. On obtient donc $O(p^2 log(p))$. □

3.3 Construction des lsv-graphes

Une fois que les variables symétriques ont été identifiées pour chaque contrainte (en utilisant le partitionnement), on peut construire un graphe coloré dédié à la recherche des symétries de variables d'un CN donné. Chaque automorphisme de ce graphe correspond à une symétrie de variables du CN. C'est une approche introduite dans [5] et exploitée, par exemple, dans [1, 14, 13]. Aucune complexité polynomiale dans le pire des cas n'est connue pour le problème consistant à trouver tous les automorphismes d'un graphe (mais il est communément admis que ce problème n'est pas NP-complet). Nous montrons dans cette section que les graphes colorés que nous construisons sont de taille limitée tout en capturant des symétries de variables.

Nous détaillons la construction de graphes colorés, notés lsv-graphes à partir de maintenant. Chaque variable d'un CN P donné est représentée par un noeud, noté "noeud-variable". Pour chaque contrainte d'arité r, on ajoute un "noeud-contrainte" et r "noeuds-liaisons", un pour chaque variable impliquée dans la contrainte. Les noeuds-liaisons permettent de connecter les noeuds-contraintes avec les noeuds-variables : si C est une contrainte impliquant X alors le noeud-contrainte correspondant à C et le noeud-variable correspondant à X sont connectés par l'intermédiaire d'un noeud-liaison.

Une couleur est associée à chaque noeud du graphe (les permutations ne sont autorisées qu'entre des noeuds de même couleur). Tout d'abord les variables ayant le même domaine ont la même couleur (alternativement, on pourrait utiliser une couleur unique pour tous les noeuds-variables en insérant des contraintes unaires représentant les domaines). Les contraintes similaires (i.e. les contraintes définies par la même relation) ont la même couleur. Pour chaque contrainte, les noeuds-liaisons correspondant aux variables localement symétriques ont la même couleur. Le même schéma de couleurs pour les noeuds-liaisons est utilisé pour des contraintes similaires. Dans tous les autres cas, des couleurs différentes doivent être utilisées.

On peut montrer que la construction ci-dessus est correcte : chaque automorphisme d'un lsv-graphe correspond à une symétrie de variables du CN. En effet, tous les éléments (domaines, contraintes) contraignant l'espace de recherche d'un CN sont pris en compte lors de la construction de la structure du graphe et de l'assignation des couleurs.

Pour illustrer la construction d'un lsv-graphe, considérons le réseau de contraintes P impliquant un ensemble de quatre variables $\{X_0 \dots X_3\}$ et un ensemble de quatre contraintes exprimées en intention $\{C_0 \dots C_3\}$. Le prédicat associé à C_0 et C_1 est $|\$1 - \$0| = 56$ tandis que pour C_2 et C_3 le prédicat est $|\$1 - \$0| > 42$. $\$i$ représente le i^{eme} paramètre formel du prédicat. La figure 2 représente le lsv-graphe construit à partir de P : les quatre noeuds blancs correspondent à des noeuds-variables (on suppose ici que les variables possèdent le même domaine)

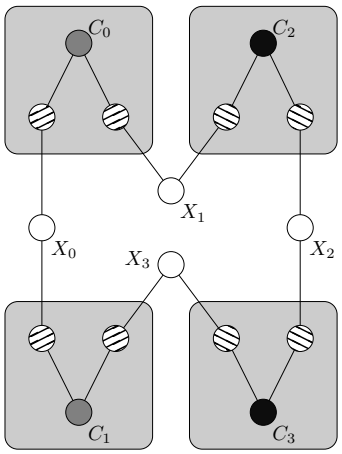

FIG. 2 – Illustration d'un lsv-graphe

et les noeuds gris et noirs correspondent à des noeuds-contraintes (les noeuds gris pour C_0 et C_1 et les noeuds noirs pour C_2 et C_3). Chaque contrainte est reliée aux deux noeuds-variables par l'intermédiaire de deux noeuds-liaisons puisque les contraintes sont binaires.

En exécutant l'algorithme 1 sur C_0, les variables X_0 et X_1 sont détectées comme localement symétriques pour C_0. Par conséquent, les deux noeuds-liaisons introduits pour C_0 reçoivent la même couleur : représentée sur le schéma par un motif hachuré à droite. Sur notre exemple, C_1 est similaire à C_0 (la relation est la même puisque les deux contraintes sont représentées par la même expression). C'est pourquoi la même couleur est utilisée pour C_0 et C_1. Le même principe s'applique aux contraintes C_2 et C_3 : la même couleur est assignée à tous les noeuds-liaisons de ces deux contraintes, représentée ici par un motif hachuré à gauche. En utilisant Saucy, un générateur pour le groupe de symétrie est identifié : celui-ci mappe X_1 en X_3, et vice-versa. Ceci peut être observé sur la figure 2.

Un avantage de notre approche réside dans la taille assez limitée des lsv-graphes. Celle-ci est linéaire en fonction de la somme des arités des contraintes, majorée en utilisant r la plus grande arité des contraintes.

Théorème 4. *Soit P un CN. Le nombre de noeuds et d'arêtes d'un lsv-graphe construit à partir de P est $O(er)$.*

Démonstration. Pour chaque variable et pour chaque contrainte de P, un noeud est construit. Pour chaque variable impliquée dans une contrainte, un noeud est également construit ainsi que deux arêtes. En considérant ici que n est $O(e)$, nous obtenons une complexité globale de $O(er)$ pour le nombre de noeuds et le nombre d'arêtes. □

Finalement, il est important de mettre en relation notre approche avec celle décrite dans [13] lorsque cette dernière est restreinte aux symétries de variables. Notre approche permet tout d'abord de s'abstraire de la représentation des contraintes : quelque soit la représentation (extension, intention, global), on manipule le même modèle de noeuds-liaisons. Un deuxième avantage réside dans la taille des lsv-graphes générés. Celle-ci est linéaire tandis que la taille des graphes de Puget croît exponentiellement avec l'arité des contraintes lorsqu'elles sont définies en extension et que le nombre de tuples autorisés et interdits n'est pas borné. A titre illustratif, une contrainte d'arité r définie par une table de t tuples nécessite $r + 1$ noeuds spécifiques (1 noeud-contrainte et r noeuds-relations) avec notre représentation, en comparaison aux $t + 1 \in O(d^r)$ noeuds (1 noeud pour la contrainte et 1 noeud par tuple) de [13]. On peut argumenter que l'utilisation d'un outil efficace comme AUTOM [13] permet d'atténuer cette différence. Cependant, à notre connaissance, cet outil n'est pas disponible.

Les deux approches permettent de détecter le même groupe de symétries de variables lorsque les contraintes sont binaires. Ceci n'est pas toujours vrai lorsque les contraintes sont non-binaires. Par exemple, la contrainte C telle que $scp(C) = \{X_1, X_2, X_3, X_4\}$ et $rel(C) = \{(4,3,2,1),(1,2,3,4)\}$ admet une symétrie de variables σ avec $\sigma(X_1) = X_4$, $\sigma(X_2) = X_3$, $\sigma(X_3) = X_2$ et $\sigma(X_4) = X_1$, mais pas de variables localement symétriques. Même si de tels cas ne sont pas si fréquents en pratique, étendre notre approche pour manipuler des groupes de variables localement symétriques (c'est à dire une généralisation des variables localement symétriques), tout en contrôlant à la fois la complexité temporelle de la phase de détection des symétries locales et la complexité spatiale des lsv-graphes générés est une perspective de ce travail.

4 GAC pour la contrainte Lex

Une fois les symétries détectées, il reste à les exploiter. Une approche connue consiste à ajouter au problème des contraintes les éliminant [5, 16]. De manière classique, une contrainte d'ordre lexicographique est postée pour chaque générateur retourné par le programme d'identification d'automorphismes. En effet, il a été montré qu'il était généralement préjudiciable de poster une contrainte pour chaque symétrie obtenue à partir de combinaisons des générateurs [1]. Nous introduisons ici un nouvel algorithme établissant GAC sur les contraintes d'ordre lexicographique. Il est important de souligner que cet algorithme peut être utilisé en présence de variables partagées, c'est-à-dire lorsque certaines variables apparaissent plusieurs fois au niveau des deux vecteurs sur lesquels est définie la contrainte. La seule restriction est qu'une variable ne doit pas être présente à la même position dans les deux vecteurs. Toutefois, il est toujours possible de réduire les vecteurs (éliminer les variables aux mêmes position) tout en préservant l'équivalence. Cet algorithme est très simple à implanter et bien adapté aux solveurs génériques.

Les contraintes d'ordre lexicographique sont donc défi-

nies sur deux vecteurs de variables. Ici, sans perte de généralité, on suppose que les deux vecteurs ont la même longueur et que le domaine de chaque variable est uniquement composé d'entiers. Un vecteur de variables est noté par $\overrightarrow{X} = \langle X_0, X_1, \ldots, X_{q-1} \rangle$, et un sous-vecteur $\langle X_i, \ldots, X_j \rangle$ de la position $i \geq 0$ à la position $j \leq q-1$ inclue par $\overrightarrow{X}_{i..j}$.

Définition 4. *Une contrainte d'ordre lexicographique est définie sur deux vecteurs $\overrightarrow{X}$ et $\overrightarrow{Y}$ de variables. On a : $\overrightarrow{X} = \langle X_0, X_1, \ldots, X_{q-1} \rangle \leq_{lex} \overrightarrow{Y} = \langle Y_0, Y_1, \ldots, Y_{q-1} \rangle$ ssi les deux vecteurs sont vides ou $X_0 < Y_0$ ou $X_0 = Y_0$ et $\langle X_1, \ldots, X_{q-1} \rangle \leq_{lex} \langle Y_1, \ldots, Y_{q-1} \rangle$; $\overrightarrow{X} <_{lex} \overrightarrow{Y}$ ssi $\overrightarrow{X} \leq_{lex} \overrightarrow{Y}$ et $\overrightarrow{X} \neq \overrightarrow{Y}$.*

Le premier type de contraintes est noté Lex. Lorsque les vecteurs ne contiennent qu'une variable, on obtient une contrainte binaire "LessThanOrEqual" noté Le. Les contraintes d'ordre lexicographique strict ($<_{lex}$) ne seront pas considérées dans cet article.

A notre connaissance, seulement deux algorithmes ont été décrits dans la littérature pour établir GAC sur les contraintes Lex. Le premier, décrit dans [7], est basé sur l'exploitation de deux indices notés α et β. L'indice α est la plus petite position de variables dans $\overrightarrow{X}$ et $\overrightarrow{Y}$ non closes et égales (q si une telle position n'existe pas). L'indice β est la plus petite position à laquelle on a $\overrightarrow{X}_{\beta..q-1} >_{lex} \overrightarrow{Y}_{\beta..q-1}$ ($q+1$ si une telle position n'existe pas). En raisonnant à partir de α et β, on peut établir efficacement GAC : la complexité temporelle dans le pire des cas d'un appel est $O(q\lambda)$ où q est la longueur des deux vecteurs et λ la complexité de réduire la borne d'un domaine. Typiquement, dans les implantations de solveurs, nous avons λ qui est soit $O(1)$ soit $O(d)$ où d désigne la taille du plus grand domaine. Le deuxième algorithme, introduit dans [3], exploite un automate à états finis qui opère sur une chaîne de caractères capturant les relations existant entre chaque couple de variables des deux vecteurs. La chaîne (signature) est construite à partir des domaines courants et fournie comme entrée à l'automate. Le filtrage est réalisé au cours des transitions dans l'automate. Cette approche élégante admet également une complexité temporelle en $O(q\lambda)$ plus un temps amorti constant par événement de propagation (en considérant λ en $O(1)$).

Une limitation des deux algorithmes proposés est qu'ils ne sont pas adaptés au cas des variables partagées. Lorsque des contraintes Lex sont utilisées pour éliminer des symétries (e.g. avec l'approche lex-leader [5]), cela peut arriver fréquemment. Par exemple,

$$\begin{array}{lccccc} \overrightarrow{X} = \langle & V_0, & V_1, & V_2, & V_3 & \rangle \\ & \{0,1\} & \{0,1\} & \{0,1\} & \{1\} & \\ & \{0,1\} & \{0,1\} & \{0,1\} & \{0\} & \\ \overrightarrow{Y} = \langle & V_1, & V_2, & V_0, & V_4 & \rangle \end{array}$$

représente la contrainte $\overrightarrow{X} \leq_{lex} \overrightarrow{Y}$ avec les variables

Algorithm 3: establishGAC($\langle X_0, X_1, \ldots, X_{q-1} \rangle \leq_{lex} \langle Y_0, Y_1, \ldots, Y_{q-1} \rangle$) : Boolean

```
1  α ← 0
2  while α < q do
3      if establishAC(X_α ≤ Y_α) = false then return false
4      if |dom(X_α)| ≠ 1 ∨ dom(X_α) ≠ dom(Y_α) then break
5      α ← α + 1
6  if α ≥ q − 1 then return true
7  if min(X_α) = min(Y_α) ∧ ¬isConsistent(α, min(Y_α)) then
8      remove min(Y_α) from dom(Y_α)
9  if max(X_α) = max(Y_α) ∧ ¬isConsistent(α, max(X_α)) then
10     remove max(X_α) from dom(X_α)
11 return true
```

Algorithm 4: isConsistent(α : integer, value : integer) : Boolean

```
1  S ← {(X_α, value), (Y_α, value)}
2  for i ranging from α + 1 to q − 1 do
3      if X_i ∈ vars(S) then valueX ← S[X_i]
4      else valueX ← min(X_i)
5      if Y_i ∈ vars(S) then valueY ← S[Y_i]
6      else valueY ← max(Y_i)
7      if valueX < valueY then return true
8      if valueX > valueY then return false
9      S ← S ∪ {(X_i, valueX), (Y_i, valueY)}
10 return true
```

partagées V_0, V_1 et V_2. Le domaine de toutes les variables est $\{0,1\}$, sauf pour V_3 et V_4 dont le domaine respectif est $\{1\}$ et $\{0\}$. Si nous utilisons les algorithmes évoqués ci-dessus, nous ne filtrons rien bien que $(V_0, 1)$ n'ait aucun support. Pour pallier ce problème, Z. Kiziltan a proposé une extension [10] à l'algorithme de [7]. Nous en discuterons après avoir introduit notre algorithme.

Pour établir GAC sur une contrainte $Lex\ C : \langle X_0, X_1, \ldots, X_{q-1} \rangle \leq_{lex} \langle Y_0, Y_1, \ldots, Y_{q-1} \rangle$, l'algorithme 3 est appelé. Aussi longtemps que X_α et Y_α sont clos et égaux (le test à la ligne 4 est la condition inverse) après avoir établi AC sur $X_\alpha \leq Y_\alpha$ (ligne 3), la valeur de α est incrémentée. Ce processus s'arrête lorsqu'une incohérence est trouvée par AC, lorsque $\alpha = q$ ou lorsque X_α et Y_α ne sont plus clos et égaux. La valeur de α obtenue est similaire à celle de [7] mais elle est calculée de manière différente (de plus, à α, nous avons déjà établi $X_\alpha \leq Y_\alpha$). Si $\alpha = q$, cela signifie que toutes les variables sont closes et égales, et donc que la contrainte est GAC. Si $\alpha = q-1$, seules les deux dernières variables (le dernier couple) ne sont pas closes et égales. Cependant, nous savons que $X_\alpha \leq Y_\alpha$, donc, il n'est pas difficile de constater que la contrainte est GAC. Les deux cas sont considérés à la ligne 6. De façon intéressante, lorsque la ligne 7 est atteinte, nous savons que seulement deux valeurs ne sont pas garanties GAC, à savoir, $(X_\alpha, max(X_\alpha))$ et $(Y_\alpha, min(Y_\alpha))$. En outre, même si ces valeurs sont éliminées, toutes les autres demeurent GAC. Ceci est établi par le lemme 1 donné ci-

dessous. Notons que $min(Y_\alpha)$ peut ne pas avoir de support(s) sous la condition $min(Y_\alpha) = min(X_\alpha)$. Si cette condition est vérifiée, nous devons chercher un support de $min(Y_\alpha)$ en appelant la fonction $isConsistent$. Ici, l'opérateur $\wedge$ doit être évalué en mode court-circuit. La valeur $max(X_\alpha)$ doit être traitée de manière similaire.

Pour trouver un support lorsque $min(X_\alpha) = min(Y_\alpha)$ ou $max(X_\alpha) = max(Y_\alpha)$, on doit donc appeler la fonction $isConsistent$ (algorithme 4). Le principe est d'enregistrer dans un ensemble S, les valeurs qui doivent être considérées comme étant assignées : $vars(S)$ représente l'ensemble des variables assignées par S, et $S[X_i]$ la valeur assignée à X_i par S. Cet ensemble est initialisé avec la même valeur pour X_α et Y_α. Pour chaque position i, nous comparons la plus petite valeur de X_i avec la plus grande valeur de Y_i (mais lorsque les variables sont pseudo-assignées, leurs valeurs enregistrées dans S doivent être considérées). A partir du résultat de cette comparaison, un support, un échec ou la nécessité d'assigner est identifié. De manière schématique ; la fonction $isConsistent$ correspond à la procédure $SeekSupport$ de [10].

Pour prouver que l'algorithme proposé établit GAC, nous introduisons le lemme suivant. Les preuves sont omises par manque de place.

Lemme 1. *Lorsque la ligne 7 de l'algorithme 3 est atteinte, nous avons :*

- *toute valeur appartenant à* $values(C) \setminus \{(X_\alpha, max(X_\alpha), (Y_\alpha, min(Y_\alpha)\}$ *est GAC ;*
- *si* $|dom(X_\alpha)| = 1$*, alors la valeur unique de* $dom(X_\alpha)$ *est GAC ;*
- *si* $|dom(Y_\alpha)| = 1$*, alors la valeur unique de* $dom(Y_\alpha)$ *est GAC.*

Théorème 5. *L'algorithme 3 établit GAC, et sa complexité temporelle dans le pire des cas est* $O(q\lambda)$.

Il est intéressant de noter que cet algorithme peut être rendu incrémental en préservant simplement la valeur de α d'un appel à l'autre. L'algorithme alternatif introduit dans [10] pour gérer les variables partagées utilise une valeur β, mais dans le pire des cas, cette valeur reste égale à $q + 1$. En conséquence, dans le pire des cas, un support devra être recherché à chaque appel en passant en revue toutes les variables entre $\alpha + 1$ et $q - 1$. Les deux algorithmes ont donc un comportement similaire dans le pire des cas. Par ailleurs, si l'effort de propagation peut parfois être réduit en utilisant β, calculer cette valeur peut être coûteuse en comparaison de notre approche (puisque notre algorithme s'arrête aussitôt que GAC est garanti). Pour finir, soulignons que notre algorithme est indépendant d'une gestion fine des événements. Avec les approches précédentes, tous les événements (concernant les bornes des domaines) doivent être pris en compte individuellement afin de mettre à jour les structures (valeurs de α et β ou états de l'automate). Avec notre approche, les événements peuvent être agrégés, économisant potentiellement de nombreux appels.

5 Expérimentations

Pour montrer l'intérêt pratique de notre approche, nous avons conduit des expérimentations sur un cluster de Xeon 3.0GHz avec 1GiB de RAM. Les performances sont mesurées en terme de temps cpu (en secondes) pour l'intégralité de la résolution (identification et exploitation des symétries incluses) et en nombre de noeuds visités. Plusieurs variantes de l'approche décrite dans ce papier ont été intégrées à l'algorithme M(G)AC, i.e. l'algorithme maintenant (G)AC à chaque noeud de l'arbre de recherche. Nous avons utilisé différentes heuristiques de choix de variables ($dom/ddeg$, $brelaz$ et $dom/wdeg$) et le temps limite a été fixé à 20 minutes par instance.

Nous avons utilisé Saucy pour identifier les symétries de variables. Pour chaque générateur du groupe de symétries retourné par Saucy, quatre procédures d'élimination des symétries ont été envisagées. Pour la première, notée MAC_{Le}, on ajoute une contrainte binaire Le dont la portée contient les deux premières variables du premier cycle du générateur. Pour la seconde, notée MAC_{Lex}, on ajoute une contrainte d'ordre lexicographique Lex (impliquant toutes les variables de tous les cycles du générateur). Naturellement, une contrainte Lex est plus forte que la contrainte Le correspondante : sa capacité de filtrage est meilleure. On peut remarquer que lorsque les deux premières variables du premier cycle d'un générateur appartiennent à la portée d'une contrainte (non globale) C du réseau, on peut fusionner C avec une contrainte binaire Le. En appliquant cette méthode de fusion, on obtient deux méthodes additionnelles, notées MAC^*_{Le} et MAC^*_{Lex}.

Dans notre première expérimentation, nous avons testé les quatre variantes (plus MAC seul) sur l'ensemble complet des instances de la compétition de solveurs CSP 2006, ainsi que sur certaines séries récemment mises en ligne à l'adresse `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks`. Le tableau 1 fournit un résumé des résultats obtenus en terme de nombre d'instances résolues dans le temps imparti. Quelque soit l'heuristique de choix de variables, le nombre d'instances résolues est augmenté lorsqu'on utilise MAC^* (méthode fusionnée). Même si globalement, MAC_{Lex} résout plus d'instances que MAC_{Le}, cette dernière approche représente une alternative efficace, qui peut être facilement implantée.

Dans le tableau 2, nous nous focalisons sur quelques instances. Nous ne fournissons que les résultats obtenus avec MAC^*_{Le} et MAC^*_{Lex}, ces derniers étant surpassés par MAC_{Le} et MAC_{Lex} (c.f. tableau 1). Sur les séries d'instances impliquant des variables booléennes ($bibd$, $chnl$, $fpga$), MAC^*_{Lex} est (sans surprise) très efficace. La différence entre MAC^*_{Lex} et MAC^*_{Le} est moins significative

	MAC	MAC_{Le}	MAC_{Lex}	MAC^*_{Le}	MAC^*_{Lex}
$dom/wdeg$	$2,886$	$2,941$	$2,921$	$2,944$	$2,952$
$dom/ddeg$	$2,394$	$2,424$	$2,444$	$2,458$	$2,486$
$brelaz$	$2,415$	$2,452$	$2,469$	$2,474$	$2,504$

TAB. 1 – Nombre d'instances résolues par MAC et ses variantes éliminant les symétries de variables sur une sélection de $4,003$ instances.

		$dom/ddeg$			$dom/wdeg$		
		MAC	MAC^*_{Le}	MAC^*_{Lex}	MAC	MAC^*_{Le}	MAC^*_{Lex}
haystack-06	cpu	9.83	0.45	0.46	$time\text{-}out$	0.44	0.44
	$nodes$	$125K$	40	12	–	28	12
haystack-08	cpu	$time\text{-}out$	0.72	0.62	$time\text{-}out$	0.61	0.61
	$nodes$	–	$1,359$	595	–	641	501
haystack-10	cpu	$time\text{-}out$	12.2	1.77	$time\text{-}out$	4.42	1.49
	$nodes$	–	$105K$	$9,822$	–	$22,764$	$4,524$
haystack-12	cpu	$time\text{-}out$	738	55.0	$time\text{-}out$	195	8.82
	$nodes$	–	$6,565K$	$416K$	–	$962K$	$38,830$
haystack-14	cpu	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	452
	$nodes$	–	–	–	–	–	$2,120K$
fpga-10-8	cpu	$time\text{-}out$	$time\text{-}out$	9.2	12.4	1.66	1.06
	$nodes$	–	–	$66,053$	$88,611$	$7,706$	$4,148$
fpga-11-9	cpu	$time\text{-}out$	$time\text{-}out$	18.3	271	19.6	1.14
	$nodes$	–	–	$140K$	$1,519K$	$116K$	$3,228$
fpga-12-10	cpu	$time\text{-}out$	$time\text{-}out$	694	$time\text{-}out$	3.28	5.96
	$nodes$	–	–	$3,005K$	–	$12,333$	$35,462$
fpga-13-11	cpu	$time\text{-}out$	$time\text{-}out$	417	$time\text{-}out$	566	5.32
	$nodes$	–	–	$1,985K$	–	$2,380K$	$23,811$
fpga-14-12	cpu	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$	10.7
	$nodes$	–	–	–	–	–	$41,809$
chnl-10-11	cpu	$time\text{-}out$	614	1.06	$time\text{-}out$	377	1.19
	$nodes$	–	$3,301K$	$1,827$	–	$1,617K$	$2,572$
chnl-10-15	cpu	$time\text{-}out$	$time\text{-}out$	3.14	$time\text{-}out$	$time\text{-}out$	5.32
	$nodes$	–	–	$2,571$	–	–	$2,706$
chnl-10-20	cpu	$time\text{-}out$	$time\text{-}out$	75.3	$time\text{-}out$	$time\text{-}out$	193
	$nodes$	–	–	$3,501$	–	–	$5,273$
chnl-15-20	cpu	$time\text{-}out$	$time\text{-}out$	762	$time\text{-}out$	$time\text{-}out$	$time\text{-}out$
	$nodes$	–	–	$101K$	–	–	–
bibd-6-60-30	cpu	746	703	1.8	477	$1,129$	2.15
	$nodes$	$2,938K$	$2,938K$	$2,949$	$1,832K$	$4,330K$	$2,876$
bibd-6-80-40	cpu	$time\text{-}out$	$time\text{-}out$	3.11	$time\text{-}out$	$time\text{-}out$	3.06
	$nodes$	–	–	$5,423$	–	–	$3,524$
bibd-7-28-12	cpu	108	108	2.67	0.82	2.58	1.59
	$nodes$	$362K$	$362K$	$5,066$	$1,052$	$10,139$	$3,310$
bibd-7-49-21	cpu	$time\text{-}out$	$time\text{-}out$	8.86	1.11	166	$time\text{-}out$
	$nodes$	–	–	$13,587$	$1,776$	$681K$	–
bibd-9-36-12	cpu	$time\text{-}out$	$time\text{-}out$	6.88	11.7	49.3	186
	$nodes$	–	–	$8,187$	$41,568$	$150K$	$554K$
bibd-9-60-20	cpu	$time\text{-}out$	$time\text{-}out$	28.3	186	$1,128$	6.63
	$nodes$	–	–	$25,919$	$438K$	$3,176K$	$8,786$

TAB. 2 – Coût de MAC et de ses variantes éliminant les symétries de variables sur quelques instances.

		MAC	MAC_{Le}	MAC_{Lex}	MAC^*_{Le}	MAC^*_{Lex}
scen11-f9	cpu	2.15	1.96	2.23	1.84	1.97
	$nodes$	$1,064$	576	922	109	90
scen11-f7	cpu	4.83	2.28	2.37	1.91	2.05
	$nodes$	$8,369$	955	$1,247$	121	135
scen11-f5	cpu	32.0	2.2	3.13	2.19	2.13
	$nodes$	$85,104$	988	$3,465$	253	226
scen11-f4	cpu	112	2.66	3.88	2.36	2.53
	$nodes$	$345K$	$1,983$	$5,007$	593	903
scen11-f3	cpu	403	3.41	7.98	2.55	2.45
	$nodes$	$1,300K$	$3,926$	$17,259$	946	696
scen11-f2	cpu	$time\text{-}out$	4.32	16.4	2.95	2.92
	$nodes$	–	$6,014$	$40,615$	$1,700$	$1,591$
scen11-f1	cpu	$time\text{-}out$	7.56	19.7	3.49	3.4
	$nodes$	–	$14,997$	$47,318$	$3,199$	$2,609$

TAB. 3 – Coût de MAC et de ses variantes éliminant les symétries de variables sur des instances RLFAP difficiles (38 générateurs). L'heuristique de choix de variables est $dom/wdeg$.

sur d'autres séries comme les $haystack$.

Le tableau 3 met en valeur les résultats obtenus pour les instances les plus difficiles (qui impliquent 680 variables et dont le domaine le plus grand contient 50 valeurs) construites à partir du problème issu du monde réel d'assignation de fréquences radios (RLFAP). Clairement, les méthodes d'élimination de symétries sont plus efficaces que l'algorithme classique MAC. Ceci est en parti dû à l'utilisation de l'opérateur "abssub" qui permet de détecter des variables localement symétriques. Notons que le résultat obtenu (moins de 4 secondes) pour la plus difficile instance $scen11\text{-}f1$ est plutôt encourageant puisqu'à notre connaissance, un seul solveur a réussi à résoudre cette instance auparavant (en 70 secondes environ - c.f. `http://www.cril.univ-artois.fr/CPAI06`).

De manière intéressante, sur le problème des "Pigeons" encodé par une clique de contraintes binaires "$NotEqual$", MAC^* peut prouver l'incohérence de ces instances (des résultats plus précis ne sont pas indiqués ici, par manque de place) en établissant juste AC pendant la phase de prétraitement. En effet, combiner les nouvelles contraintes Le avec les contraintes de la clique initiale revient à fixer un ordre total sur les variables du problème.

6 Conclusion

Dans ce papier, nous avons introduit une nouvelle représentation sous forme de graphes des CNs en s'appuyant sur l'identification de variables localement symétriques. Cette représentation est rendue homogène en faisant abstraction du type de contraintes, et la taille des lsv-graphes ainsi obtenus est linéaire en fonction de la somme des arités des contraintes. Inspirée à la fois des règles de réécriture [8] et de l'identification d'automorphisme de graphes [5, 14, 13], notre approche permet alors de détecter automatiquement un grand nombre de symétries de variables. Pour éliminer les symétries, des contraintes d'ordre lexicographique sont ajoutées. Ici, nous proposons un algorithme simple et général établissant GAC sur ces contraintes qui peuvent prendre en compte des variables partagées.

L'intérêt pratique de notre approche a été montré sur de nombreuses séries d'instances. En effet, le nombre d'instances résolues a été amélioré de façon significative démontrant ainsi la robustesse de cette approche. Il est important de noter que le temps nécessaire pour identifier les variables localement symétriques et pour calculer les automorphismes des lsv-graphes a été observé comme négligeable durant nos expérimentations. Pour résumer, nos résultats confirment le fait que l'élimination automatique de symétries constitue une amélioration significative pour les solveurs CSP de type black-box.

Remerciements

Ce travail a été supporté par l'IUT de Lens, le CNRS et le projet ANR Planevo.

Références

[1] F.A. Aloul, K.A. Sakallah, and I.L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5) :549–558, 2006.

[2] B. Benhamou. Study of symmetry in constraint satisfaction problems. In *Proceedings of CP'94*, pages 246–254, 1994.

[3] M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002.

[4] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3) :115–137, 2006.

[5] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR'96*, pages 148–159, 1996.

[6] P.T. Darga, M.H. Liffiton, K.A. Sakallah, and I.L. Markov. Exploiting structure in symmetry generation for CNF. In *Proceedings of DAC'04*, pages 530–534, 2004.

[7] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10) :803–834, 2006.

[8] A. Frisch, I. Miguel, and T. Walsh. CGRASS : A system for transforming constraint satisfaction problems. In *Proceedings of CSCLP'02*, pages 23–36, 2002.

[9] I.P. Gent, K.E. Petrie, and J.F. Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, chapter 10, pages 329–376. Elsevier, 2006.

[10] Z. Kiziltan. *Symmetry breaking ordering constraints*. PhD thesis, Uppsala University, 2004.

[11] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30 :45–87, 1981.

[12] J.F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of ISMIS'93*, pages 475–489, 1993.

[13] J.F. Puget. Automatic detection of variable and value symmetries. In *Proceedings of CP'05*, pages 475–489, 2005.

[14] A. Ramanai and I.L. Markov. Automatically exploiting symmetries in constraint programming. In *Proceedings of CSCLP'04*, pages 98–112, 2004.

[15] P. Roy and F. Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *Proceedings of the workshop on non-binary constraints, held with ECAI'98*, 1998.

[16] T. Walsh. General symmetry breaking constraints. In *Proceedings of CP'06*, pages 650–664, 2006.