Génération rapide et filtrage de configurations canoniques

Laurent Henocque, Nicolas Prcovic

Génération rapide et filtrage de configurations canoniques

Laurent Henocque **Nicolas Prcovic**

LSIS - Universités d'Aix-Marseille II et III

{laurent.henocque,nicolas.prcovic}@lsis.org

Résumé

La configuration sous contraintes présente une nouvelle difficulté à prendre en compte par les méthodes d'élimination de symétries connues par la communauté CSP car elle y introduit un aspect dynamique. Nous présentons ici une amélioration significative d'un algorithme de génération de configurations canoniques. Cette nouvelle version exploite l'incrémentalité que l'on peut faire ressortir de la génération de solutions canoniques et de l'ordre total sur les arbres sur laquelle elle repose. La complexité du test de canonicité passe ainsi de O(Nlog(N)) à O(N). De plus, une technique de filtrage nous permet d'éliminer à l'avance des configurations non canoniques. Des résultats expérimentaux montrent l'intérêt de cette approche sur des problèmes classiques.

Abstract

Constraint based configuration challenges symmetry elimination methods known to the constraint solving community by introducing dynamics. We present here a significant improvement of an algorithm for generating canonical configurations. The new version fully exploits the incremental generation of canonical solutions both at the level of the canonicity test and in the tree ordering function, which turns the cost of canonicity testing down from O(Nlog(N)) to O(N). Filtering additionally provides the possibility to proactively discard failure situations. Experimental results provide evidence of the significance of this approach, on test problems and known benchmarks.

1 Introduction

La configuration sous contraintes [3, 15, 1, 16, 17, 18, 14] a la particularité qu'il faut gérer l'aspect dynamique du traitement des symétries qu'elle fait apparaître en cours de résolution. La taille des solutions des problèmes de configuration étant potentiellement infini, le problème de la configuration est semi-décidable, ce qui nécessite une approche spécifique pour traiter les problèmes d'isomorphisme des structures des configurations. Alors, une fois la structure fixée, le reste du problème de configuration revient à un CSP standard, où toutes les variables sont connues et à partir duquel on peut appliquer les méthodes classiques d'élimination de symétries.

Les symétries des CSP standards sont des bijections sur l'ensemble des affectations de variables (une variable x est affectée à une valeur v) qui préserve l'ensemble des solutions [4]. D'elles découlent naturellement deux sous-catégories : les symétries de variables et/ou de valeurs. Dans les CSP standards, toutes les variables, leurs domaines et leurs contraintes sont connues dès le départ et donc aussi leur ensemble de symétries (leur groupe d'automorphisme). Bien que certaines approches traitent du changement du groupe d'automorphisme au fur et à mesure (quand plusieurs variables affectées induisent un sous-problème), aucune ne correspond au type de changement auquel on a affaire en configuration.

Les symétries en configuration ajoutent la dimension de décider si l'ajout d'un composant, ainsi que l'ensemble des variables qui constituent ses attributs, doit être fait ou pas afin d'éviter de générer des structures isomorphes. Le traitement de ces symétries structurelles diffère significativement par nature de celle concernant ses affectations de variables. Dans ce dernier cas, on utilise le groupe d'automorphisme (ses symétries "internes") pour éviter des instanciations redondantes. Dans le premier cas, on utilise son groupe d'isomorphisme (ses symétries "externes") pour éviter de générer des structures redondantes.

La contribution de cet article est une amélioration significative des travaux dans [10]. Le test de canonicité peut être grandement simplifié en exploitant mieux les propriétés de la définition de la canonicité. L'idée principale consiste à exploiter le fait que lorsqu'une extension canonique survient, seul un nombre limité d'opérations est né-

cessaire pour vérifier la canonicité, tout simplement parce que l'arbre étendu reste en grande partie inchangé. La méthode que nous présentons réduit le surcoût du test de canonicité encore plus que nous le pensions tout en permettant à un filtrage d'avoir lieu.

Nous présentons des expérimentations non seulement sur les problèmes simplifiés que nous avions étudié dans [10] mais aussi sur deux problèmes classiques : le problème des "racks" et le problème de Vellino [11].

Pour autant que nous le sachions, le filtrage n'a jamais été mis en oeuvre lors de l'énumération canonique de configuration.

Plan de l'article

La section 2 introduit les définitions, les concepts et le contexte minimal qui seront utiles par la suite. La section 3 présente les hypothèses de base et le théorème qui fonde la contribution de l'article. En section 4, une nouvelle fonction de filtrage et de test de canonicité de structures arborescentes de configuration est présentée. On y prouve que sa complexité est une fonction linéaire de la taille de l'arbre et on la compare avec d'autres techniques d'élimination de symétries utilisées dans le cadre des CSP standards. Les résulats expérimentaux sont présentés en section 5. La section 6 contient la conclusion.

2 Definitions

Un *problème de configuration* décrit un produit générique sous la forme d'énoncés déclaratifs (règles ou axiomes) portant sur la bonne formation d'un produit. Une *configuration* est une instance valide de problème. Par exemple[1], nous pouvons devoir configurer un immeuble de F étages, chacun possédant R pièces, chacune d'elles ayant d'une à trois portes et au maximum quatre fenêtres (cf figure 1). Dans cet exemple, nous ne mentionnons pas les attributs et contraintes nécessaires (ex : dimension et position des portes et fenêtres, contraintes de non recouvrement) qui permettraient de définir complétement un vrai problème pour nous focaliser sur le sous-problème consistant à déterminer la structure d'une solution en termes de composants et de façon dont ils sont liés. En effet, une fois la structure choisie, ce qu'il reste à faire revient à résoudre un CSP standard en affectant des valeurs à des attributs contraints, phase qu'on peut accélérer grâce à des techniques traditionnnelles d'élimination de symétries.

Les configurations mettent en jeu des objets interconnectés, comme illustré en figure 2, où l'existence d'isomorphismes structurels apparaît clairement.

[1] Utilisé plus tard dans les expérimentations.

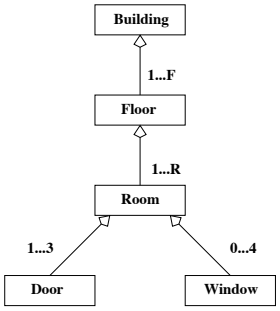

FIG. 1 – Une modélisation de la structure d'un immeuble

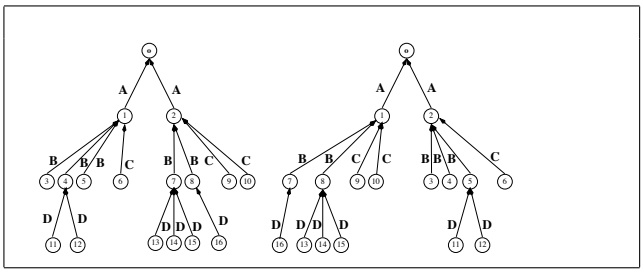

FIG. 2 – Deux structures arborescentes isomorphes.

2.1 Sous-problèmes structurels

A partir du problème de configuration dans sa généralité, nous isolons ce qu'on appelle le *sous-problème structurel*, qui consiste à déterminer une structure de la solution à partir des relations binaires entre les différents types d'objets disponibles et en fonction des contraintes structurelles qui s'appliquent sur ces relations.

Nous considérons un ensemble totalement ordonné d'objets O (on utilisera en pratique $O = \{1, 2, \ldots\}$), un ensemble totalement ordonné de types T_C et un ensemble totalement ordonné R_C de relations binaires sur O, tels que :

$$\forall R_1, R_2 \in R_C, \forall o_1, o_2, o_3 \in O, R_1(o_1, o_2) \Rightarrow \neg R_2(o_1, o_3)$$

Les conditions ci-dessus interdisent à un objet d'apparaître plus d'une fois dans une structure de configuration, ce qui fait de cette structure un arbre[2]. Nous notons $\prec_O$, $\prec_{T_C}$ et $\prec_{R_C}$ les ordres totaux correspondants.

Définition 1 (syntaxe) *Un problème* structurel *est défini par un quadruplet (t, T_C, R_C, C), où $t \in T_C$ est le type de l'objet configuré (ou type racine de la configuration) et C est un ensemble de contraintes structurelles portant sur les éléments de T_C et R_C (types des composants, cardinalités notamment).*

Définition 2 (sémantique) *Une* instance *d'un problème structurel (t, T_C, R_C, C) est une interprétation I associée*

[2] Cette condition est forte mais elle se rencontre dans un nombre significatif de problèmes de configurations. Le choix de R_C est facile.

à t et aux éléments de T_C et R_C, construite sur le domaine O des objets. Si cette interprétation satisfait les contraintes de C, c'est une solution *du problème structurel.*

Dans le cas général, on peut représenter une configuration par un DAG (directed acyclic graph) coloré G=(t,X,E,L) avec X $\subset O$, E $\subset O \times O$ et L $\subset O \times T_C$. t est le type racine (dont l'unique instance sert de racine à la structure), X est l'ensemble des sommets, E est l'ensemble des arcs et L est la fonction qui associe chaque sommet à un type (une couleur), comme l'illustre la figure 2. Mais grâce aux hypothèses formulés plus haut, nous n'avons ici affaire qu'à des arbres. Leur structure étant arborescente, les configurations peuvent tout aussi bien être représentées par des arbres colorés (en leurs sommets) que nous appelons T-arbres (figure 3).

Pour faciliter les comparaisons, nous utilisons les mêmes définitions et notations que dans [10], que nous rappelons en partie afin de rendre l'article auto-suffisant.

Définition 3 (T-arbre) *Un* T-arbre *est un arbre fini non vide dont les nœuds sont étiquetés par T_C. On note $(T, \langle c_1, \dots c_k \rangle)$ le T-arbre dont les sous-arbres sont $c_1, \dots c_k$ et l'étiquette de la racine est T.*

2.2 Isomorphismes

Tester l'isomorphisme de deux graphes est un problème de NP qui n'a toujours pas été ni prouvé NP-complet ni prouvé polynomial. Pour certains types de graphes, les arbres ou les graphes de degré borné par une constante, le test d'isomorphisme est polynomial [13]. Les T-arbres, structures arborescentes des configurations, étant des arbres, ils sont isomorphes, égaux et superposables sous les mêmes conditions que les arbres standards. Une *classe d'isomorphisme* représente l'ensemble de tous les isomorphes d'un graphe. Tous les graphes d'une classe d'isomorphisme sont équivalents. En conséquence, une procédure de génération de graphes ne doit idéalement générer qu'un représentant, dit *canonique*, par classe.

Il est montré dans [10] que disposer d'un test efficace de canonicité d'un graphe (ne serait-ce même qu'un arbre) n'est pas suffisant pour traiter les symétries structurelles en configuration. La façon dont on définit le test de canonicité détermine la possibilité (ou non) à une procédure de génération de structures par extension unitaire (cf définition dans la section suivante) de backtracker dès qu'elle détecte une structure non canonique. Un exemple d'une telle procédure est donnée dans [9].

3 Génération de structures sans isomorphes

Afin d'isoler un représentant canonique pour chaque classe d'équivalence des T-arbres, nous définissons un ordre total sur les T-arbres (illustrée en figure 3). La relation $\preccurlyeq$ est l'ordre total qui généralise $\prec_{T_C}$ aux T-arbres. $\preccurlyeq_{lex}$ est la généralisation lexicographique de $\preccurlyeq$ sur des listes de T-arbres.

Définition 4 (La relation $\preccurlyeq$) *Etant donnés deux T-arbres C = (T, L) et C' = (T', L'), $\preccurlyeq$ est définie récursivement par : C $\preccurlyeq$ C' ssi T $\prec_{T_C}$ T' ou T = T' et L $\preccurlyeq_{lex}$ L'.*

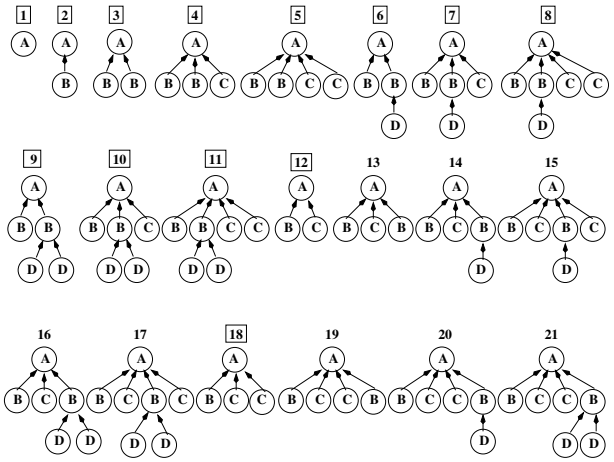

FIG. 3 – T-arbres ordonnés selon $\preccurlyeq$. L'index de chaque représentant $\preccurlyeq$-minimal est encadré. Au plus deux D peuvent se connecter à un B, tout comme les B et les C vis-à-vis d'un A.

Nous définissons récursivement les T-arbres canoniques ainsi (dans la perspective d'une procédure de test) :

Définition 5 (Canonicité d'un T-arbre) *Un T-arbre C = (T, L) est* canonique *ssi L est vide ou si L est trié selon $\preccurlyeq$ et chaque c de L est lui-même canonique.*

Il a été montré que les T-arbres canoniques définis ainsi sont les représentants $\preccurlyeq$-minimaux de leur classe d'isomorphisme.

L'opération élémentaire de génération de T-arbres est appelée *extension unitaire*.

Définition 6 (Extension unitaire, canonique) *Nous appelons* extension unitaire*, l'opération d'ajout d'une feuille à la fin d'une liste de sous-arbres dans un T-arbre C. Si en plus C ainsi que son extension sont canoniques, l'opération est appelée* extension unitaire canonique*.*

Le but d'une procédure de recherche constructive est de générer par extension unitaire tous les T-arbres à partir de $(t, \langle\rangle)$ (rappel : t est le type racine de toute configuration) qui respectent les contraintes (structurelles ou autres) du problème. L'élimination des structures isomorphes revient à ne générer que les solutions canoniques. Notre définition de canonicité choisie assure que tout T-arbre canonique

peut être atteint par une suite d'extensions unitaires canoniques, ce qui donne au test de canonicité une importance centrale.

Proposition 1 *Soit C un T-arbre. Soit C' un T-arbre résultant d'une extension unitaire de C. On a $C \preccurlyeq C'$.*

Preuve 1 *Quelle que soit la profondeur considérée dans C et C', il y a un sous-arbre (T, L) dans C et un sous-arbre (T, L') dans C' qui font que C et C' diffèrent. Si le noeud a été placé à la fin de L (pour former L') alors $L \preccurlyeq_{lex} L'$ donc (T, L) $\preccurlyeq$ (T, L'). Sinon, il existe une seule position i où L et L' différent et contiennent respectivement un sous-arbre c_i et un sous-arbre c'_i. Par induction, on a $c_i \preccurlyeq c'_i$ qui implique que (T, L) $\preccurlyeq$ (T, L'), et finalement que C $\preccurlyeq$ C'.*

Nous savons grâce à la proposition 1 qu'ajouter un sommet à un T-arbre ne peut le rendre inférieur selon $\preccurlyeq$ à son prédécesseur dans la liste de T-arbres auquel il appartient. Lorsqu'on vérifie incrémentalement la canonicité du T-arbre étendu, cela permet de n'avoir à comparer chacun de ces sous-arbres modifiés qu'avec leur voisin de droite, et pas celui de gauche.

fonction **CompareT-arbres**(C, C')
in : $C = (T, L)$ et $C' = (T', L')$,
out : EGAL si $C = C'$,
INF si $C \neq C'$ et $C \preccurlyeq C'$,
SUP dans les autres cas,

si $T < T'$ **alors retourne** INF
si $T' < T$ **alors retourne** SUP
si $L = \langle\rangle$ **et** $L' = \langle\rangle$ **alors retourne** EGAL
si $L = \langle\rangle$ **alors retourne** INF
si $L' = \langle\rangle$ **alors retourne** SUP
soit $L = \langle a_1, \ldots, a_j \rangle$,
soit $L' = \langle b_1, \ldots, b_k \rangle$,
pour $i := 1$ **à** j **faire**
  si $k < i$ **alors retourne** SUP,
  $result$:=CompareT-arbres(a_i, b_i),
  si $result \neq egal$ **alors**
  retourne $result$,
retourne EGAL,

FIG. 4 – La fonction **CompareT-arbres**

4 Test de canonicité amélioré

L'algorithme le plus simple, déjà présenté dans [10], est de complexité pseudo-linéaire (O(n log n)) en la taille de l'arbre et exploite directement la définition de la canonicité en utilisant une fonction pour comparer les T-arbres, rappelée en figure 4. Schématiquement, cet algorithme teste que tous les sous-arbres dans le T-arbre sont récursivement triés selon $\preccurlyeq$, c'est-à-dire qu'ils sont tous supérieurs ou égaux selon $\preccurlyeq$ à leur voisin de gauche (lorsqu'il existe). Or, nous pouvons améliorer cet algorithme en exploitant la proposition 1 pour faire du filtrage et utiliser le fait que les T-arbres sont construits incrémentalement.

4.1 Filtrage

La proposition 1 mène directement à un algorithme de filtrage. En considérant toutes les positions dans le T-arbre où peut s'ajouter le sommet de l'extension unitaire, nous voyons que puisque les T-arbres générés sont canoniques, par conséquent lexicographiquement triés en interne, il n'est possible d'exécuter aucun ajout de sommet à l'intérieur d'un sous-arbre identique à son voisin de droite (puisqu'il deviendrait plus grand selon $\preccurlyeq$, et le T-arbre serait non canonique). Une nouvelle procédure de recherche peut en tirer profit en excluant ces choix pendant la recherche.

Il suffit de compléter la fonction `compareT-arbres` afin que, à chaque fois qu'elle compare deux sous arbres, elle mémorise qu'il y a égalité. Cette opération se fait en temps constant. Dès lors, la procédure de génération d'arbres canoniques n'a plus qu'à tester si un sous-arbre à étendre unitairement est égal à son voisin de gauche, uniquement en lisant (en temps constant) l'information stockée précédemment. Si c'est le cas, on backtracke.

Dans le cas où la procédure d'énumération serait reliée à un système CSP standard pour générer des configurations impliquant des variables d'attribut ou d'autres relations, ce filtrage permet de fermer le port des variables représentant les relations, et permet par conséquent d'autres propagations.

4.2 Canonicité incrémentale

Quand nous testons la canonicité d'un T-arbre nouvellement généré, nous savons que seuls un nœud et une arête ont été ajoutés dans un arbre qui était à l'origine canonique (condition pour continuer). La fonction en figure 5 détaille l'algorithme qui peut évaluer la canonicité du T-arbre obtenu par extension unitaire d'un T-arbre précédemment canonique.

fonction **IncrementalCanonical**(C, N)
in : un T-arbre $C = (T, L)$, N un noeud nouvellement ajouté dans C
out : VRAI si C est canonique, FAUX sinon,

pour $(n = N\,;\, n \neq C\,;\, n = parent(n))$ {
  si (**CompareT-arbres(**$n, right(n)$**)**==SUP)
    retourne FAUX
}
retourne VRAI,

FIG. 5 – Test incrémental de canonicité

Seuls les sous-arbres enracinés en un ancêtre du noeud nouvellement ajouté doivent être examinés. La procédure commence à partir du noeud nouvellement ajouté, en remontant vers le haut du T-arbre, et effectue des comparaisons à chaque niveau. À chaque invocation de `compareT-arbres`, la structure de données est mise à jour en temps constant en enregistrant le fait que le premier sous-arbre est identique au deuxième (ie, égal selon $\preccurlyeq$).

Proposition 2 *Soit C un T-arbre canonique. Soit C' un T-arbre résultant d'une extension unitaire de C. Tester la canonicité de C' a une complexité linéaire en la taille de C'.*

Preuve 2 *Dans le pire des cas, il faut faire une comparaison entre deux sous-arbres à chacune des profondeurs d'un T-arbre de taille N et de profondeur d. Cette comparaison est de complexité linéaire en fonction de la taille du plus petit des deux sous-arbres. Donc, dans le pire des cas, ces deux sous-arbres sont identiques et il n'y a que ces deux sous-arbres à cette profondeur. Donc, l'arbre correspondant au pire des cas est un arbre binaire complet. La complexité du test incrémental de canonicité est donc $\Sigma_{i=0}^{d}(N/2^i)$ qui converge vers (mais reste inférieur à) 2N quand d croît.*

4.3 Travaux connexes en CSP et configuration

Si on accepte de borner la taille des solutions d'un problème de configuration, on peut évidemment le résoudre en utilisant des techniques CSP standards après une phase de traduction. Ainsi, le traitement des symétries peut être réalisé en utilisant des techniques CSP connues [2, 6, 7, 8, 5]. Mais, utiliser de telles traductions peut être extrêmement et inutilement consommateur de ressources. Par exemple, si un bâtiment a au plus 50 étages avec au plus 20 pièces dont chacune a au plus 5 fenêtres et 3 portes, nous devons traiter un CSP ayant au moins 8000 variables, sans parler de l'augmentation de l'ensemble des contraintes de symétries posées par des techniques comme SBDD, SBDS ou DLC (Dynamic Lex Constraint). Cependant les contraintes du problème peuvent être telles que beaucoup moins de 8000 fenêtres et portes apparaissent dans les solutions. C'est pourquoi les problèmes de configuration sont habituellement résolus en utilisant des variantes du formalisme CSP. Par exemple, les CSP composites [16] gèrent l'aspect dynamique en introduisant dynamiquement des fragments de CSP pendant la recherche. Les DCSP [17] exploitent des variables d'activité et des règles d'activation pour contrôler comment la génération de la structure configurée introduit de nouveaux éléments. Ces méthodes ajoutent ou activent des variables pendant la résolution chaque fois qu'un nouvel objet est inséré, relié ou activé. La technique d'élimination de symétries que nous proposons vise précisément à empêcher des extensions inutiles de la structure.

Néanmoins, il reste intéressant de comparer notre méthode à celles employées pour résoudre les CSP standards. Pour ce faire, il faut modéliser le problème de génération de structures de l'objet à configurer sous la forme d'un CSP. Nous le ferons sur un petit exemple, où un objet P peut se composer de 2 composants maximum de type A, eux-mêmes composés de 2 composants maximum de type B. Une façon très efficace de modéliser cet exemple sous la forme d'un CSP est de considérer l'ensemble de variables {B1, B2, B3, B4, A1, A2 } représentant les composants qui peuvent apparaître. Ainsi les variables Ai prennent leur valeur dans {0, p} et les variables Bi dans {0, a_1, a_2}. A1=0 signifie qu'A1 n'est pas utilisé. A2=p signifie qu'A2 est un composant de P. B3=a_2 signifie que B3 est un composant d'A2. L'ensemble de symétries du CSP sont toutes les compositions de permutations entre (1) les variables Ai, (2) les variables Bi ou (3) les valeurs a_1 et a_2. La partie gauche de la figure 6 montre l'arbre de recherche de toutes les solutions canoniques du CSP générés grâce à la méthode SBDS. La partie droite de la figure 6 montre l'arbre de recherche du problème structurel équivalent résolu par notre méthode (les T-arbres sont générés par des extensions unitaires canoniques).

SBDS pose des contraintes conditionnelles à chaque fois qu'un noeud de l'arbre est généré afin d'interdire la génération de noeuds correspondant à des instanciations partielles symétriques. Or, le test d'une telle contrainte est de complexité linéaire en fonction de la taille de l'instanciation. De plus, il y a une contrainte à tester par symétrie. Par contre, notre test incrémental de canonicité est linéaire dans le pire des cas et il n'est effectué qu'une fois par structure générée (par incrémentation unitaire d'une structure précédente). Or, il y a moins de structures générées que de noeuds dans l'arbre de recherche du CSP. Cette comparaison vaut aussi pour les méthodes SBDD et DLC car, bien qu'elles génèrent un autre arbre de recherche, le nombre de noeuds reste supérieur et le test de symétrie en chaque noeud est de complexité similaire à celle de SBDS. Notre méthode constitue donc un moyen extrêmement économique d'empêcher la génération des structures non canoniques.

Par ailleurs, ce travail prend la suite de plusieurs contributions plus anciennes relatives à l'élimination de symétries dans les problèmes de configuration (comme par exemple [14] qui s'attaque à des cas plus restreintes : interchangeabilité des objets inutilisés, utilisation des cardinalités au lieu des objets quand ils restent interchangeables).

5 Résultats expérimentaux

Nous avons effectué une série de tests avec trois méthodes d'élimination d'isomorphismes : celle qui teste basiquement la canonicité ("sans iso") et celles que nous avons introduites dans le papier : le test de canonicité al-

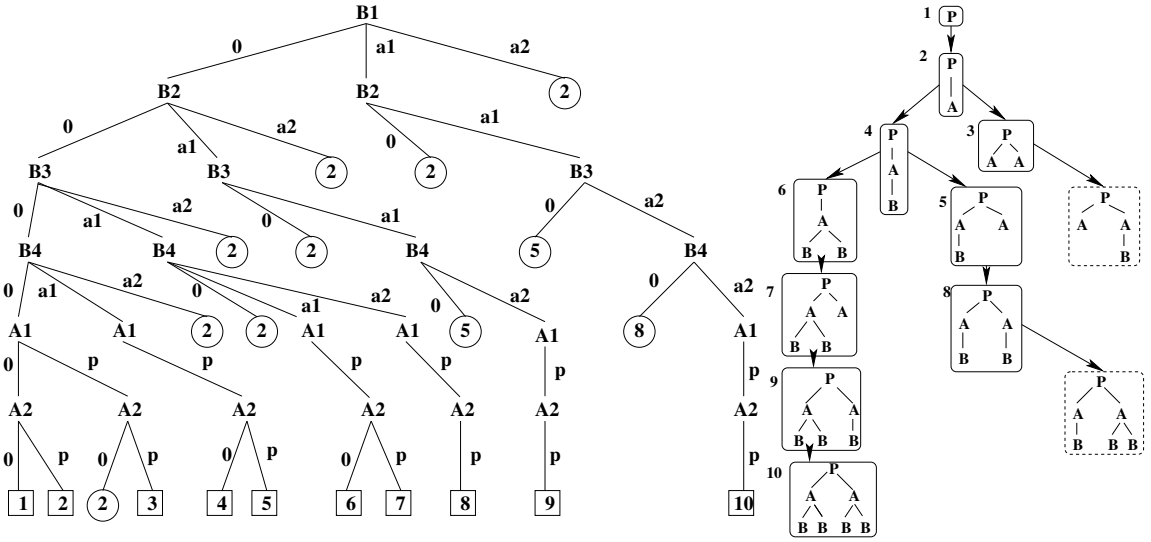

FIG. 6 – A gauche, l'arbre de recherche du CSP modélisant une structure P composée d'au plus 2 A, eux-mêmes composés d'au plus 2 B. Un numéro encadré indique que la feuille est une solution. Un numéro i cerclé indique que l'instanciation est symétrique à celle du i^e noeud du même niveau dans l'arbre (et donc qu'on a backtracké). On suppose que les domaines sont filtrés en fonction des contraintes structurelles : quand une valeur a_i a été affectée, on retire la valeur 0 de la variable Ai, et si a_i a été affectée deux fois, elle est retirée du domaine des Bi restants. A droite, un arbre de recherche du problème structurel équivalent. Le numéro d'une structure correspond au numéro de la solution du CSP équivalent. Les structures entourées de pointillés sont non canoniques. On y backtracke.

légé qui tient compte de l'incrémentalité de la canonicité ("sans iso rapide") et la même avec le filtrage en plus ("sans iso rapide + filtrage"). Nous donnons les temps de résolution (obtenus par un programme Java exécuté sous Linux sur un PC à 2.4 gigahertz), le nombre de T-arbres générés, et le nombre d'appels à la fonction de comparaison.

Nos premières expérimentations ont été réalisées sur le problème de configuration d'immeuble illustré en figure 1. Afin d'explorer les propriétés combinatoires du problème, nous avons fait varier les paramètres suivants : F (le nombre maximum d'étages dans un immeuble) et R (le nombre maximum de pièces dans un étage). Nous avons écrit un configurateur en Java qui génère toutes les solutions possibles du problème de configuration d'immeuble en fonction des paramètres F et R. Nous n'effectuons pas de comparaison avec l'énumération des solutions du cas où on ne supprimerait pas les structures non canoniques tout simplement parce que les temps de résolution seraient trop longs. Par exemple, avec $F = 1$ et $R = 3$, il faut 712ms mais avec $F = 1$ et $R = 3$ on passe déjà à 40s.

Nous constatons que la version incrémentale du test de canonicité permet de réduire énormément le nombre d'appels à `compareT-Trees` et de diminuer significativement les temps de résolution. Le filtrage supprime une partie non négligeable de T-arbres et améliore encore les temps de résolution. Les meilleurs résultats surviennent pour les problèmes de grande taille, pour lesquels les 2 améliorations cumulées proposées dans cet article permettent de diviser les temps par plus que 2.

Nous avons également résolu deux problèmes classiques en configuration : le problème des "racks" (problème 031 de la CSPLib [3]) et le problème de Vellino [11]. Ce sont des problèmes d'optimisation : le coût des racks choisies et respectivement le nombre de conteneurs doivent être réduits au minimum. Nous avons ajouté à un algorithme de Branch and Bound de génération de structures des tests de contraintes définies par ces problèmes (celles qui ne sont pas directement liées à la structure des solutions). Pour les racks : le nombre limité de racks disponibles, la puissance qu'un rack peut fournir supérieure à la somme de puissances que ses cartes connectées exigent, etc. Pour le problème de Vellino : contraintes de compatibilité entre composants, nombre maximum de composants contenus dans chaque type de conteneur, etc. Nous avons seulement mis en oeuvre une approche minimaliste de traitement de ces contraintes, en les testant après chaque extension unitaire de structure. Mettre en oeuvre des techniques de filtrage habituelles (Forward-Checking (FC), MAC) pourrait aider à réduire les temps de résolution (avec ou sans suppression d'isomorphisme). Cependant, seul nous intéressait la mesure de l'impact de la suppression d'isomorphisme. Ajouter du filtrage de domaines ne changerait pas la comparaison car ces deux mécanismes d'accélération de

[3] http ://4c.ucc.ie/ tw/csplib/prob/prob031/

(F, R)	#sol	sans iso			sans iso rapide		sans iso rapide + filtrage		
		#arbres	#appels	temps	#appels	temps	#arbres	#appels	temps
(1, 6)	54×10^3	111×10^3	2.5×10^6	0.54s	233×10^3	0.43s	84×10^3	180×10^3	0.34s
(1, 7)	170×10^3	358×10^3	9.8×10^6	1.9s	721×10^3	1.5s	262×10^3	541×10^3	1.1s
(1, 8)	490×10^3	1.0×10^6	33×10^6	5.6s	2.0×10^6	4.3s	746×10^3	1.5×10^6	3.2s
(1, 9)	1.3×10^6	2.8×10^6	105×10^6	16s	5.3×10^6	11.6s	2.0×10^6	3.8×10^6	8.2s
(1, 10)	3.2×10^6	7.2×10^6	303×10^6	42s	13×10^6	29s	4.9×10^6	9.3×10^6	20s
(2,3)	334×10^3	645×10^3	15×10^6	3.2s	3.4×10^6	2.7s	530×10^3	3.1×10^6	2.2s
(2, 4)	7.5×10^6	15×10^6	511×10^6	85s	79×10^6	63s	12×10^6	72×10^6	51s
(2, 5)	120×10^6	245×10^6	2.0×10^9	1459s	1.3×10^9	1071s	187×10^6	1.2×10^9	835s

TAB. 1 – Nombre de solutions, de T-arbres, d'appels à `compareT-arbres` et temps obtenus quand on varie les valeurs de F et R avec ou sans incrémentalité dans le test de canonicité et avec ou sans filtrage.

la recherche sont orthogonaux : si le filtrage supprime une structure non canonique, il supprime également ses isomorphes. Le filtrage élimine soit complètement une classe d'isomorphisme soit la laisse intacte. Le facteur de gain que nous avons obtenu devrait donc être équivalent si on ajoute des techniques de filtrage.

Pour autant que nous le sachions, les résultats expérimentaux les plus récents concernant le problème des racks sont donnés dans [12] (inst1 : 0.34s, inst2 : 3.7s, inst3 : 45s). Les auteurs relient deux modèles duals du problème sous la forme de CSP classiques en utilisant des "channeling constraints". Ils ne résolvent pas l'instance 4. Nous n'avons trouvé aucune solution de l'instance 4 nulle part. La voici : coût = 1150 avec un R250 contenant un C150 et un C100, un R300 contenant un C100, deux C75 et un C50, un R300 contenant trois C50, trois C40 et un C20, et un R300 contenant trois C40 et huit C20. Ces bons résultats (cf table 2) sont aussi dus à notre fonction de maximisation du coût restant qui permet au Branch and Bound de backtracker. Cette fonction très précisément définie est proche du coût exact d'une solution optimale et permet d'élaguer presque tout l'espace de recherche dès lors qu'une solution optimale a été rencontrée.

6 Conclusion

Notre approche traite de l'élimination de symétries des structures des problèmes de configuration avec un algorithme spécialisé. Celui-ci garantit que seule l'exacte quantité nécessaire de ressources (en mémoire et donc en temps) est utilisée et permet un surcoût de traitement extrêmement bas. Par exemple, dans ce cadre, l'apparition de strutures isomorphes peut être détecté à l'avance pour ne pas les générer. Ce travail exploite la nature incrémentale de la génération canonique de configurations tout en introduisant du filtrage, ce qui permet d'obtenir une amélioration significative de la complexité de notre ancienne procédure. Nous voyons que les résultats expérimentaux obtenus ici sur différents problèmes montrent des accélérations significatives alors que la taille de ces problèmes reste réduite, ce qui laisse augurer que ce gain soit encore accru pour des problèmes de taille supérieure. Ces travaux peuvent s'étendre automatiquement aux cas où la structure est un graphe. En effet, dans [9], le concept de pseudo-canonicité s'appuie sur la canonicité de l'arbre couvrant de la structure.

De plus, la génération de structures de configurations est destinée à être couplée avec un configurateur ou un solveur de contraintes. Dans ce cas, notre filtrage pourrait permettre d'établir l'impossiblité de modifier les relations internes entre composants qu'on ne peut plus modifier (à cause du filtrage). Ceci peut avoir comme conséquence la propagation de contraintes supplémentaires dans le reste du problème. Cet aspect est actuellement à l'étude.

Références

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps—-application to configuration. *Artificial Intelligence*, 135(1-2) :199–234, 2002.

[2] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *Principles and Practice of Constraint Programming*, pages 73–87, 1999.

[3] Virginia Barker, Dennis O'Connor, Judith Bachant, and Elliot Soloway. Expert systems for configuration at digital : Xcon and beyond. *Communications of the ACM*, 32 :298–318, 1989.

[4] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3) :115–137, 2006.

[5] Jean François Puget. Dynamic lex constraints. In *In Benhamou, F., ed., Principles and Practice of. Constraint Programming - CP 2006, LNCS 4204*, page 453 ?467, 2006.

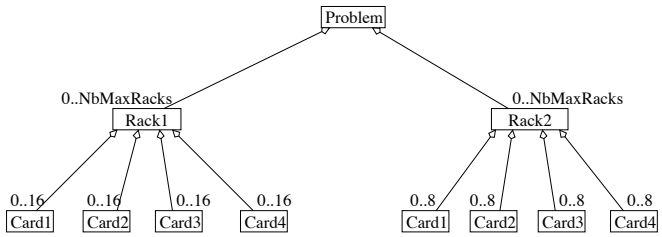

FIG. 7 – Hiérarchie des composants du problème des racks (les flèches représentent les relations de composition). Une instance de problème implique jusquà `NbMaxRacks` racks de type 1 ou 2 nécessaires pour connecter des cartes de différents types. Les racks de type 1 peuvent être connectées avec au plus 16 cartes, les racks de type 2 avec au plus 8 cartes.

	tout		sans iso			sans iso rapide		sans iso rapide + filtrage		
Instance	#arbres	temps	#arbres	#appels	temps	#appels	temps	#arbres	#appels	temps
inst1	2424	55ms	1647	6849	51ms	3280	51 ms	1645	3263	51ms
inst2	237×10^6	1732s	4.9×10^3	71×10^3	36s	12.2×10^6	33s	4.8×10^6	11.9×10^6	33s
inst3	683	66ms	683	10×10^3	66ms	1.4×10^3	63ms	683	1.4×10^3	63ms
inst4	821×10^3	5.2 s	237×10^3	1.8×10^6	1.8s	526×10^3	1.6s	237×10^3	525×10^3	1.6s

TAB. 2 – Les quatre instances du problème des racks (CSPLib 031).

[6] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96 : Principles of Knowledge Representation and Reasoning*, pages 148–159, San Francisco, California, 1996. Morgan Kaufmann.

[7] I. P. Gent and B. M. Smith . Symmetry breaking in constraint programming. In *in Horn W., ed., 14th European Conference on Artificial Intelligence, Berlin, ECAI 2000, IOS press*, pages 599–603, 2000.

[8] I. P Gent, W. Harvey, and T. Kelsey. Groups and constraints : Symmetry breaking during search. In *in Hentenryck,P. ed., Principles and Practice of. Constraint Programming - CP 2002 - LNCS 2470*, pages 415–431, 2002.

[9] Laurent Henocque, Mathias Kleiner, and Nicolas Prcovic. Advances in polytime isomorph elimination for configuration. In *proceedings of Principles and Practice of Constraint Programming - CP 2005,*, pages 301–313, Sitges Barcelona, Spain, 2005. Springer.

[10] Laurent Henocque and Nicolas Prcovic. Practically handling configuration automorphisms. In *proceedings of the 16th IEEE International Conference on Tools for Artificial Intelligence*, Boca Raton, Florida, November 15–17 2004.

[11] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in opl. In *Principles and Practice of Declarative Programming*, 1999.

[12] Z. Kiziltan and B. Hnich. Symmetry breaking in a rack configuration problem. In *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints, Seattle*, 2001.

[13] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. System Sci.*, 25 :42–49, 1982.

[14] Daniel Mailharro. A classification and constraint-based framework for configuration. *AI in Engineering, Design and Manufacturing, (12)*, pages 383–397, 1998.

[15] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, Boston, MA, 1990.

[16] Daniel Sabin and Eugene C. Freuder. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[17] Timo Soininen, Ilkka Niemela, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming : Towards Efficient and Scalable Knowledge*, pages 195–201, March 2001.

[18] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2) :111–125, June 1997.

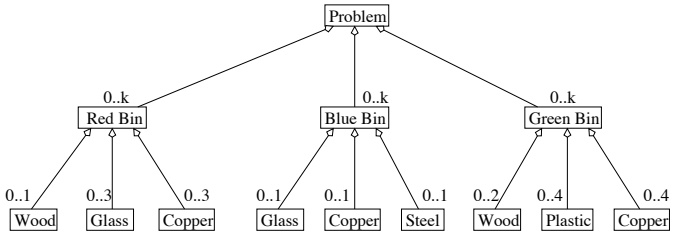

FIG. 8 – La hiérarchie de composants du problème de Vellino. k composants doivent être placés dans les conteneurs donc il y en a maximum k de chaque type. La définition du problème inclut un certain nombre de contraintes : les conteneurs rouges ne peuvent contenir ni plastique ni acier, les conteneurs bleus ni bois ni plastique, les conteneurs verts ni acier ni verre. Les conteneurs rouges peuvent contenir jusqu'à trois composants dont au plus un est en bois. Les conteneurs bleus ne peuvent contenir qu'un composant. Les conteneurs verts ne peuvent contenir plus de quatre composants dont au plus deux sont en bois.

Instances					tout		sans iso			sans iso rapide		sans iso rapide + filtrage		
G	P	S	W	C	#arbres	temps	#arbres	#appels	temps	#appels	temps	#arbres	#appels	temps
1	2	1	3	2	32×10^3	462ms	14×10^3	74×10^3	179ms	60×10^3	174ms	12×10^3	55×10^3	166ms
2	2	1	3	2	105×10^3	1.4s	40×10^3	446×10^3	442ms	171×10^3	429ms	33×10^3	156×10^3	403ms
2	4	1	3	2	195×10^3	3.1s	65×10^3	458×10^3	815ms	292×10^3	787ms	55×10^3	266×10^3	732ms
2	4	3	3	2	6.9×10^6	108s	715×10^3	6.1×10^6	7.3s	2.5×10^6	6.7s	517×10^3	2.2×10^6	6.0s
2	4	3	6	2	119×10^6	1958s	4.0×10^6	48×10^6	38s	15×10^6	33s	2.8×10^6	12×10^6	28s
2	4	3	6	4	533×10^6	8615s	13×10^6	177×10^6	151s	48×10^6	124s	9.8×10^6	41×10^6	111s

TAB. 3 – Résultats pour le problème de Vellino. Les cinq premières colonnes indiquent les besoins en verre (Glass), plastique (Plastic), acier (Steel), bois (Wood) et cuivre (Copper).