Une procédure générale d'élimination d'isomorphismes pour les problèmes de configuration

Laurent Hénocque, Mathias Kleiner, Nicolas Prcovic

Une procédure générale d'élimination d'isomorphismes pour les problèmes de configuration

Laurent Hénocque Mathias Kleiner Nicolas Prcovic
LSIS - Université Paul Cézanne - Aix-Marseille III
laurent.henocque,mathias.kleiner,nicolas.prcovic@lsis.org

Résumé

Une difficulté intrinséque à la résolution de problèmes de configuration réside dans l'existence de nombreux isomorphismes structurels dans les solutions. Nous définissons deux procédures de recherche permettant la suppression de grandes portions de l'espace de recherche dont on montre qu'elles ne renferment que des solutions non canoniques. On y parvient grâce à un test en chaque noeud de l'arbre de recherche de complexité temporelle linéaire. Nous présentons des résultats sur un exemple de configuration simple mais représentatif de ce qu'on pourra obtenir sur des problèmes réels.

Abstract

An inherent difficulty in solving configuration problems is the existence of many structural isomorphisms. We define two search procedures allowing the removal of large portions of the search space that provably solely contain non canonical solutions. The tests performed on each node are time polynomial. Experimental results are reported on a simple yet realistic configuration example.

1 Introduction

Configurer consiste à simuler la réalisation d'un produit complexe à partir de composants choisis dans un catalogue de types, en s'appuyant sur les relations connues entre les différents types de composants et en faisant des choix d'instanciation pour leurs différents attributs. Le besoin industriel de configurateurs est ancien[9, 2] et a conduit au développement de nombreux programmes de configuration et à la définition de divers formalismes [2, 12, 15, 1, 14, 16, 17, 8].

Notre objectif principal est de réduire l'explosion combinatoire superflue due à la présence d'isomorphismes dans la recherche de configurations. Nous nous intéressons précisément à la nature dynamique de la configuration, qui consiste à générer les structures des solutions potentielles[1]. Ce problème est l'un des plus importants à résoudre dans le cadre de la configuration, si on veut résoudre des problèmes pratiques dont la modélisation structurelle permet une réelle variabilité de la structure d'une solution. Actuellement, les solveurs ne sont pas capables de déterminer de solutions à ces problèmes en temps raisonnable à cause du nombre exponentiel d'isomorphes qu'ils génèrent inutilement pour chaque structure de solution.

Nous proposons une procédure générale de recherche de solutions qui élimine efficacement une grande partie des solutions isomorphes des problèmes de configuration. Plus précisément, nous présentons une procédure dédiée aux structures d'arbres, qui est complète, non redondante et élimine efficacement toute (sous-)structure non canonique. A partir de cette procédure, nous montrons comment en obtenir une autre pour générer tout type de structure sans restriction, de façon complète et non redondante, en éliminant un nombre conséquent de structures isomorphes. Ces travaux utilisent des résultats obtenus dans [4] et [6] qui donnaient seulement des conditions nécessaires d'existence de telles procédures.

Plan de l'article

La section 2 décrit le formalisme utilisé tout au long de l'article, la notion de *sous-problème structurel* et la problématique liée à leur génération. La section 3 présente une procédure de génération de solutions des sous-problèmes structurels lorsqu'ils ont une structure d'arbre, en démontrant qu'elle est complète tout en

[1] Ce qui nous place en aval du cadre des CSP, car ceux-ci ont une structure (un ensemble de variables) qui est déjà donné.

évitant tout isomorphisme. La section 4 étend cette procédure pour qu'elle traite tout type de problème de configuration. Dans ce cadre, il n'y a qu'une portion des solutions isomorphes qui sont éliminées. En section 5, on raffine la procédure pour qu'elle rejette encore plus de solutions isomorphes en exploitant leurs automorphismes. La section 6 fournit des résultats expérimentaux montrant les gains obtenus grâce à notre approche.

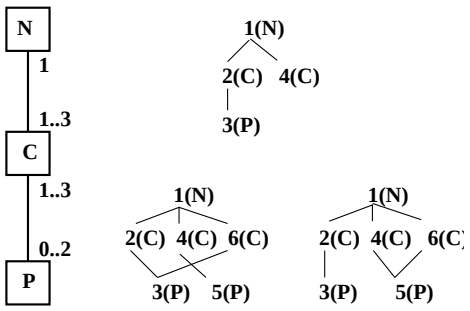

FIG. 1 – Un problème de connection d'ordinateurs et d'imprimantes en réseaux. A gauche, la modélisation de l'ensemble des types de composants et de leurs relations. A droite, 3 exemples de structures solutions. Les deux structures du bas sont isomorphes et représentent deux solutions équivalentes.

2 Problèmes de configuration, sous-problèmes structurels et isomorphismes

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. Les instances valides d'un modèle de configuration (appelées *configurations*) indiquent les objets les constituant ainsi que leurs relations, notamment de *type* (relation unaire de taxonomie) ou de *composition* (relation binaire spécifiant qu'un objet n'est le composant que d'au plus un composite).

Voici un exemple très simple de problème de configuration qui nous servira à illustrer toutes les notions dont nous traiterons dans l'article. Il s'agit de configurer un réseau (N) d'ordinateurs (C) et d'imprimantes (P) (voir Figure 1). Le réseau doit contenir entre un et trois ordinateurs dont chacun peut être relié à un maximum de deux imprimantes. Chaque imprimante doit elle-même être en liaison avec au moins un ordinateur et trois au maximum. Par ailleurs, on a les contraintes globales suivantes : il n'y a qu'un réseau et il n'y a pas plus de deux imprimantes disponibles. Dans un problème réel, les ordinateurs et les imprimantes pourraient posséder un certain nombre d'attributs à déterminer en fonction d'un certain nombre de contraintes mais nous laissons ces éléments de côté car ils ne sont pas utiles pour illustrer notre approche.

Les solutions de problèmes de configuration contiennent des objets interconnectés, comme l'illustre la figure 1, qui par ailleurs rend explicite l'existence d'isomorphismes structurels.

Nous isolons ce qu'on appelle le *sous-problème structurel* d'un problème de configuration, 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. Ce sont les isomorphismes qui apparaissent lors de la résolution de ce sous-problème que nous traitons. Afin de simplifier le propos, nous faisons abstraction de tout formalisme habituellement utilisé pour la description des problèmes de configuration et nous considérons un ensemble totalement ordonné d'objets O (on utilisera en pratique $O = \{1, 2, \ldots\}$), un ensemble totalement ordonné de (symboles de) types T_C et un ensemble totalement ordonné R_C de (symboles de) relations binaires sur O. Pour n'importe quelle relation binaire injective R, nous écrirons indifféremment $(x, y) \in R$ ou $y = R(x)$.

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).*

Dans le problème du réseau de la figure 1, t=`N`, T_C ={`N`, `C`, `P`}, R_C = {(`N`,`C`), (`C`,`P`)} et C contient toutes les contraintes structurelles indiquant le minimum et le maximum d'objets pouvant être reliés à chaque objet pour chaque relation binaire.

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

Nous emploierons indifféremment les termes *structure* ou *configuration* pour désigner une solution du problème structurel.

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, 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). Ex : la solution du haut de la

figure 1 peut se représenter par (N, {1,2,3,4}, {(1,2), (2,3), (1,4)}, {(1,N), (2, C), (3,P), (4, C)}).

Définition 3 (configurations isomorphes) *Deux configurations G=(t ,X ,E ,L) et G'=(t',X',E',L') sont isomorphes ssi t=t', L=L' et il existe une bijection σ entre X et X' telle que $\forall$ (x,y) $\in$ E, (σ(x), σ(y)) $\in$ E et $\forall$ (x,l) $\in$ L, (σ(x),l) $\in$ L.*

Ex : les 2 solutions du bas de la Figure 1 sont isomorphes car σ =((1,1), (2,4), (3,5), (4,2), (5,3), (6,6)) est une bijection qui correspond aux critères de la définition.

Tester l'isomorphisme de deux graphes est un problème de NP qui n'a toujours pas été ni prouvé NP-complet ni prouvé polynomial. Son importance a conduit à définir la classe des problèmes *graph isomorphism complete*, qui représente tous les problèmes qui lui sont équivalents en terme de complexité. Parmi ces problèmes équivalents, nous retrouvons notre problème d'isomorphisme de DAG colorés. Pour certains types de graphes, les arbres ou les graphes de degré borné par une constante, le test d'isomorphisme est polynomial [7]. Par ailleurs, il existe des algorithmes efficaces en pratique, le plus connu étant Nauty [11]. Ceci étant dit, nous devons insister sur le fait que Nauty ne peut être utilisé dans notre situation. En effet, nous devons maintenir la propriété que toute structure canonique doit pouvoir être obtenue à partir d'une structure plus petite et elle-même canonique. Cette propriété n'appartient pas à la définition de la canonicité utilisée par Nauty. Si Nauty détecte qu'une structure n'est pas canonique, on ne peut pas la rejeter car en la complétant on pourrait obtenir une structure canonique solution. Nous donnons plus loin de plus amples explications à ce sujet.

Une classe d'isomorphisme représente l'ensemble des graphes isomorphes entre eux. Dans une classe d'isomorphisme, tous les graphes sont équivalents et une procédure de génération de graphes ayant certaines propriétés (dans notre contexte, le respect du modèle structurel) ne doit idéalement générer qu'un seul représentant, appelé *graphe canonique*, par classe d'isomorphisme. C'est d'autant plus crucial que le cardinal d'une classe d'isomorphisme contenant des graphes de n sommets est $n!$ (le nombre de permutations de l'ensemble des sommets qui produisent un graphe différent) si ses graphes ne contiennent pas d'automorphismes (de permutations qui laissent le graphe inchangé[2]), ce qui est le cas de la majorité des graphes. Seuls les graphes complets et les graphes sans arêtes sont chacun l'unique élément de leur classe d'isomorphisme.

Les procédures de génération de graphes fonctionnent généralement par *extension unitaire*, c'est-à-dire par ajout d'une arête à un graphe donné, de toutes les manières possibles (et valides en fonction du contexte), afin de produire un ensemble de graphes, qui seront étendus à leur tour. Il y a deux types d'arête ajoutée : l'arête interne, qui lie deux sommets qui étaient déjà présents dans le graphe, et l'arête externe, qui lie un sommet déjà présent et un sommet que l'on crée pour l'occasion. Avoir une procédure efficace de test de canonicité (problème graph-iso complet) ne suffit pas à obtenir une procédure efficace de génération de graphes canoniques. En effet, vérifier la canonicité des graphes solutions après qu'ils aient été générés ne fait pas gagner de temps. Idéalement, il faut pouvoir rejeter immédiatement les graphes dont tous les supergraphes solutions sont non canoniques. Le moyen le plus classique est de définir le critère de canonicité de telle manière que *tout graphe canonique possède au moins un sous-graphe résultant du retrait d'une seule de ses arêtes qui est lui-même canonique*. Nous disons alors que ce critère de canonicité a la *propriété de rétractabilité canonique*. C'est une condition nécessaire (mais non suffisante, cf plus bas) pour pouvoir backtracker dès qu'un graphe canonique est détecté pendant l'exécution d'une procédure de recherche. En effet, s'il existe un seul graphe canonique qui ne possède pas un tel sous-graphe canonique, alors il faut nécessairement étendre un graphe non canonique pour le générer. Or, un tel critère de canonicité est difficile à trouver et ne correspond pas en général à ceux utilisés pour les tests de canonicité classiques, y compris celui de Nauty. Il existe des procédures générales de génération de graphes sans isomorphe qui imposent des conditions sur la canonicité, les plus connus étant les *orderly algorithm* [13], mais ils n'explicitent pas un test de canonicité qui soit efficace. A notre connaissance, un tel test efficace n'a pas encore été trouvé dans un contexte général (si tant est qu'il existe). Par contre, il existe des procédures spécialisées pour générer efficacement les graphes qui ont certaines propriétés : les arbres, les graphes cubiques [3] et, plus généralement, les graphes ayant des propriétés héréditaires[3] [10]. Mais les problèmes de configuration ne rentrent pas dans ce cadre et il nous faut donc trouver nos propres procédures. Pour ceci, nous allons nous reposer sur des travaux qui ont déjà effectué sur le sujet en configuration.

[2] Ce qui signifie (peut-être contre-intuitivement) qu'un graphe possède d'autant moins d'isomorphes qu'il possède beaucoup de symétries internes.

[3] Une propriété d'un graphe est héréditaire si tous ses sous-graphes possèdent aussi cette propriété.

Travaux connexes

Plusieurs approches ont été expérimentées pour traiter les isomorphismes structurels. Mais, bien souvent, on ne raisonne que sur le dernier niveau de composition d'objets, l'idée étant d'éviter l'ajout d'objets interchangeables pendant la recherche ou de substituer l'ajout effectif d'un type d'objet par l'incrémentation d'un compteur de ce type d'objet [8]. Dans [4] est donné un test de canonicité de complexité pseudo-linéaire qui a la propriété de rétractabilité canonique lorsque les problèmes de configuration ne possèdent que des relations de composition, ce qui implique que les structures des solutions soient des arbres. Dans [6], le résultat est généralisé sous une forme affaiblie à tout problème de configuration, sans restriction sur les relations entre types de composants. Mais toute procédure de génération de configurations n'est pas compatible avec un critère de canonicité donné même s'il a la propriété de rétractabilité canonique.

Graphe d'états d'un problème de configuration

Considérons le *graphe d'états* $G_P = (X_P, E_P)$ d'un problème de configuration. Un état de l'ensemble X_P est simplement une structure (un DAG coloré) respectant la modélisation struturelle du problème. E_P contient tous les couples (g, h) tels que g, h $\in X_P$ et h est le résultat d'une extension unitaire de g. (G_P est lui-même un DAG dont la racine est l'état (t, {1}, $\emptyset$, {(1,t)})). Une procédure de génération de structures doit être complète et non redondante, c'est-à-dire générer une seule fois chacune des structures de X_P en parcourant G_P. Ce parcours peut être représenté par un arbre couvrant T_P de G_P. Considérons maintenant le graphe d'états G'_P qui est le sous-graphe de G_P ne contenant que les structures canoniques. La propriété de rétractabilité canonique garantit que G'_P est connexe et donc qu'il existe une procédure de génération de structures canoniques pour laquelle on peut backtracker dès qu'on rencontre une structure non canonique. Mais celà ne signifie pas que toute procédure de génération de structure peut convenir. En effet, si l'intersection T'_P entre T_P et G'_P n'est pas un graphe connexe, backtracker sur les structures non canoniques fera que la procédure ne sera pas complète. Pour que la procédure soit complète, il faut que T'_P soit un arbre couvrant de G'_P. Ce sont de telles procédures que nous allons présenter maintenant.

3 Generation de structures d'arbre sans isomorphes

Nous présentons ici une procédure de génération de structures canoniques de problèmes de configuration lorsque le modèle structurel ne contient que des relations de composition. Une relation de composition entre un type T1, dit *composite*, et un autre type T2, dit *composant*, est une relation binaire entre T1 et T2 spécifiant que tout objet de type T2 ne peut être relié qu'à un objet de type T1 au maximum. Ex : la relation entre le type `N` et le type `C` de la Figure 1 est une relation de composition. Ce n'est pas le cas de la relation entre le type `C` et le type `P`. Pour que ça le soit, il faudrait qu'on impose qu'une imprimante ne puisse être reliée à plus d'un ordinateur. Dans le cadre de cette section, nous considérerons que l'exemple de la Figure 1 comporte cette restriction et qu'il n'a donc que des relations de composition. Les structures de tels problèmes sont alors nécessairement arborescentes.

```
procedure generate(T, F)
   if canonical(T) then
      output T
      // génère l'ensemble E = {(x1,y1), ..., (x|E|,y|E|)} des
      // arêtes permettant des extensions unitaires valides
      E = extensions(T, F)
      for  i := 1 to |E| do
         generate(T ∪ {(xi,yi)},
                  F ∪ {(x1,L(y1)),... , (xi−1,L(yi−1))})
```

FIG. 2 – La procédure **generate**. Pour générer tous les arbres, on lance la procédure par l'appel `generate((t, {1}, ∅, {(1,t)}),{})`

La procédure indiquée en Figure 2 est complète, non redondante et génère exclusivement des structures canoniques. La fonction `extensions(T, F)` retourne la suite E des extensions unitaires pertinentes de T, c'est-à-dire toutes celles qui sont compatibles avec le graphe de relations entre les types (qui modélise le problème structurel) et qui ne sont pas interdites par F. L'ensemble E contient des arcs e_i liant deux sommets de l'ensemble O des objets. Un des sommets était déjà dans l'arbre T et l'autre n'y était pas. Le paramètre F contient l'ensemble des extensions unitaires qu'il ne faut pas faire et permet ainsi d'éviter de générer plusieurs fois un même arbre. Une telle redondance surviendrait si à partir de T, on produisait un arbre T1 par ajout de l'arc e_1 et un arbre T2 par l'ajout de l'arc e_2, puis qu'on ajoutait e_2 à T1 et e_1 à T2, ce qui produirait deux fois le même arbre. Pour éviter ceci, il faut partitionner la recherche en deux : une recherche d'extensions de T $\cup \{e_1\}$ et une recherche d'extensions de T $\cup \{e_2\}$ avec interdiction d'utiliser e_1. Plus précisément, ce qu'on interdit, c'est d'ajouter un arc (x, z) si e_1=(x, y) et L(y)=L(z). Parce que même si ces deux arbres sont différents, ils sont isomorphes : il suffit d'échanger y et z pour passer d'un arbre à l'autre. Ce sont donc les couples de la forme (x, L(y)) qui font partie de F. Ils signifient qu'on ne doit pas ajouter un arc constitué de l'objet x relié à un nouvel objet

de type L(y). Dans le cas général, on a $|E|$ extensions possibles et donc on partitionne en $|E|$ parties par la suite d'appels

$$generate(T \cup \{(x_i, y_i)\},\ F \cup \{(x_1, L(y_1)), \ldots (x_{i-1}, L(y_{i-1}))\})$$

L'ordre des arcs de E pourrait être arbitraire si on n'essayait pas d'éliminer les arbres non canoniques. Mais, comme nous l'avons vu en fin de la section précédente, il doit être choisi en fonction du critère de canonicité si on veut que la procédure soit complète. Nous allons donner un ordre adéquat en fonction du critère de canonicité donné dans [4]. Dans cet article, les auteurs définissent un ordre total $\preccurlyeq$ sur les arbres afin de définir la canonicité d'un arbre comme étant le représentant $\preccurlyeq$-minimal de sa classe d'isomorphisme. Il y est démontré que ce critère de canonicité a la propriété de rétractabilité canonique. Pour que la procédure soit complète, il faut que les arêtes de E soit ordonnées de la manière suivante : lorsqu'il faudra choisir quelle arête ajouter en premier, l'ajout d'une arête e_i sera essayé avant l'ajout d'une arête e_j si $T \cup \{e_i\} \preccurlyeq T \cup \{e_j\}$.

Proposition 1 *La procédure* `generate` *est complète.*

Preuve 1 *(sketch)*

On montre d'abord par induction que les arêtes sont ajoutées en les reliant à chacun des sommets de la branche droite d'un arbre, en commençant par le sommet le plus profond et en remontant jusqu'à la racine. C'est trivialement vrai pour l'arbre vide. Si c'est vrai pour tout arbre T à n sommets, ça signifie que le sommet y qui a été relié à un sommet x de la branche droite d'un arbre pour former T est maintenant l'extrêmité de la branche droite de T. Or, avec l'ajout de y, on a aussi interdit de faire des extensions unitaires à partir des sommets situés sous x (en complétant l'ensemble F). Les seules possibilités restantes sont les sommets ancêtres de x, ainsi que x et y, c'est-à-dire tous les sommets de la branche droite de T. Ensuite, dans [5], la rétractabilité canonique est montrée grâce au fait que le retrait de l'extrêmité de la branche droite d'un arbre préserve sa canonicité. En conséquence, comme, à partir de tout arbre T, la procédure `generate` *produit toutes les extensions de T telles que le retrait de l'extrêmité de leur branche droite ferait retomber sur T, elle produit tous les arbres canoniques possibles que l'on peut obtenir par extension unitaire à partir de T. La procédure est donc complète.*

Dans notre exemple de réseau de machines et d'imprimantes, si nous n'autorisions les imprimantes qu'à être reliées à une machine au plus, la relation imprimante-machine serait une relation de composition. Alors, les solutions structurelles de notre problème auraient toutes une forme d'arbre. La Figure 3 montre l'arbre de recherche qui en résulterait. La procédure backtrackerait sur les arbres non canoniques 6, 13 et 17. Les arbres non canoniques 10, 12, 16, 18 et 19 ne seraient donc pas générés.

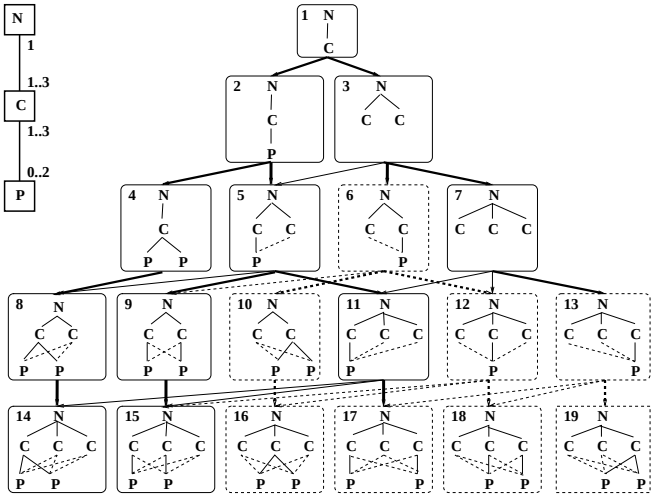

FIG. 3 – Une partie du graphe d'état du problème de configuration de réseau machine-imprimantes. Les arbres sont représentés avec uniquement le type de leurs sommets. Les lignes pointillés joignant des sommets sont les arêtes internes qui peuvent être ajoutés pour faire la complétion. Les arbres encadrés par des pointillés ne sont pas canoniques. Tous les arcs, pointillés ou non, à traits fins ou larges, sont des extensions unitaires a priori possibles entre les arbres. Un arc entre deux arbres non canoniques est en pointillé. Un arc à trait large (pointillé ou pas) est un arc emprunté par une procédure de génération de graphe. Seuls les arcs pleins et à traits larges sont des transitions effectuées par la procédure qui rejette les arbres non canoniques.

4 Génération de DAG sans certains isomomorphes

Nous présentons maintenant une procédure generant uniquement ce que nous appelerons des *DAG faiblement canoniques*, c'est-à-dire des DAG dont l'arbre couvrant minimal selon $\preccurlyeq$ est canonique. Nous évitons ainsi de générer tous les DAG non faiblement canoniques car ils sont isomorphes à leur DAG faiblement canonique, pour la simple raison que la permutation qui rendrait leur arbre couvrant canonique est la même que celle qui les rendrait faiblement canonique. Rappelons que la génération efficace d'un seul représentant (canonique) par classe d'isomorphisme de DAG est un

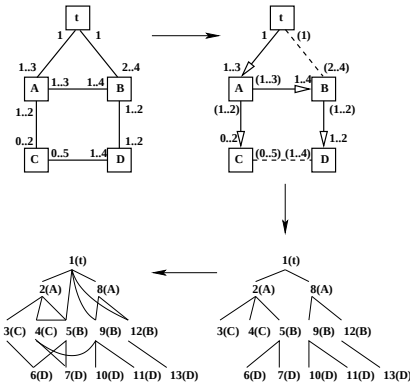

FIG. 4 – Génération de DAG à partir d'arbres. En haut à gauche, un modèle de structure. En haut à droite, un arbre couvrant de composition du modèle. En bas à droite, une structure possible du modèle relaxé. En bas à gauche, une solution possible du modèle après complétion de l'arbre.

problème ouvert. C'est pourquoi nous nous restreignerons ici à la génération de DAG faiblement canoniques.

Le principe est de générer d'abord un arbre canonique, appelé *arbre structurel*, puis d'effectuer des extensions unitaires qui n'ajoutent que des arêtes internes. Nous pouvons générer tous les arbres canoniques en utilisant notre procedure de génération d'arbres canoniques qui backtracke dès qu'un arbre non canonique est produit. A partir de chacun de ces arbres, nous générons tous les DAG dont il est l'arbre structurel.

La Figure 4 montre comment mettre cette idée en oeuvre. On part d'un modèle structurel contenant des relations binaires quelconques et on en extrait un sous-modèle ne contenant que des relations de composition. Ce sous-modèle est un arbre couvrant (arbitraire) du modèle d'origine. A partir de ce sous-modèle, on peut générer des solutions arborescentes, qui seront complétées pour former des DAGs solutions du problème d'origine. Il faut faire attention à ne pas mettre en oeuvre l'idée de façon naïve sous peine de générer plusieurs fois les mêmes DAGs.

Premièrement, les arêtes qui sont des extensions possibles d'un arbre sont ordonnées selon un ordre (arbitraire) $<$. Selon le même principe que pour l'ajout d'arête que nous avons vu précédemment pour la construction d'arbre, les arêtes sont toujours ajoutées selon l'ordre $<$ et une arête ne peut plus être ajoutée s'il existe une arête e' qui fait déjà partie du DAG et qui est telle que e $<$ e'. Comme pour la constitution d'arbre, il est évident que cela évite de générer des DAG déjà générés auparavant. Soit **a** le nombre d'arêtes internes qu'on peut ajouter à un arbre T, le nombre de DAG qu'on peut générer à partir de T sera $2^{|\mathbf{a}|}$ au lieu de $|\mathbf{a}|^{|\mathbf{a}|}$. Mais cela ne suffit pas à éliminer toutes les redondances dues à la génération de DAG identiques. Pour y parvenir, nous allons effectuer un test supplémentaire sur chaque DAG généré : nous allons vérifier s'il contient un plus petit (selon $\preccurlyeq$) arbre recouvrant que son arbre structurel. Si ce n'est pas le cas, cela signifie qu'on peut backtracker sur ce DAG. En effet, il existe alors un arbre canonique isomorphe à cet arbre couvrant qui est plus petit (selon $\preccurlyeq$) que l'arbre structurel du DAG. Cet arbre canonique est lui-même l'arbre structurel d'un autre DAG qui a donc déja produit (ou qui produira) par complétion un DAG isomorphe au DAG courant. La procédure qui vérifie l'identité entre arbre couvrant minimal et arbre structurel est de complexité O(n), n étant le nombre de sommets du DAG. Elle consiste simplement en la comparaison selon $\preccurlyeq$ entre les deux arbres. Un des principaux avantage de cette technique est qu?il est suffisant de trouver n?importe quel arbre recouvrant qui soit inférieur à l?arbre structurel, même si il n?est pas canonique.

Considérons l'arête (x,y) nouvellement ajoutée, x étant le prédécesseur de y dans le DAG. Le DAG a déjà un autre prédécesseur p qui est aussi le père de y dans l'arbre structurel T du DAG. Considérons maintenant l'arbre T' résultant du remplacement de l'arête (p,y) par l'arête (x,y). Si T' $\preccurlyeq$ T alors nous rejetons le DAG résultant de l'extension (x,y).

Par exemple, dans l'arbre 15 de la Figure 3, l'arête interne reliant le premier `C` au deuxième `P` ne doit pas être ajoutée le plus petit arbre recouvrant du DAG serait alors l'arbre 14.

```
procedure completion(G, F)
    output G
    // génère l'ensemble E=(e_1, ...e_|E|) des arêtes internes
    // permettant des extensions unitaires pas dans F
    E = extensions-internes(G, F)
    for i := 1 to |E| do
        completion(G ∪ {e_i}, F ∪ {e_1, ...e_{i-1}})
```

FIG. 5 – La procédure **completion**.

Proposition 2 *Notre procédure* `generate` *avec un appel à* `completion` *à chaque arbre canonique génère une seule fois chaque DAG faiblement canonique.*

Preuve 2 *(sketch) La procédure* `completion` *ne génère jamais deux fois le même DAG à partir d'un arbre canonique donné et jamais non plus un DAG qui résulterait de la complétion d'un autre arbre.*

5 Exploitation d'automorphismes

La procédure `completion(G,F)` peut être améliorée en faisant en sorte d'éliminer les DAG isomorphes résultant des extensions unitaires d'un DAG. L'idée générale est simple : si e_1 et e_2 sont deux arcs internes qui peuvent compléter G pour former G1 et G2 et que G1 et G2 sont isomorphes alors on ne fait pas d'extension unitaire ajoutant e_2. Par exemple, si on considère le DAG en bas à droite de la Figure 1, lui rajouter l'arc (4,3) produit un DAG isomorphe à celui qui résulterait de l'ajout de l'arc (6,3). On évitera donc de faire une de ces deux extensions unitaires.

Une première idée (coûteuse) est donc de considérer tout couple de graphes complétés par un arc de E et de tester s'ils sont isomorphes (grace à Nauty, par exemple). Si c'est le cas, on supprime de E un des arcs de complétion. L'inconvénient de cette méthode est qu'il y a potentiellement $O(n^2)$ extensions unitaires pour un graphe à n sommets, donc $0(n^2)$ graphes dont on peut trouver le représentant canonique grâce à une procédure telle que celle de Nauty, ce qui fait de l'ordre de $O(n^4)$ couples de graphes canoniques à comparer. Par ailleurs, même si Nauty a un comportement polynomial sur la plupart des graphes, il est de complexité exponentielle dans le pire des cas. Nous allons donc utiliser une méthode de détection de graphes isomorphes qui sera incomplète mais garantira une complexité très faible.

Cette méthode utilise les symétries éventuelles (précisément, le groupe d'automorphisme) de l'arbre structurel : la canonicité de ces arbres a été définie récursivement dans [4] de telle manière que, pour tout sommet de l'arbre, les sous-arbres dont il est la racine sont ordonnés lexicographiquement selon $\preccurlyeq$. En conséquence, deux sous-arbres sont permutables en laissant l'arbre inchangé seulement si leurs racines sont fils d'un même sommet et sont voisines. Il suffit alors que ces deux sous-arbres (canoniques) soient identiques pour qu'ils soient permutables, ce qui se teste en temps linéaire en fonction du nombre n de sommets de l'arbre. En conséquence, déterminer quels sont tous les couples de sommets interchangeables dans un arbre est une opération en $O(n^3)$ qui peut se faire une seule fois avant que ne commence la complétion d'un arbre structurel. Grâce à la connaissance des sommets interchangeables, nous pouvons éviter l'ajout inutile d'un certains nombre d'arêtes. Il est évident qu'un DAG résultant d'une extension unitaire (x,z) est isomorphe au DAG résultant du remplacement de l'arc (x,z) par l'arc (y,z) si x et y sont interchangeables et si z n'est pas un sommet à l'intérieur des sous-arbres de racine x ou y. Une des deux extensions unitaires n'a donc pas lieu d'être effectuée.

Afin de tenir compte du fait que l'interchangeabilité entre deux sommets est perdue dès lors qu'un sommet est utilisé dans une extension unitaire complémentante, on introduit un système de marquage supplémentaire. Dès qu'un arc complémentant est ajouté, il se forme un nouveau cycle (mais pas un circuit) dans le T-dag. Tous les sommets de ce cycle sortent de leur classe d'interchangeablité et seront donc marqués comme faisant partie d'un cycle et donc disponibles à l'avenir pour faire partie d'un nouvel arc complémentant. Pour obtenir l'ensemble des sommets de ce cycle, il suffit de remonter de père en père tous les sommets du T-arbre à partir de chacune des deux extrêmités de l'arc ajouté jusqu'à leur ancêtre commun (cf figure 6).

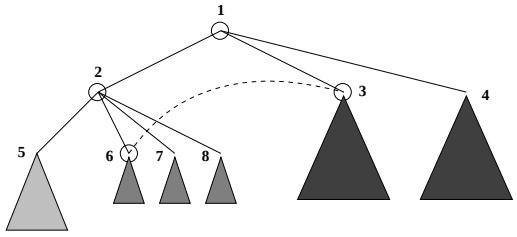

FIG. 6 – Ajout d'une arête interne et marquage d'un cycle.

Dans le l'arbre canonique de la figure 6, les sous-arbres de racine 6, 7 et 8 sont identiques, ainsi que les sous-arbres de racine 3 et 4. Si le choix d'interconnecter des sommets de ces deux groupes est fait, alors on le fait par exemple uniquement avec les sommets 3 et 6 car les sommets 4, 7 et 8 sont interchangeables avec les sommets 3 ou 6. Dès que l'arc (3,6) est ajouté, les sommets 1, 2, 3 et 6 sont marqués comme faisant partie d'un cycle (et ils ont perdu leur interchangeabilité) et il redevient possible d'utiliser ultérieurement les sommets 4 ou 7 comme extrêmité d'arête interne à ajouter.

6 Résultats expérimentaux

Nos expérimentations ont été menées sur l'exemple du problème de configuration du réseau machines-imprimantes illustré figure 1 en faisant varier la taille du problème, sur un PC à 1.7 Ghz avec 512Mo de RAM sous Linux. Pour chaque choix de nombres c de machines et p d'imprimantes, nous avons généré toutes les structures (DAG) possibles de solutions grâce à deux variantes d'algorithme : *Covering Tree* ou *ct* (génération des arbres canoniques, chacun complété en DAG grâce à l'ensemble ordonné de toutes ses extensions internes possibles avec backtrack sur les DAG dont l'arbre structurel devient inférieur selon $\preccurlyeq$ à cause de l'ajout de l'arête) et *full* (*ct* + élimination d'arêtes grâce à la détection d'interchangeabilité de sommets).

c	p			*ct*		*full*	
1	3	8	4	4	0	4	0
2	3	64	16	32	0	30	0
3	3	512	46	273	0	262	0
4	3	4096	109	2234	0.01	2078	0.02
5	3	32768	219	17099	0.12	13095	0.1
6	3	262144	393	130404	1.01	69757	0.64
7	3	$2.1\ 10^6$	649	993197	8.34	329495	3.43
8	3	$1.6\ 10^7$	1006	/	/	$1.45\ 10^6$	17.23

TAB. 1 – Resultats le problème de réseau de (c) PC - (p) imprimantes. (temps en secondes, "/" = temps > 60 secondes)

Nous avons comparé le nombre de DAG générés pour les deux algorithmes avec le nombre de DAG solutions du problème structurel si aucun isomorphe n'était éliminé. Ce nombre de solutions serait égal au nombre de graphes bipartis (canoniques ou pas) reliant un ensemble de c sommets à un ensemble de p sommets : $2^{c.p}$.

La table 1 nous montre que le nombre de DAG est significativement diminué grâce à l'algorithme *ct*, grâce au nombre important de DAG isomorphes dont on évite la génération. L'algorithme *full* permet une diminution encore plus importante, en nombre de DAG comme en temps d'exécution.

Les configurateurs actuels ne parviennent à résoudre que des problèmes de taille limitée. L'utilisation de ces stratégies d'élimination d'isomorphes pourra donc permettre de traiter des problèmes de plus grande taille. Il faut noter que le problème sur lequel nous avons fait nos tests ne doit pas être considéré comme artificiel mais plutôt comme générique : il met en jeu un type de relation couramment utilisé dans la modélisation de problème de configuration : une relation binaire générale soumise à une limitation du nombre d'objets connectables à chaque type d'objet. Même avec un problème qui ne contient qu'une seule de ces relations, les structures des solutions sont des DAG bipartis dont il n'existe pas de procèdure d'élimination efficace (avec test polynomial de rejet de DAG non canoniques) de tous les isomorphes. Le test de canonicité des DAG bipartis est d'ailleurs graph-iso complet. Aussi simple et petit à première vue que semble notre exemple, les configurateurs actuels généreraient tous les DAG ($2^{p.c}$) bipartis avec tous les isomorphes. Notons aussi que si un problème mettait en jeu plusieurs relations binaires générales entre des types (par exemple une relation R1 entre des types A et B et une relation R2 entre le type B et un type C) alors le gain que nous avons obtenu sur une seule relation s'obtiendrait sur chacune des relations et le gain global résulterait des gains obtenus sur chaque relation (R1 et R2).

Insertion dans une recherche globale de configuration

Un problème de configuration est normalement défini aussi par des attributs associés aux objets et devant respecter certaines contraintes. La génération de la structure de la solution n'est donc qu'une partie de la résolution du problème, le reste consistant à affecter une valeur à chaque attribut en respectant des contraintes, c'est-à-dire à résoudre un CSP. Toutes les techniques de résolution de CSP sont alors applicables et on peut mélanger les phases de complétion de structure et d'affectation d'attribut. Par ailleurs, remarquons que notre technique d'élimination d'arêtes repose sur la connaissance d'interchangeabilité de sommets de la structure, ce qui implique aussi l'interchangeabilité des attributs des objets que représentent ces sommets. Ainsi, les techniques existantes de traitement de symétries dans les CSP peuvent être utilisées après la constitution de la structure pour affecter les attributs et bénificieront de cette pré-recherche d'éléments symétriques.

7 Conclusion

Les travaux que nous avons présentés étendent significativement le traitement des nombreux isomorphismes présents dans pratiquement tous les problèmes de configuration, alors qu'il était limité jusqu'à présent à la détection d'interchangeabilité d'objets non encore utilisés ou au comptage d'objets non composites. La procédure de génération de DAG colorés que nous avons présentée s'attaque au problème des isomorphismes de structure des solutions des problèmes de configuration dans toute sa généralité et permet des gains de temps de résolution très appréciables même pour des problèmes de taille réduite.

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] 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.

[3] G. Brinkmann. Fast generation of cubic graphs. *J. Graph Theory*, 23 :139–149, 1996.

[4] Stephane Grandcolas, Laurent Henocque, and Nicolas Prcovic. A canonicity test for configuration. In *Proceedings of CP'2003*, September 2003.

[5] Stephane Grandcolas, Laurent Henocque, and Nicolas Prcovic. Pruning isomorphic structural sub-problems in configuration. Technical report, LSIS, June 2003. Available from the CoRR archive at http ://arXiv.org/abs/cs/0306135.

[6] 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.

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

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

[9] John P. McDermott. R1 : A rule-based configurer of computer systems. *Artificial Intelligence*, 19 :39–88, 1982.

[10] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2) :306–324, 1998.

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

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

[13] Ronald C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics*, 2 :107–120, 1978.

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

[15] Timo Soininen, Esther Gelle, and Ilkka Niemela. A fixpoint definition of dynamic constraint satisfaction. In *Proc. of CP'99*, pages 419–433, 1999.

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

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