

# Advances in Polytime Isomorph Elimination for Configuration

Laurent Hénoque, Mathias Kleiner and Nicolas Prcovic

LSIS - Universités d'Aix-Marseille II et III  
Faculté de St-Jérôme - Avenue Escadrille Normandie-Niemen - 13013 Marseille  
laurent.henocque,mathias.kleiner,nicolas.prcovic@lsis.org

**Abstract.** An inherent and often very underestimated difficulty in solving configuration problems is the existence of many structural isomorphisms. This issue of considerable importance attracted little research interest despite its applicability to almost all configuration problems. 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 generic configuration example.

## 1 Introduction

Configuring consists in simulating the constrained realization of a complex product from a catalog of component parts, using known relations between types, and instantiating object attributes. The industrial need for configuration applications is ancient [1], and has triggered the development of many configuration programs and formalisms [2–8].

The main objective of this research is to reduce the undue combinatorial effort incurred by isomorphisms during configuration search. We focus on the dynamic nature of configuration, which consists in generating the structure of possible solutions.

This difficulty is one among the most important to tackle in configuration, if one expects to solve problems having highly variable solutions. Solvers cannot currently handle the search space of these problems because of the exponential number of isomorphs that they generate for each canonical solution structure<sup>1</sup>. We propose a general search procedure eliminating a great number of such isomorphisms in configuration problems. More precisely, we first present a complete, non redundant procedure restricted to the generation of typed tree structures which also efficiently eliminates any non-canonical structure. Starting from this procedure, we generalize it to another able to generate all type of structures (DAGs), in a complete and non redundant way, while still avoiding the generation of an important number of isomorphic structures. This work is based upon the results obtained in [9] and [10], which demonstrated necessary existence conditions for such procedures but did not explicit any of them, leaving the crucial point of how defining them unmentioned.

---

<sup>1</sup> Industrial problems currently solved with configurators only involve models of limited size.

## Plan of the article

Section 2 describes the formalism used throughout the paper, the notion of *structural sub-problems* and the problem of generating them. We also briefly recall why representing the structure of a configuration problem as a graph is more efficient than defining it as a CSP. Section 3 presents a complete procedure for generating canonical structural problems when they have a tree structure. Section 4 presents a way to extend the procedure so that it applies to any configuration problem. Section 5 describes how to exploit the symmetries of covering tree structures so as to reject more isomorphisms. Section 6 provides experimental results on the benefits of using such an approach. Section 7 concludes.

## 2 Configuration problems, structural sub-problems and isomorphism

A configuration problem describes a generic product, in the form of declarative statements (rules or axioms) about product well-formedness. Valid configuration model instances (called *configurations*) involve objects and their relationships, notably *types* (unary relations involved in taxonomies) and binary relations. Some binary relations are *composition* relations having stronger functional semantics (an object is a component of at most one composite).

Here is a very simple example of a configuration problem which will allow us to illustrate notions throughout this paper. The problem is to configure a network of computers (C) and printers (P) (as illustrated in Figure 1). The network involves up to three computers, each of which being connected to at most two printers<sup>2</sup>. Conversely, each printer must be connected to at least one and at most three computers. Besides this, we have two global constraints: there is only one network, and there are only two printers available. In a real problem, computers and printers could have specific attributes that would be instantiated while obeying other constraints. This can be left aside as we solely focus on structural constraints.

Solutions to configuration problems involve interconnected objects, as illustrated in Figure 1, which makes explicit the existence of structural isomorphisms.

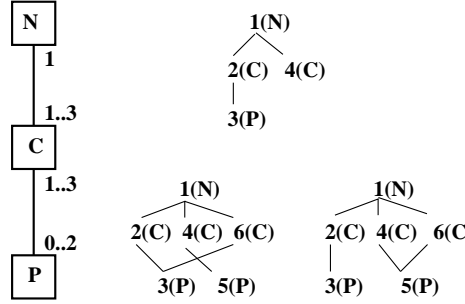
We isolate configuration sub-problems called *structural problems*, that are built from the binary relations, the related types and the structural constraints alone, and study their isomorphisms. For simplicity, we abstract from any configuration formalism, and consider a totally ordered set  $O$  of objects (we normally use  $O = \{1, 2, \dots\}$ ), a totally ordered set  $T_C$  of type symbols (unary relations) and a totally ordered set  $R_C$  of binary relation symbols.

For any binary injective relation  $R$ , we will use either  $(x, y) \in R$  or  $y = R(x)$ .

**Definition 1 (syntax).** A structural problem, is a tuple  $(t, T_C, R_C, C)$ , where  $t \in T_C$  is the root configuration type, and  $C$  is a set of structural constraints applied to the elements of  $T_C$  and  $R_C$ .

---

<sup>2</sup> This is the Figure data, our experiments involve more computers and printers.



**Fig. 1.** A network connection problem. On the left, the model for the components types (network, computers and printers) and their relations. On the right, 3 examples of possible structures. The two structures at the bottom are isomorphic and therefore represent equivalent solutions.

In the network problem of Figure 1, we have  $t=N$ ,  $T_C = \{N, C, P\}$ ,  $R_C = \{(N, C), (C, P)\}$  and  $C$  is the set of structural constraints which enforce the minimum and maximum number of objects that can be connected for each binary relation.

**Definition 2 (semantics).** An instance of a structural problem  $(t, T_C, R_C, C)$  is an interpretation  $I$  of  $t$  and of the elements of  $T_C$  and  $R_C$ , over the set  $O$  of objects. If an interpretation satisfies the constraints in  $C$ , it is a solution of the structural problem.

We use the term *structure* or *configuration* to denote a structural problem solution.

A configuration can be represented using a vertex-colored DAG (directed acyclic graph)  $G=(t,X,E,L)$  with  $X \subset O$ ,  $E \subset O \times O$  and  $L \subset O \times T_C$ . The symbol  $t$  is the root type,  $X$  the vertex set,  $E$  the edge set and  $L$  is the function which associates each vertex to a type.

As an example, the upper solution of Figure 1 can be represented by the quadruple  $(N, \{1,2,3,4\}, \{(1,2), (2,3), (1,4)\}, \{(1,N), (2, C), (3,P), (4, C)\})$ .

For those who are not familiar with configuration problems, we recall now why it is more efficient to solve the structural problem by defining it as the construction of a graph rather than using the CSP formalism. Indeed, with CSPs, we would have to consider the maximal number  $C$  of PC's and  $P$  of printers to define the CSP variables. One possible model is to assign a variable to each PC and each printer ( $C+P$  variables). The domain of a variable assigned to a PC (resp. a printer) would contain all parts of the set  $\{1, \dots, P\}$  (resp.  $\{1, \dots, C\}$ ) and therefore would be of size  $2^P$  (resp.  $2^C$ ). The search space would then be of size  $(2^P)^C \times (2^C)^P = 4^{C \cdot P}$ . This is the option chosen in [8] with the ports concept implemented using set variables. Another possibility is to define a variable  $e_{i,j}$  that represents the choice of connecting the machine  $i$  with the printer  $j$ . Those  $P \times C$  variables are boolean. The search space is of size  $2^{P \cdot C}$ . There is however a drawback in using standard CSPs: generally, a solution contains fewer objects than the maximum number you had to define at start. Unused objects yield unwanted combinatorics and filtering. This is why extensions to the CSP formalism have been proposed [3, 5] that allow to add variables during the resolution. The construction of a vertex-colored DAG that we consider in this paper captures this in an abstract way.

**Definition 3 (Isomorphic configurations).** Two configurations  $G=(t, X, E, L)$  and  $G'=(t', X', E', L')$  are isomorphic iff  $t=t'$ ,  $L=L'$  and there exists a one-to-one mapping  $\sigma$  between  $X$  and  $X'$  such that  $\forall x,y \in X, (x,y) \in E \Leftrightarrow (\sigma(x), \sigma(y)) \in E'$  and  $\forall (x,l) \in L, (\sigma(x),l) \in L'$ .

For instance the two solutions at the bottom of Figure 1 are isomorphic since  $\sigma =((1,1), (2,4), (3,5), (4,2), (5,3), (6,6))$  is a one-to-one mapping satisfying the definition criterias.

Testing whether two graphs are isomorphic is an NP problem until today unclassified as either NP-complete or polynomial. The corresponding *graph isomorphism complete* class holds all the problems having similar complexity<sup>3</sup>. For several categories of graphs, like trees of course but also graphs having a bounded vertex degree, this isomorphism test is polynomial [11]. The graph iso problem is known as weakly exponential, and there exists practically efficient algorithms for solving it, the most efficient one being Nauty[12]. This being said, we must emphasize the fact that Nauty cannot be used in our situation. The reason is that we must maintain the property that all canonical structures can be obtained from at least one smaller solution itself being canonical. Using Nauty from within an arbitrary graph enumeration procedure yields a generate and test algorithm: the portions of the search space that can be explored by adding to a non canonical structure must still be generated, in case they would contain canonical representatives which cannot be obtained differently. This situation will be explained in more detail in a forthcoming section.

An *isomorphism class* represents a set of isomorphic graphs. All the graphs from a given isomorphism class are equivalent, therefore a graph generation procedure should ideally generate only one *canonical* representative per class. This is of crucial importance since the size of an isomorphism class containing graphs with  $n$  vertices can be up to  $n!$  (the number of permutations on the vertex set that actually create a different graph). Isomorphism classes are huge in size in most cases because, counter-intuitively, the less symmetrical a graph is, the more isomorphic graphs it has. This means that when current configurators (which do not avoid isomorphisms or in a very restricted way) generate a solution, partial or complete, they also generate an often exponential number of isomorphic solutions.

Most graph generating procedures rely upon the central operation of adding an edge to an existing graph (*unit extension*). Starting from a graph, the unit extensions that are valid wrt. the model constraints yield a new set of graphs. We consider two distinct situations for inserting an edge in a graph: an *internal edge* connects two existing vertices, whereas an *extraneous edge* connects an existing vertex to a newly created one. Having an efficient canonicity test is of little help for generating canonical graphs. Testing graphs for canonicity can be used to reject redundant solutions, but in so doing one has to explore the entire search space. Ideally we should be able to directly reject graphs for which we know that all extended supergraphs are not canonical. To achieve this, the canonicity criterion must be defined in such a way that *any canonical graph has at least a canonical subgraph resulting from the removal of one of its edges*. We call this the *canonical retractability property*. This condition is necessary (but not sufficient, see below) to allow for backtracking as soon as a non-canonical graph is detected during

<sup>3</sup> For instance, the vertex-colored DAG isomorphism problem.

the search. Indeed if there exists a canonical graph not obtainable via extension of a canonical subgraph, the extension of a non-canonical graph will be needed to reach it. Such a canonicity criterion is not trivial to find, and most known canonicity tests, Nauty inclusive, do not respect it. There exist isomorphism-free graph generation procedures that impose conditions on the canonicity test, as for instance the *orderly algorithms* from [13] which however do not propose an efficient canonicity test. To the best of our knowledge, such an efficient test has not yet been found in the general case (if ever one exists). Specialized and efficient procedures for generating canonical graphs exist for trees, for cubic graphs [14] and more generally, for graphs having hereditary properties<sup>4</sup> [15]. Configuration problems unfortunately do not comply with these restrictions, which led us to develop specific procedures. In order to achieve this, we have based our research upon existing work around configuration problems.

## 2.1 Related work in CSP and configuration

There exists a large body of work on symmetry elimination methods for CSPs (eg, [16–20]). Unfortunately, transposing those techniques to graph generation is far from obvious. The common principle for symmetry breaking in CSP is to avoid generating two isomorphic partial solutions: either by adding additional constraints to the problem, or by checking during resolution whether an isomorphic partial solution has already been generated. Our approach would be close to the first method as our canonicity criteria is defined beforehand according to the particular context of graph generation. However, if we were to transpose the graph generation problem to the CSP formalism we would have to deal with a dynamic CSP containing particular constraints (the structural constraints), and this would make the comparison with symmetry breaking methods in CSP very difficult. This work is connected to graph isomorphism detection techniques rather than CSP symmetry breaking methods.

Several approaches were experimented to tackle configuration isomorphisms, mostly by reasoning at a single level. One possibility is to prevent redundant connections of interchangeable objects during search. Also experimented is the substitution of connecting actual objects by counting them according to their target types [8]. A pseudo-linear time canonicity test that complies with the canonical retractability property is given in [9] when the configuration problem only involves composition relations (in which case all structural solutions are trees). This result was generalized to generic configuration problems in [10], by describing a weak canonicity criterion compatible with canonical retractability, in the case of DAGs. However, not all configuration generation procedures are compatible with this canonicity criterion. This important aspect was left unmentioned in our previous papers in order to simplify our point by restricting ourselves to explaining the main ideas. Now, we go into the details of this practical aspect.

## 2.2 State graph of a configuration problem

Let us consider the *state graph*  $G_P = (X_P, E_P)$  of a configuration problem. The state set  $X_P$  contains all structures (vertex-colored DAGs) corresponding to the structural

---

<sup>4</sup> A graph property is hereditary if all its subgraphs respect it.

model, and  $E_P$  are all the pairs  $(g, h)$  such that  $g, h \in X_P$  and  $h$  is the result of a unit extension from  $g$ . ( $G_P$  is itself a DAG for which the root is the state  $(t, \{1\}, \emptyset, \{(1,t)\})$ ). A structure generation procedure must be complete and non-redundant, i.e. able to generate all structures of  $X_P$  only once while exploring  $G_P$ . The search can be represented with a covering tree  $T_P$  of  $G_P$ . Let us consider now the state graph  $G'_P$ , which is the subgraph of  $G_P$  containing only canonical structures. The canonical retractability property ensures that  $G'_P$  is connected and therefore the existence of at least one complete search procedure able to backtrack on non-canonical graphs. However, this does not imply that all search procedures will meet the requirements if the intersection  $T'_P$  between  $T_P$  and  $G'_P$  is not a connected graph, backtracking on non-canonical structures will yield an incomplete procedure. As a consequence,  $T'_P$  must be a covering tree of  $G_P$ . We will now present procedures respecting these criteria.

### 3 Isomorph-free tree structure generation

We present a generation procedure for canonical configurations that can be used when the structural model only contains composition relations. A *composition* relation between type  $T_1$  (called *composite*) and another type  $T_2$  is a binary relation specifying that any  $T_2$  instance can connect to at most one  $T_1$ . As an example, the relation between  $N$  and  $C$  in Figure 1 is of the composition kind, although this is not the case for the relation between  $C$  and  $P$ . In the composition case, solutions to the configuration problem can only be trees.

```

procedure generate( $T, F$ )
  if canonical( $T$ ) then output  $T$ ; return; endif
  // generate the set  $E = \{(x_1, y_1), \dots, (x_{|E|}, y_{|E|})\}$  of acceptable unit extensions
   $E = \text{extensions}(T, F)$ 
  for  $i := 1$  to  $|E|$  do
    generate( $T \cup \{(x_i, y_i)\}, F \cup \{(x_1, L(y_1)), \dots, (x_{i-1}, L(y_{i-1}))\}$ )

```

**Fig. 2.** The procedure **generate**. To generate all trees, the initial call is `generate((t, {1}, ∅, {(1,t)}), {t})`

The procedure listed in Figure 2 is complete, non-redundant and generates exclusively canonical structures. The function `extensions(T, F)` returns the sequence  $E$  of unit extensions for  $T$  that are compatible with the structural model and not forbidden by  $F$ . Then, set  $E$  contains extraneous edges  $e_i$  linking two vertices of the object set  $O$ : one vertex was already in the tree  $T$  whereas the other extremity has been created. All unit extensions that must be discarded are stored in parameter  $F$ . This avoids generating the same tree multiple times. Such a redundancy would happen if starting from  $T$ , we first produced  $T_1$  by adding  $e_1$  and  $T_2$  by adding  $e_2$ , then later adding  $e_2$  to  $T_1$  and  $e_1$  to  $T_2$ , resulting in producing the same tree twice. In order to avoid this, we split the search into extensions of  $T \cup \{e_1\}$ , and extensions of  $T \cup \{e_2\}$  with  $e_1$  removed

from possible extensions. In more precise terms, not only  $e_1=(x, y)$ : for all  $z$ , we forbid to add edge  $(x, z)$  if  $L(y)=L(z)$ . Even if those two trees are different, they are isomorphic since swapping  $y$  and  $z$  yields the other. All such pairs  $(x, L(y))$  are members of  $F$ , which forbids adding an edge connecting an object  $x$  to a new object of type  $L(y)$ . In the general case, starting from a given tree there exist  $|E|$  possible extensions. We hence split the search into  $|E|$  parts by calling

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

The edge sequence in  $E$  could be arbitrary if we didn't need to remove non canonical trees. However, as seen at the end of section 2, it has to be chosen according to the canonicity criterion to ensure completeness. We hence sort trees according to the total order  $\preceq$  from [9], and define as canonical a tree being the  $\preceq$ -minimal in its isomorphism class. [9] proves that this canonicity criterion has the canonical retractability property. To ensure completeness, the edges of  $E$  must be sorted as follows: edge  $e_i$  is before  $e_j$  in  $E$  iff  $T \cup \{e_i\} \preceq T \cup \{e_j\}$ .

**Proposition 1.** *The procedure generate is complete.*

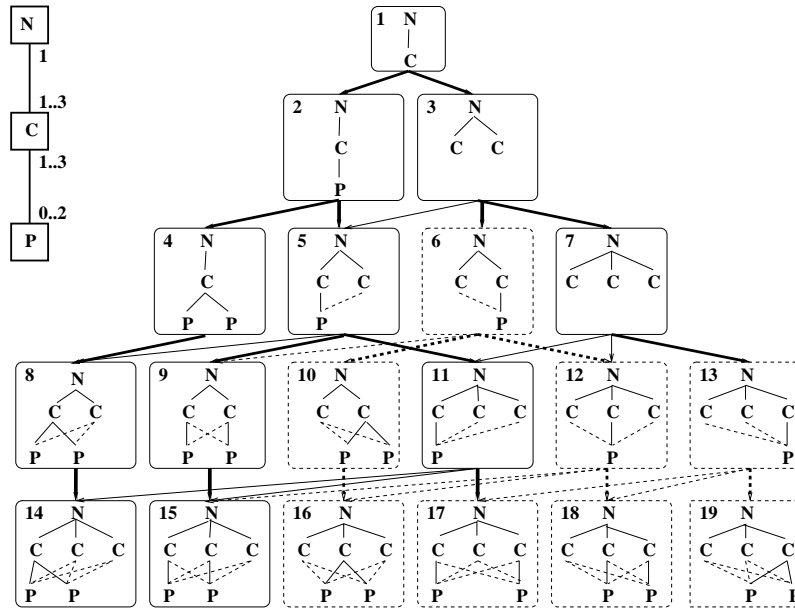
*Proof.* (sketch) We first inductively show that the edges are added by connecting new vertices to the rightmost branch of a tree, starting from the deepest vertex, and going up to the root vertex. This is true of the tree made of a single vertex (the start of a configuration). If the property holds for any tree  $T$  having  $n$  vertices, it means that the vertex  $y$  connected to an  $x$  in the right branch of the previous tree to form  $T$  now is the extremity of this branch. By inserting  $x$ , we lost the capacity to perform unit extensions to vertices located below  $x$  (by completing the set  $F$ ). The sole remaining possibilities are parent nodes to  $x$ , as well as  $x$  and  $y$ , hence all the vertices in the right branch of  $T$ .

Now, from Proposition 6 in [21], we know that removing a node from the right branch preserves canonicity. As a consequence, since from any tree  $T$ , the procedure generate produces all  $T$  extensions such that the removal of their rightmost branch would yield  $T$ , it produces all the canonical trees that can be obtained by unit extension from  $T$ . The procedure is hence complete.

In our network example, if we restricted printers to be connected to at most one computer, it would become a composition relation. Then, the structural solutions of our problem would necessarily be trees. Figure 3 helps view the search tree that would result from this. The procedure would backtrack on non canonical trees 6, 13 and 17. As a consequence, the non canonical trees 10, 12, 16, 18 and 19 would not be generated either.

## 4 Isomorph aware DAG generation

We now present an instance of a procedure generating only what we will call *weakly canonical DAGs*, defined as DAGs for which the minimal covering tree for the order  $\preceq$  is canonical. As the permutation that would make its covering tree canonical is the



**Fig. 3.** A portion of the state graph for the network configuration problem. Nodes are labeled with their type alone. Trees framed in dotted lines are not canonical. Dotted lines joining nodes inside frames denote possible complementing internal edges. All edges of the state graph denote unit extensions. Edges between non canonical trees are dotted. Bold edges are explored by procedure without canonicity check. Only continuous and bold edges are transitions explored by the procedure generate.

same that would make the DAG weakly canonical, this avoids generating all non weakly canonical DAGs<sup>5</sup>.

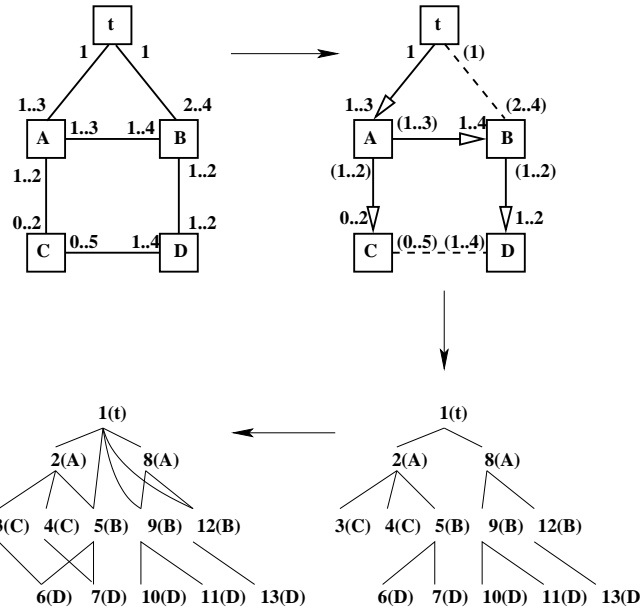
The leading idea is to first generate a canonical tree, called the *structure tree*, then perform unit extensions that solely create internal edges. As presented before, we can generate all canonical trees very efficiently. From such canonical trees, we generate all the DAGs sharing it as a structure tree, by adding internal edges.

Figure 4 illustrates this idea. We start from a structural model containing general binary relations, from which we extract a sub-model having only composition relations<sup>6</sup>. The trees solution of this sub-model can be completed to produce solution DAGs of the original problem.

This procedure must however be implemented carefully to prevent from generating the same DAG multiple times. First, the possible extensions of a tree are ordered according with some order  $<$ . Edges are always added according with  $<$  and an edge  $e$  cannot be added anymore if there exists an edge  $e'$  already added and  $e < e'$ . As for trees, it is obvious that this discards a certain amount of redundancies. Let  $a$  be the set of possible internal edges on a tree  $T$ , the number of DAGs that can be generated from  $T$  will be  $2^{|a|}$  instead of  $|a|^{|a|}$ . This however does not suffice to remove all DAG

<sup>5</sup> The tractable generation of only one DAG per isomorphism class is an open problem.

<sup>6</sup> This sub-model is a covering tree of the original model!



**Fig. 4.** Generating DAGs from trees. To the upper left, a structural model. To the upper right, a composition covering tree of the model. To the bottom right, a possible solution of the relaxed model. To the bottom left, a corresponding real solution after tree completion.

redundancies. To achieve this, and for each newly generated DAG, we search for the existence of a covering tree being ( $\preceq$ ) less than the current structure tree, but not necessarily canonical. If such a covering tree exists, it means that the current DAG can be discarded whether the found covering tree is canonical or not<sup>7</sup>. Our procedure for finding such covering trees has the complexity of depth-first search in the worst case:  $O(n)$ .

#### Alternative structure tree search algorithm

At each newly created DAG (generated from tree  $T$ ), we build the canonical covering tree  $T'$  by doing a depth-first search on the DAG. If at one point, the selected edge differs from  $T$ , the DAG is rejected as it means the current working tree  $T$  is not the canonical one anymore. For instance, in the tree number 15 in Figure 3, the internal edge connecting the first  $C$  to the second  $P$  must not be inserted, since the smallest ( $\preceq$ ) covering tree becomes the tree number 14.

**Proposition 2.** *Our procedure generate with a call to completion (see fig. 5) at each canonical tree generates only once each weakly canonical DAG.*

<sup>7</sup> There exists a canonical tree that is isomorphic to it, and thus the current DAG (or an isomorphic one) is already obtained by completion when this canonical tree is generated (and our tree generation procedure ensures that it has been or will be generated during the search).

```

procedure completion(G, F)
  output G
  // generate the set E=(e1, ...e|E|) of acceptable unit extensions not in F
  E = internal-extensions(G, F)
  for i := 1 to |E| do
    completion(G ∪ {ei}, F ∪ {e1, ...ei-1})

```

**Fig. 5.** The procedure **completion**.

*Proof.* (sketch) The procedure `completion` never generates the same DAG twice from a given canonical tree and never a DAG that would result from the completion of another tree.

## 5 Exploiting symmetries

The procedure `completion(G)` can be further improved to eliminate some isomorphic DAGS resulting from unit extensions. The intuition is as follows: if the internal edges  $e_1$  and  $e_2$  that can complete  $G$  lead to two isomorphic graphs  $G_1$  and  $G_2$ , then we forbid the unit extension  $e_2$ .

For example, adding the edge (4,3) to the DAG on the bottom right of Figure 1 produces a DAG isomorphic to the one obtained by adding edge (6,3). We might want to avoid one of the two extensions.

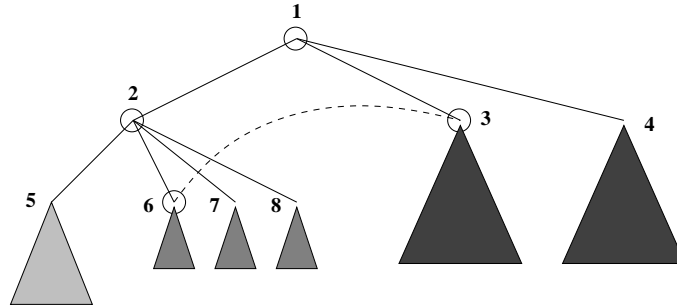
One expensive approach is to consider each pair of graphs completed with an edge from the set  $E$  of valid extensions, and test whether they are isomorphic or not (using Nauty for instance). In case they are, we delete from  $E$  one of these edges. The major drawback of this method is that there are potentially  $O(n^2)$  unit extensions for a graph with  $n$  nodes, that is  $O(n^2)$  that can be canonically labelled (thanks to Nauty for instance), thus leading to  $O(n^4)$  pairs of canonical graphs to be compared (or  $O(n^2 \log n)$  comparisons if we sort the graphs). In addition, even if Nauty has a polynomial behavior on most graphs, it still has an exponential complexity in the worst case which disqualifies its use for large configuration problems.

We henceforth use an incomplete method for removing such isomorphisms, by using the automorphism group (ie, the set of symmetries) of the current DAG: the covering trees of the DAGS are canonical, hence all their subtrees are  $\preceq$  sorted. Henceforth, at any level in the tree, there may exist nodes equal wrt.  $\preceq$ . They are interchangeable, and are immediate neighbors, and all their sub-trees are pairwise interchangeable.

Although node interchangeability is costly to detect in the general case of unrestricted graphs, it is fast and obvious in the case of canonical trees. Testing whether two sub-trees having the same parent are interchangeable simply consists in testing if they are identical, an operation of time linear complexity. As a consequence, marking which node pairs are interchangeable in a tree is an operation in  $O(n^3)$  that can be done at once before the completion of a structure tree.

To account for the fact that interchangeability is lost by nodes newly connected by an internal unit extension, we introduce a Boolean marker. The connected nodes must

be marked, as well as the whole list of their parents up to the root of the tree. The marking is illustrated in Figure 6 by small circles around the nodes. A search procedure can reject all DAGs in which a newly inserted internal edge results in marking a node not being the leftmost in its equivalence class of interchangeability.



**Fig. 6.** Adding an internal edge and marking

In the canonical tree represented by Figure 6, the trees rooted in nodes 6, 7 and 8 are identical, and so are the trees rooted in nodes 3 and 4. If the choice of interconnecting nodes from this two groups must be made, the search procedure can select only nodes within the trees 3 and 6. No node appearing within the sub-trees rooted in 4, 7 and 8 can be connected by a newly inserted internal edge. Once a connection between 3 and 6 is established for instance, node 3 loses its interchangeability with 4, and 6 loses its interchangeability with 7 and 8.

## 6 Experimental Results

Our experiments were conducted for the computer-printer planning problem illustrated in Figure 1, on a 1.7 Ghz PC with 512M RAM, under Linux. We have chosen this simple problem because it is generic: it involves a cardinality constrained relation between two types, which occurs very frequently in configuration problems. It must not be seen as a real application example, but rather as a way to reveal the interest and efficiency of such a procedure for eliminating isomorphisms. Indeed, the results on real problems involving many relations would benefit from the gain on each relation. For each choice of numbers of printers and computers, we have generated all DAGs using two algorithm variants: *Covering Tree* or *ct* (generation of canonical trees, each being completed to DAGs using an ordered set of possible extensions and backtrack on DAGs that have a covering T-tree less than the current) and *full* (*ct* + backtrack on equivalent internal edges for interchangeability). We compare the number of graphs generated by both algorithms with the number of graphs that are a solution of the problem. There are as many of them as the number of bipartite graphs (canonical or not) joining a set of  $c$  vertices to  $p$  vertices:  $2^{c \cdot p}$ .

From Table 1 we see that the number of DAGs is significantly decreased when using the *ct* algorithm, due to the large number of avoided isomorphic DAGs. The *full* algorithm provides a good cut in the number of isomorphic DAGs, and overall computation

**Table 1.** Results for the (C) PC - (P) printers problem. ( times in seconds, ”/” = time > 60 seconds)

C	P			<i>ct</i>		<i>full</i>	
		all graphs	structure trees	graphs	time	graphs	time
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 <sup>6</sup>	649	993197	8.34	329495	3.43
8	3	1.6 10 <sup>7</sup>	1006	/	/	1.45 10 <sup>6</sup>	17.23

time is also noticeably decreased.

Existing configurators are restricted to problems of limited size. Using these strategies lets us address larger problems, while avoiding the generation of useless solutions. Our computer/printer test problem should not be seen as artificial: any binary relation in an object model implies that a certain number of structures contain bipartite sub-graphs. The canonicity test for such graphs is graph iso complete, and current configurators would generate the graphs corresponding to the *all graphs* column of Table 1. These early results show that we can generate significantly fewer DAGs when the model involves only one binary relation. Should there be more than this (this is the common situation), the overall gain factor would benefit from individual gains, and in the particular case of a tree structural model it would be the product of the gains on each relation.

### Insertion in a general configuration search

A configuration problem statement normally involves classes, relations, and constrained attributes. Generating the configuration structure is hence a fragment of the whole problem. Our approach is interesting in several respects in this general case. On the one hand, once a structure has been generated, the problem amounts to a standard CSP, hence amenable to usual techniques (including incomplete search methods). Also, as shown before, the automorphism group of the structure built is easily exploited. Further search may benefit from this in the process of instantiating attributes as well.

## 7 Conclusion

This work greatly extends the possibilities of dealing with configuration isomorphisms, until today limited either to the detection of the interchangeability of all yet unused individuals of each type or to the use of non configurable object counters. The generation procedures for tree-shape and vertex colored DAG structures that we have presented addresses the structural isomorphism problem of configurations and allows for important gains for any configuration problem, even of small size. Not all the non canonical structures are discarded in the general case of DAG structures. Polytime methods for eliminating more isomorphisms probably exist.

## References

1. McDermott, J.P.: R1: A rule-based configurer of computer systems. *Artificial Intelligence* **19** (1982) 39–88
2. Barker, V., O'Connor, D., Bachant, J., Soloway, E.: Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM* **32** (1989) 298–318
3. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proc. of AAAI-90, Boston, MA (1990) 25–32
4. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic cpsp-application to configuration. *Artificial Intelligence* **135** (2002) 199–234
5. Sabin, D., Freuder, E.C.: Composite constraint satisfaction. In: *Artificial Intelligence and Manufacturing Research Planning Workshop*. (1996) 153–161
6. Soininen, T., Niemela, I., Tiihonen, J., Sulonen, R.: Representing configuration knowledge with weight constraint rules. In: Proc. of the AAAI Spring Symp. on Answer Set Programming: Towards Efficient and Scalable Knowledge. (2001) 195–201
7. Stumptner, M.: An overview of knowledge-based configuration. *AI Communications* **10(2)** (1997) 111–125
8. Mailharro, D.: A classification and constraint-based framework for configuration. *AI in Engineering, Design and Manufacturing*, (12) (1998) 383–397
9. Grandcolas, S., Henocque, L., Prcovic, N.: A canonicity test for configuration. In: *Proceedings of CP'2003*. (2003)
10. Henocque, L., Prcovic, N.: Practically handling configuration automorphisms. In: *proceedings of the 16th IEEE International Conference on Tools for Artificial Intelligence*, Boca Raton, Florida (2004)
11. Luks, E.M.: Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. System Sci.* **25** (1982) 42–49
12. McKay, B.D.: Practical graph isomorphism. *Congressus Numerantium* **30** (1981) 45–87
13. Read, R.C.: Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics* **2** (1978) 107–120
14. Brinkmann, G.: Fast generation of cubic graphs. *J. Graph Theory* **23** (1996) 139–149
15. McKay, B.D.: Isomorph-free exhaustive generation. *J. Algorithms* **26** (1998) 306–324
16. Pascal Van Hentenrick, P. Flener, J.P., Agren, M.: Tractable symmetry breaking for cpsps with interchangeable values. In: *proceedings of IJCAI 03*. (2003) 277–282
17. Meseguer, P., Torras, C.: Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence* **29(1-2)** (2001) 133–163
18. Backofen, R., Will, S.: Excluding symmetries in constraint-based search. In: *Principles and Practice of Constraint Programming*. (1999) 73–87
19. Gent, I., Smith, B.: Symmetry breaking during search in constraint programming. In: *proceedings of ECAI*. (2000)
20. Puget, J.F.: Symmetry breaking revisited. In: *proceedings of CP'02*. (2000)
21. Grandcolas, S., Henocque, L., Prcovic, N.: Pruning isomorphic structural sub-problems in configuration. Technical report, LSIS (2003) Available from the CoRR archive at <http://arXiv.org/abs/cs/0306135>.