Practically Handling Some Configuration Isomorphisms

Laurent Henocque and Nicolas Prcovic
Laboratoire des Sciences de l’Information et des Systèmes, LSIS (UMR CNRS 6168), Campus Scientifique de Saint Jérôme, Avenue Escadrille Normandie Niemen, 13397 MARSEILLE Cedex 20

Abstract

Configuring consists in simulating the realization of a complex product from a catalog of component parts, using known relations between types, and picking values for object attributes. A configuration can be viewed as a graph of interconnected components. An inherent difficulty in solving configuration problems is the existence of many structural isomorphisms. A practical way of dealing with isomorphisms is by isolating one configuration in each equivalence class called a canonical representative. Since no polytime algorithm is known for the general graph isomorphism problem, it is interesting to explore sub-problems that can efficiently be exploited by backtracking search procedures. We describe a formalism independent approach to detect a large number of structure related configuration isomorphisms. The algorithm has the essential property that isomorphism detection is both incremental and can be performed in pseudo linear time, two essential conditions for backtracking search. Backtrack occurs as soon as a non weakly canonical DAG structure is generated for a configuration, which allows to extend the range of practically tractable problems, as shown experimentally. Weakly canonical configurations explicitly expose their automorphism group, which is readily available thanks to the lexicographic ordering chosen. The efficiency of this approach is assessed both theoretically and by experimental results obtained for a range of realistic configuration problems.

1. Introduction

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, hence placing ourselves outside the scope of traditional symmetry elimination methods for CSPs [3, 6, 12, 17, 16], which expect a problem having a fixed size (i.e. number of variables). This work builds upon the results published in the short paper [7]. The original contribution in this early work was a pseudo linear canonicity testing procedure for configuration tree structures allowing for early backtracks as soon a non canonical structure is being generated. We wish to maintain in the sequel the essential two aforementioned properties that a/ canonicity testing must remain tractable 1 (in comparison the general graph iso problem belongs to NP, in its own graph iso complete class), and b/ that the backtrack search procedure that exploits the test must be able to backtrack as soon as a non canonical structure is being generated, so as to avoid a “generate and test” search pattern.

This paper’s first contribution is to provide experimental evidence of the importance of the results in [7] where only theoretical claims where made. We do so by studying the efficiency of a configuration generator on a realistic problem. Experimental results confirm the practical impact of the theoretical combinatorial bounds presented in [7]. Our second contribution is to generalize the application range of the canonicity test previously proposed for configuration structures being trees to directed acyclic graphs. It should be noted that extending from trees to DAGs cannot be expected to be trivial, specially under the objectives previously listed. Extending to DAGS allows for capturing many more configuration isomorphisms since it partly releases the restriction that isomorphism testing only concerns composition relations, which are responsible for the existence of a rooted tree structure in configurations. This can be achieved while retaining an essential property : canonicity testing can be used at all the nodes of the search tree in a backtrack search procedure and trigger immediate backtrack in case of non canonicity.

1 Practical experience in constraint programming systems has shown that algorithms with a complexity exceeding pseudo linearity are of seldom interest when called at all search nodes
In order to backtrack at non canonical nodes, we require that any canonical configuration can be reached by a continuous sequence of canonical configurations that differ by a canonical unit extension (adding a single edge to a new object). It must be emphasized that to the best of our knowledge no other (eventually more general) definition of graph canonicity, as for instance the one from Nauty [11] has the same property. Using such a system in a backtrack search procedure does not grant the right to backtrack the search as long as the entire graph structure has not been generated. Our experimental results clearly disqualify this option as they differ by up to six orders of magnitude, even on small problems. So even though our approach is less general than full graph isomorphism detection, it is tractable (computable in pseudo linear time), and it allows to cut entire portions of the configuration structure generation search space.

1.1. Configuration

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 [10, 4], and has triggered the development of many configuration programs and formalisms [13, 20, 1, 14, 19, 21, 22, 9, 15].

Configuration isomorphisms belong to two distinct categories. Given a relation structure (i.e. known links between components), the problem becomes a standard CSP amenable to standard CSP symmetry elimination techniques [3, 6, 12, 17, 16]. But there also exist graph related isomorphisms among structures that none of these CSP methods can easily address in a truly generative framework. The problem is to efficiently avoid generating multiple solutions having isomorphic relation structures, which is our current field of study.

1.2. An example

When an architect designs a building, he must configure wrt. many design and customer constraints. This kind of problem has already been addressed using CSP based techniques [5], but it deserves a more dynamic approach as soon as we do not fix the numbers of floors, of rooms per floor, nor of windows and doors per room. This configuration problem is well described using the object model presented in figure 1. Multiplicity constraints are explicit in such a model: a building has at most $F$ floors, each floor owns one to $R$ rooms, each room has one to three doors and at most four windows.

![Figure 1. A simplified multi story building model](image)

The solutions to configuration problems involve interconnected objects, as illustrated in the figure 2, which makes explicit the existence of structural isomorphisms. We propose a general approach for the elimination of such isomorphisms in configuration problems, which generalizes existing methods (the interchangeability of "unused" objects, or the use of cardinality counters) while not requiring to adapt the configuration model, and extends a strategy successfully applied to finite model search [2].

![Figure 2. Two isomorphic configuration trees. The letters F, R, D, W respectively stand for floor, room, door, window.](image)

1.3. Plan of the article

The section 2 describes the formalism used throughout the paper and structural sub-problems (section 2). Since our definition of T-dag canonicity exploits covering T-trees, and in order of keeping the paper self contained, section 3 presents T-trees and their canonical representatives. The section 4 proposes an algorithm to test the canonicity of configurations. A generalization to non composition relations exploiting the automorphisms present in canonical configurations is described in section 5. Section 6 provides experimental results on the benefits of using such an ap-
proach. Section 7 compares our work with other approaches and section 8 concludes.

2. Configuration problems, and structural sub-problems

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 composition relations (an object is a component of at most one composite). We isolate configuration sub-problems called structural problems, that are built from the composition 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 $O = \{1, 2, \ldots\}$), a totally ordered set $T_C$ of type symbols (unary relations) and a totally ordered set $R_C$ of composition relation symbols (binary relations). We note $\prec_O$, $\prec_{T_C}$ and $\prec_{R_C}$ the corresponding total orders.

**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 $R_C$.

In the spirit of usual finite model semantics, $T_C$ members are interpreted by elements of $\mathcal{P}(O)^2$, and $R_C$ members by elements of $\mathcal{P}(O \times O)$ (relations).

**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 (or model) of the structural problem.

We use the term configuration to denote a structural problem model.

**Definition 3 (root, composite, component)** A configuration, solution of a structural problem $(t, T_C, R_C, C)$, can be described by the set $U$ of interpretations of all the elements of $R_C$. If $R_U$ denotes the union of the relations in $U$ ($R_U = \bigcup_{r \in U} r$), and $R_t$ denotes its transitive closure, then we have:

1. \( \exists \text{ root} \in O \text{ called root of the configuration}^3 \) for which \( \forall o \in O \ (o, \text{ root}) \notin R_U \).
2. \( \forall o \in O \text{ s.t. } o \neq \text{ root}, \exists c \in O \text{ s.t. } (c, o) \in R_U \); we call $c$ the composite of $o$ and $o$ a component of $c$.
3. \( \forall o \in O \text{ s.t. } o \neq \text{ root}, (\text{root}, o) \in R_t \).

2 the set of all subsets of $o$
3 root unicity does not restrict generality.

**Definition 4** We note $U(r)$ the relation interpreting the relational symbol $r \in R_C$ in $U$. Two configurations $U$ and $U'$ are isomorphic if and only if there exists a permutation $\theta$ over the set $O$, such that $\forall r \in R_C, \theta(U)(r) = U'(r)$

3. Basic notions : T-trees

For paper self containment reasons, and since the definition of T-dag weak canonicity exploits these notions this section recalls the definition of T-trees and their properties. Because composition relations bind component objects to at most one composite object, their configurations can naturally be represented by trees where nodes are labeled by objects of $O$, edges are labeled by the component side type of the corresponding relation, and child nodes are sorted first by their type according to $\prec_{T_C}$, then by their label according to $\prec_O$. The figure 2 shows that object numbers are redundant. We thus introduce T-trees (illustrated in figure 3):

**Definition 5 (T-tree)** A T-tree is a finite and non empty ordered tree where nodes are labeled by types and children are ordered according to $\prec_{T_C}$. We note $(T, (c_1, \ldots, c_k))$ the T-tree with sub-trees $c_1, \ldots, c_k$ and root label $T$.

![Figure 3. The first 25 T-trees ordered by $\preceq$ for a sample problem where the root A can involve up to two B’s and two C’s and where a B can involve up to 2 D’s. The $\preceq$-minimal representatives are framed.](image)

**Proposition 1** Let $A_1$ be a configuration tree, $C_1$ the corresponding T-tree. Then any configuration tree $A_2$ rebuilt from $C_1$ is isomorphic to $A_1$.

Configuration trees and T-trees being trees, they are isomorphic, equal, superposable, under the same assumptions as standard trees. We have the following corollary:
Proposition 2 Two configurations are isomorphic iff their corresponding $T$-trees are isomorphic (two $T$-trees are isomorphic if there exists a set of permutations of their lists of subtrees that makes them identical).

As a means of isolating a canonical representative of each equivalence class of $T$-trees, we define a lexical total order $\preceq$ over $T$-trees. Since we are dealing with trees having nodes labeled with the object types, this lexical ordering can be presented in several different ways. The reader is kindly directed to [7, 8] for details and proofs.

Definition 6 (Canonicity of a $T$-tree) A $T$-tree $C$ is canonical iff it has no child or if $\forall i$, $T_i(C)$ is sorted by $\preceq$ and $\forall c \in T_i(C)$, $c$ itself is canonical.

Proposition 3 A $T$-tree is the $\preceq$-minimal representative of its equivalence class (wrt. $T$-tree isomorphism) iff it is canonical.

4. Algorithms

A test of canonicity straightforwardly follows from the definition of canonicity. It is defined by two functions, Canonical and Less, listed in pseudo code by the figures 5 and 4. We note $ct(T)$ the list of component types of $T$, sorted according to $\prec_{T}$, and by extension, as the labels of nodes of a $T$-tree are types, we generalize these notations to $ct(C)$ for a given $T$-tree $C$. The worst case complexity of Less is $\Theta(n)$, $n$ being the number of nodes of the smallest $T$-tree. Canonical is of complexity $\Theta(n \log n)$.

![Figure 4. The function Less](image)

The function Less generalizes over the corresponding definition in [7] by gathering the information that its argument $T$-trees are equal. Less returns INF if $C \not\preceq C'$ but not $C \preceq C'$ et EQ when its argument $T$-trees are identical.

The list of $T$-trees in figure 3 illustrates the behaviour of the functions Less and Canonical. The $T$-trees are listed in increasing lexicographic order, hence according to Less. Canonical $T$-trees are the least (according to Less) in their equivalence class. For instance, the $T$-trees $\#10$ and $\#12$ are obviously isomorphic, and the first one is retained as canonical. Note that in the $T$-tree $\#10$ the two B-rooted sub-trees are themselves listed in increasing order (a consequence of $\preceq$ being lexicographic).

5. Tractable weak canonicity in the general case

A configuration problem where only composition relations are involved can be filtered for isomorphisms by a constraint implementing the canonicity test. However, practical configuration problems also involve non composition relations. For instance, a relation stipulating that $m$ objects of a given type may connect to at most $n$ objects of another type yields a bipartite graph structure. This situation is illustrated by the figure 7.

Because edges can be added to a structure between pre-existing nodes, non composition relations yield plain graphs as their models, which can be viewed as directed acyclic graphs (DAGs) if a root node is considered. Since any configuration problem can be adapted to involve such a root object, and without loss of generality, the full structural isomorphism problem for configurations can be viewed as a DAG isomorphism problem. This case is illustrated by the figure 6. We thus generalize to DAGs the former notions:

4 The bipartite graph isomorphism problem amounts to graph isomorphism.
**Definition 7 (T-dag)** A T-dag is a finite directed acyclic graph with nodes labeled by types and neighbors ordered according to $\prec_{T_C}$.

Testing the canonicity of a DAG amounts to the graph isomorphism problem, an NP problem not proven polynomial or NP complete. There exist practically efficient algorithms like Nauty [11], but their definition of canonicity does not match the crucial connexity requirement ensured in our case by the proposition 5 below. A backtrack search procedure for the enumeration of T-dags cannot hence remain complete if it fails when a non canonical structure in the sense of [11] is generated, which forbids using this definition. Despite these difficulties, we now show how the canonicity of T-trees can be easily exploited to achieve at low cost a weaker form of canonicity having useful properties, in the case of T-dags.

**Definition 8 (Unit extension, extraneous, structural edge)** A unit extension of a T-dag is obtained by adding a single edge, either between two existing nodes - this is an extraneous edge - or between a preexisting node and a new one - here called a structural edge.

**Definition 9 (structure T-tree)** The structure T-tree of a T-dag is the covering T-tree built from the sole edges having introduced a new node during its incremental constitution.

**Definition 10 (Weak canonicity of a T-dag)** A T-dag is weakly canonical if its structure T-tree is canonical.

**Proposition 4** Every T-dag has a weakly canonical isomorph.

**Proof 1** Let $D$ be a non weakly canonical T-dag. There exists a permutation of its nodes yielding a canonical equivalent of its structure T-tree. This permutation hence maps $D$ to an equivalent weakly canonical T-dag.

It follows that a search procedure which only generates weakly canonical T-dags remains complete. Unlike with T-trees however, two weakly canonical T-dags can be isomorphic: such a procedure does not fully prevent from generating some isomorphic configurations.

The search space of a configuration problem can be described by a state graph $G = (V, E)$ where the nodes in $V$ correspond to valid (solution) T-dags and the edges correspond to unit extensions. The goal of a constructive search procedure is to find a path in $G$ starting from the singleton node (root object) T-dag and reaching a T-dag which respect all the problem constraints.

**Proposition 5** Let $G$ be the state graph of a configuration problem. Its sub-graph $G_s$ obtained by removing all non weakly canonical T-dags is connected.

**Proof 2** It amounts to proving that any canonical T-dag can be reached by a sequence of canonical unit extensions from the starting T-dag (where only the root type occurs as a single node), or that (taken from the opposite side) the weak canonicity of any T-dag is preserved by removal of at least one of its edges. The operation of removing an extraneous edge obviously preserves canonicity since the structural T-tree remains unchanged. Furthermore, if a T-dag has no extraneous edge, it is a T-tree. The proposition hence holds because proposition 4 in [7] ensures that the state graph of canonical T-trees is itself connected.

The figure 3 practically illustrates this proposition in the case of T-trees: each canonical T-tree (the ones where the numbers are "boxed") can be obtained by a single canonical unit extension performed on a smaller canonical T-tree. For instance, the canonical T-tree #13 is obtained from the sequence of canonical T-trees #1, #2, #5, #6, #9 and #10.

It follows that a configuration procedure that backtracks as soon as a non weakly canonical T-dag is generated remains complete. In other words, it is possible to implement a constraint that discards all the non weakly canonical structures built as models of the relations.

We now present an instance of such a procedure generating only weakly canonical T-dags. The idea is simple: first generate the structure T-tree, then perform unit extensions that solely create extraneous edges. We can generate all canonical T-trees using an orderly algorithm as defined in [18], which backtracks as soon as a non canonical T-tree is produced. From such canonical T-trees, we generate all the T-dags sharing it as a structure T-tree, again using an orderly algorithm.

**Exploiting automorphisms**

This strategy can be further improved, so as to exploit the automorphism group of T-trees, which is made explicit in the structure of canonical T-trees. A canonical T-tree sorts its nodes according to the order $\preceq$. At any level in the tree, there may exist nodes being equal wrt. $\preceq$. Such nodes are interchangeable, and are immediate neighbors. We call a node class such a set of sibling nodes. The definition of $\preceq$ additionally ensures that the sub-trees rooted at these nodes are identical: all their nodes are pairwise interchangeable. It is obvious that a T-dag resulting from a unit extension involving a node $x$ interchangeable with a node $y$ is isomorphic to the T-dag obtained by the permutation $(x, y)$. One among both unit extensions therefore needs not to be generated. Although node interchangeability is costly to detect in the general case of unrestricted graphs, it is fast and obvious in the case of T-trees: the calls to Less performed by

---

5 Note that no accurate definition of canonicity is given for nauty which in that respect remains obscure (the reader is directed to the source code).
the function Canonical gather that information for use by constant time caches.

To account for the fact that interchangeability is lost by nodes newly connected by an extraneous 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 T-tree. The marking is illustrated in the figure 6 by small circles around the nodes. A search procedure can reject all T-dags in which a newly inserted extraneous edge results in marking a node not being the leftmost in its equivalence class.

Examples

In the canonical tree represented by the 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. Doing so preserves the fact that the remaining interchangeable nodes (e.g. here 7 and 8) are neighbors, hence the capacity of easily detecting interchangeability. No node appearing within the sub-trees rooted in 4, 7 and 8 can be connected by a newly inserted extraneous edge. More interchangeable nodes can occur in any sub-tree. 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.

The figure 7 gives an example of a state graph for a simple configuration problem involving computers and printers. The relation states that one to three computers can be linked with zero to two printers. This problem requires generating a connection structure in the form of a bipartite graph. This example shows that we can eliminate a significant number of redundant structures. Our experimental results show that the gains further increase quickly with the problem sizes. In this figure, only the structure T-trees are represented, but the extraneous edges that can be potentially added are drawn as dotted lines. The two framed structures are non canonical: the topmost hence yields a backtrack, and the "son" never gets generated. Within each T-dag, interchangeable nodes are shown using a circle around their class. For instance, in the last line, third tree from the left, the extraneous edge using the third "C" is forbidden, and the corresponding redundant DAG will never be generated.

Here, exploiting isomorphisms and explicit automorphisms allows for generating only 27 among the 35 possible T-dags.

6. Experimental Results

Our experiments where conducted for the floor planning problem illustrated in figure 1. In order to experimentally explore the combinatorial properties of the problem, we generalized the model so as to render it easy to parameterize. The numbers $F$ (counting the max number of floors in a building) and $R$ (the max count of rooms in a floor) are the parameters that we let vary in the experiments. We have made two groups of tests: without and with additional constraints. When run without constraints, the configurator produces all configurations including isomorphs. This allows to count them, compare their number with the total count of their canonical representatives, and observe that the cost of testing the canonicity is negligible compared with the benefits. In other experiments we added a global constraint stating that the total number of windows in the building is fixed to a parameter $W$. This is realistic since most often a building is designed for a given usage, and the number of windows will only slightly vary, unlike the possible floor or room count. The parameters $W, F, R$ allow for the gener-
ation of many different problems, varying in size (by \( F, R \)) and in hardness to solve (by \( W \)). We guess that such a simple model can serve as reference testbench for other configurator programs.

We have written a configurator which generates all the possible solutions to the problem described before, according to the parameters \( W, F, R \). Highly constrained problems are obtained by setting the parameter \( W \) to the value 2. With \( W = 10 \), we obtain more realistic, yet still constrained problems. Tables 1 and 2 list the results. The times are in seconds. Non measurable times are denoted with a "-". Results obtained after 10 minutes have not been listed. The computation times obtained when listing all (isomorph inclusive) solutions raise rapidly because of the tremendous number of such solutions, even for small problems, and are in no ways a consequence of poor program performance. It should be noted that using a canonicity definition not obeying the connected state graph property requires to pay that price, which is non realistic.

<table>
<thead>
<tr>
<th>( F )</th>
<th>( R )</th>
<th>( W = 10 )</th>
<th>( W = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>all solutions</td>
<td>no isomorphism</td>
<td>time</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>0.01</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>0.13</td>
<td>0.01</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>1.12</td>
<td>0.03</td>
</tr>
<tr>
<td>17</td>
<td>7</td>
<td>9.77</td>
<td>0.05</td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>72</td>
<td>0.09</td>
</tr>
<tr>
<td>17</td>
<td>9</td>
<td>0.16</td>
<td>0.51</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>4.4</td>
<td>4.4</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>152</td>
<td>0.70</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>9.4</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 2. Computation times (in seconds) obtained when varying the values of \( F, R \) and \( W \).**

As from these results, it appears that the elimination of isomorphic solutions benefits in all cases, whatever the problem size or difficulty. We note that the exponential factor is much more favorable when non canonical solutions are rejected by the constraint. These results confirm the theoretical observations made in [7].

7. Related work

Isomorphisms naturally arise in configuration from the fact that many constraints are universally quantified\(^7\). This

\(^7\) for instance: "for all motherboards, it holds that their connected processors have the exact same type"

The issue is technically discussed in several papers[9, 24, 23]. The most straightforward approach is to treat during the search all yet unused objects as interchangeable. This is a widely known technique in constraint programming, applied to configuration in [9, 23]. However, this does not account for the isomorphisms arising during the search because substructures are themselves isomorphic (e.g. two exactly identical PCs with the same motherboards and processors are interchangeable).

The work in [9], implemented within the ILOG\(^8\) commercial configurators, suggests to replace some relations between objects with cardinality variables counting the number of connected elements for each type. This technique is very efficient and intuitively addresses many situations. For instance, to model a purse, it suffices to count how many coins of each type it contains, and it would be lost effort to model each coin as an isolated object. This solution has two drawbacks: it requires a change in the model on one hand, and the counted objects cannot themselves be configured. Hence the isomorphisms arising from the existence of isomorphic substructures cannot be handled this way.

\(^8\) http://www.ilog.com
[24] applies a notion called "context dependant interchangeability" to configuration. This is more general than the two approaches seen before, but applies to the specific area of case adaptation. Also, since context dependant interchangeability detection is non polynomial, [24] only involves an approximation of the general concept. Furthermore, the underlying formalism, standard CSPs, is known as too restrictive for configuration in general.

8. 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 [9]. We have shown that the isomorphisms stemming from the properties of a sub-problem called the structural problem, can be tackled by using a time pseudo linear algorithm. A search procedure can accept only weakly canonical DAGs as valid configurations, which allows to extend the range of practically tractable problems. Once the canonical configuration structure is known, the interchangeability of a number of nodes can be readily exploited by the remaining search, which involves decisions relative to object classification and attribute variables. The corresponding automorphisms are known as a side effect of canonicity testing, and thus induce no overhead.

References
