Fast Canonical Configuration Generation and Filtering

Laurent Henocque and **Nicolas Prcovic**
LSIS - Universités d'Aix-Marseille II et III
Faculté de St-Jérôme - Avenue Escadrille Normandie-Niemen - 13013 Marseille

Abstract

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

Introduction

Constraint based configuration (Barker *et al.* 1989; Mittal & Falkenhainer 1990; Amilhastre, Fargier, & Marquis 2002; Sabin & Freuder 1996; Soininen *et al.* 2001; Stumptner 1997; Mailharro 1998) challenges symmetry elimination methods known to the constraint solving community by introducing dynamics. Because the size of solutions to configuration problems is potentially infinite, the configuration problem is semi decidable, which calls for specific approaches to tackle the isomorphism problem at the structure level. Then, once the structure of a configuration is known, the remainder of the search amounts to a standard CSP search, where all CSP variables are known , hence to standard CSP symmetry elimination strategies.

The contribution of this paper is a significant enhancement to the work in (Henocque & Prcovic 2004). Canonicity testing can be greatly simplified by exploiting further the properties of the canonicity definition. The main idea amounts to exploiting the fact that when canonical extension occurs, only a limited number of operations are required to test for canonicity, because the extended tree remains to a large extent sorted in a predictive manner. The presented results hence reduce the overhead incurred by canonicity testing further than was expected, and also allow for filtering to take place. We also further extend the range of experiments both by addressing known benchmarks as the "Rack" or "Vellino" problems(Hentenryck *et al.* 1999) in addition to the test problems studied in (Henocque & Prcovic 2004) .

To the best of our knowledge, filtering has never been applied to canonical enumeration.

Plan of the article

Section 1 "Definitions" introduces necessary definitions, concepts and minimal background. Section 2 "Isomorph free structure generation" presents the fundamental assumptions and theorems relevant to the subject. Section 3 "Improved Canonicity Testing" presents a new function for filtering and incrementally testing the canonicity of configuration trees and proves that it is linear in the tree size. Section 4 "Experimental Results" provides experimental results on a range of problems and Section 5 concludes.

Definitions

A *configuration problem* (or *model*) describes a generic product, in the form of declarative statements (rules or axioms) about product well-formedness. A *configuration* is a valid problem instance. As an example[1], we configure a building having at most F floors, each floor owning one to R rooms, each room having one to three doors and at most four windows (illustrated in Figure 1). We ignore here the additional attributes and constraints that exist in most problems to focus on structural constraints alone. Indeed, once a structure has been chosen for a configuration problem, what remains amounts to a classical CSP, to which any CSP symmetry breaking procedures applies. Configurations involve

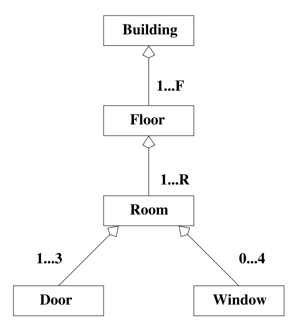

Figure 1: A simplified multi story building model

[1] Later used in the experiments.

interconnected objects, as illustrated in Figure 2, where the existence of structural isomorphisms is obvious.

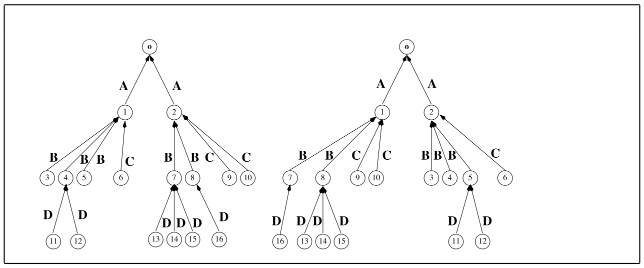

Figure 2: Two isomorphic configuration trees

Structural (sub) problems

From general configuration problems, we isolate subproblems called *structural problems* obtained by abstracting away everything but binary composite relations, the related types and structural constraints. A formalization follows. We consider a totally ordered set O of objects ($O = \{1, 2, \ldots\}$), a totally ordered set T_C of type *symbols* (unary relations) and a totally ordered set R_C of binary relation *symbols* such that:

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

The above condition forbids that an object occurs twice in a structural configuration, which hence is a tree[2]. By $\prec_O$, $\prec_{T_C}$ and $\prec_{R_C}$ we denote 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*.*

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* configuration (solution) *of the structural problem.*

In the general case, a configuration can be represented using a vertex-colored directed acyclic graph (DAG) G=(t,X,E,L) with X $\subset O$, E $\subset O \times O$ and L $\subset O \times T_C$ where t denotes the root type[3], X the vertex set, E the edge set and L is the function which maps each vertex to a type, as illustrated in Figure 2. But thanks to the above assumptions, we only deal with trees here. Because they are trees, configurations can be equivalently represented using vertex colored trees called T-trees (Figure 3). To ease comparisons, we use the same definitions and notations as in (Henocque & Prcovic 2004), some of them being recalled here for self contained-ness.

Definition 3 (T-tree) *A* T-tree *is a finite and non empty tree where nodes are labeled by* T_C*. We note* $(T, \langle c_1, \ldots c_k \rangle)$ *the T-tree with sub-trees* $c_1, \ldots c_k$ *and root label* T*.*

[2]This condition is strong but met by a significant structural kernel of most configuration problems. The choice of R_C is easy.

[3]The root type is the type of objects that occur at the root of the configuration.

Isomorphisms

Testing whether two graphs are isomorphic is an NP problem until today unclassified as either NP-complete or polynomial. For several categories of graphs, like the trees but also graphs having a bounded vertex degree, this isomorphism test is polynomial (Luks 1982). Configuration trees and T-trees being trees, they are isomorphic, equal, superposable, under the same assumptions as standard trees. An *isomorphism class* represents a set of isomorphic graphs. should ideally generate only one *canonical* representative per class. Strong arguments in (Henocque & Prcovic 2004) show that having an efficient canonicity test is not enough to address structural symmetry in configurations. The canonicity definition impacts on the possibility to use canonicity to trigger backtrack within search procedures: each canonical configuration must extend another one by unit extension (defined in the next section). An example of such a search procedure is given in (Henocque, Kleiner, & Prcovic 2005)

Related work in CSP and configuration

Symmetries in classical CSPs are bijections over the set of literals (a literal is a variable ('x') assignment to a value 'v' :"x=v") that preserve solutions (Cohen *et al.* 2006). They naturally involve two subcategories: variable and/or value symmetries. In a classical CSP, all values, variables and constraints are known beforehand, hence the set of symmetries (the automorphism group) is known before the search begins. Of course, some approaches deal with the changes in the automorphism group that occur during search (when several variables are assigned and we face a subproblem), but nothing corresponds to the kind of changes that may arise in configuration.

Symmetry in configuration adds the dimension of deciding whether the choice of adding a component, and also the set of its node attribute variables, must or not be performed in order to avoid generating isomorphic structures. Dealing with structural symmetries significantly differs in nature from its counterpart in variable assignments. In the latter case, we use the automorphism group of the current structure (its "internal" symmetries) to prevent from redundant assignments. In the former case, we use the isomorphism group of the current structure (its "external" symmetries) to prevent from generating redundant structures.

If one accepts to bound the target size of the solutions of a configuration problem, it can obviously be solved using standard CSP techniques after a translation phase. Then of course, dealing with symmetry can be achieved using known techniques from the CSP area (Backofen & Will 1999; Crawford *et al.* 1996; Gent & . 2000; Gent, Harvey, & Kelsey 2002; cois Puget 2006). Using such translations can be extremely and needlessly resource consuming. For instance, if "a building has at most 50 stories with at most 20 rooms and at most 5 windows and 3 doors with a size", we must deal with a CSP having at least 8000 variables, not to mention the growth of the constraint set in the SBDD, SBDS or DLC cases. However, the problem constraints may result in the fact that much less than 8000 windows and doors occur in solutions. This is why configuration problems are usually solved using variations of the CSP formalism. Composite CSP for instance (Sabin & Freuder 1996) tackle dynamicity by dynamically introducing CSP fragments during search. Dynamic CSP or conditional CSP (Soininen *et al.* 2001; Gelle & Faltings 2003) exploit "active" variables and "activation" rules to control how the construction of the configuration structure introduces new elements. The symmetry elimination technique that we are proposing precisely aims at preventing undue extensions of the solution.

Our approach addresses structural symmetry elimination in configuration problems with a specialized algorithm. This warrants that only exactly the required amount of resources is used and allows for extremely low overheads. For instance in this framework, symmetries can be predicted, and not computed. This research significantly improves over results originally presented in (Henocque & Prcovic 2004) then generalized in (Henocque, Kleiner, & Prcovic 2005). The main contribution of this body of work is to exploit the existence of a definition of tree canonicity whereby each canonical tree can be reached by unit extension (e.g. adding one new node) from another canonical tree. Canonicity is defined using a total ordering over labeled trees. This work also followed several earlier contributions to symmetry elimination in configuration problems (as e.g. in (Mailharro 1998) that addressed limited issues: interchangeability of unused objects, use of cardinalities instead of plain objects when they remain interchangeable).

Isomorph-free structure generation

As a means of isolating a canonical representative of each equivalence class of T-trees, we define a total order over T-trees (illustrated Figure 3) . The relation : $\preccurlyeq$ is the total order that generalizes $\prec_{T_C}$ to T-trees.

Definition 4 (The relation $\preccurlyeq$) *Given two T-trees C = (T, L) and C' = (T', L'), $\preccurlyeq$ is recursively defined as follows : C $\preccurlyeq$ C' iff T $\prec_{T_C}$ T' or T = T' and L $\preccurlyeq_{lex}$ L'.*

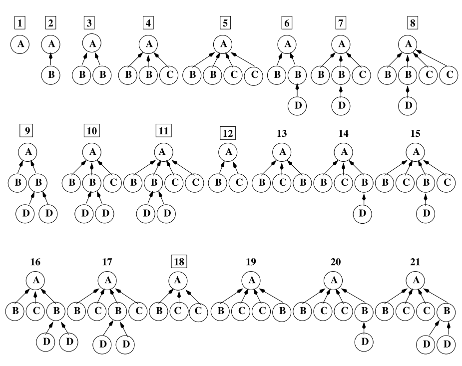

Figure 3: T-trees ordered by $\preccurlyeq$. The index of each $\preccurlyeq$-minimal representatives is framed. At most two D can connect to a B, two B may connect to an A and two C may connect to an A.

We define canonical T-trees recursively as follows (with a testing algorithm in view):

Definition 5 (Canonicity of a T-tree) *A T-tree C = (T, L) is* canonical *iff L is empty or if L is $\preccurlyeq$ -sorted and each c in L is itself canonical.*

It can be shown that, as defined, canonical T-trees are the $\preccurlyeq$-minimal representatives of their isomorphism class. The elementary operation used to generate T-trees is called a unit extension.

Definition 6 (Unit Extension, Canonical Unit Extension) *We call* unit extension*, the operation of adding a single terminal node in a T-tree C. If additionally both C and the result are canonical, the operation is called* canonical unit extension*.*

The goal of a constructive search procedure is to produce T-trees starting from $(t, \langle\rangle)$ (recall that t is the type of the root object in the configuration) which respect all the problem constraints (i.e. not only the constraints involved in the structural problem) using unit extension. Eliminating isomorphisms requires to generate only canonical solutions. The chosen definition of canonicity ensures that each canonical T-tree can be reached by canonical unit extension, which makes canonicity testing a central issue.

Proposition 1 *Let C be a T-tree. Let C' be a T-tree resulting from unit extension on C. We have $C \preccurlyeq C'$.*

Proof 1 *The proof is by induction on T-trees. The proposition is true for a T-tree with no sub-node. Looking at the recursion in Function "CompareT-Trees" in Figure 4, we see that if the unit extension was performed on a sub-tree before index k, by the induction hypothesis, the function will return* GREATER*, as well as if the unit extension was obtained by adding the new node at position $k + 1$.*

We henceforth know according to Proposition 1 that adding a node may never yield a tree that would be $\preccurlyeq$-less than its predecessor. When incrementally checking for canonicity, this allows to only compare each modified sub-tree with its successor sibling, not with its predecessor.

function **CompareT-Trees**(C, C')
in : $C = (T, L)$ and $C' = (T', L')$,
out : EQUAL if $C = C'$,
LESS if $C \neq C'$ and $C \preccurlyeq C'$,
GREATER in the other cases,

if $T < T'$ **then return** LESS
if $T' < T$ **then return** GREATER
if $L = \langle\rangle$ **and** $L' = \langle\rangle$ **then return** EQUAL
if $L = \langle\rangle$ **then return** LESS
if $L' = \langle\rangle$ **then return** GREATER
let $L = \langle a_1, \ldots, a_k \rangle$,
let $L' = \langle b_1, \ldots, b_l \rangle$,
for $i := 1$ **to** k **do**
  if $l < i$ **then return** GREATER,
  $result$:=CompareT-Trees(a_i, b_i),
  if $result \neq equal$ **then**
  return $result$,
return EQUAL,

Figure 4: The function **CompareT-Trees**

Improved canonicity testing

The simplest algorithm is pseudo linear (O(n log n)) in the tree size, and straightforwardly exploits the definition of canonicity using a function for comparing T-tree recalled for clarity in Figure 4. Basically, the algorithm tests that T-trees are internally recursively sorted according to the definition. Each sub-tree at any level is compared to its right neighbor (if any) to test that it is $\preccurlyeq$-lower. We can improve this algorithm by exploiting Proposition 1 for filtering and the fact that T-trees are incrementally built.

Filtering

Proposition 1 naturally yields a filtering algorithm. When considering all the positions open for unit extension wrt. the relation cardinalities we see that since the T-trees generated are canonical, hence internally lexicographically sorted, it is not possible to perform any unit extension inside a sub-tree being equal to its successor (since it becomes greater, and the whole T-tree non canonical). A new search procedure can take advantage of this by excluding these choices from search.

It suffices to adapt the function `compareT-Trees` so that each time it compares two trees, it memorizes whether there is strict inequality or not. This is performed in constant time. Later, the same information can be read again in constant time, allowing an efficient usage of Proposition 1 . In the case the enumeration procedure would be connected to a standard CSP system to compute configurations involving attribute variables or other relations, this filtering allows to close the port variables representing the relations, hence resulting in further propagations.

Incremental Canonicity

Now, when we test the canonicity of a newly generated T-tree we know that only one node and edge were added, to a tree that originally was canonical (a condition for continuation). The Function in Figure 5 details the algorithm that can assess the canonicity of the T-tree obtained from unit extension of a previously canonical T-tree. Only the valid placement of the subtrees rooted at an ancestor of the newly inserted node must be tested. This must be done starting from the inserted node up to the root. The procedure starts from the newly inserted node, climbing up the T-tree, and performs comparisons at each level.

function **IncrementalCanonical**(C, N)
in : a T-tree $C = (T, L)$, N a newly inserted node in C
out : TRUE if C is canonical, FALSE if not,

for $(n = N \,;\, n \neq C \,;\, n = parent(n))$ {
  if (**CompareT-Trees(**$n, right(n)$**)**==GREATER)
    return FALSE
}
return TRUE,

Figure 5: Incremental canonicity testing

Proposition 2 *Let C be a canonical T-tree. Let C' be a T-tree resulting from a unit extension on C. Testing the canonicity of C' has a cost linear in the size of C'*

Proof 2 *In the worst case of a perfectly balanced binary T-tree of size S and depth d where all pairwise subtrees are equal, hence require to be entirely scanned by calls to compareT-Trees, the cost of canonicity testing is $2\Sigma_{i=0}^{d}(S/2^i)$, which converges towards $4S$ when d increases.*

Experimental Results

We have performed tests to compare three isomorphism elimination methods : one that basically tests canonicity (“no iso”) and the ones introduced here: improved incremental testing (”no iso fast”) and the same with filtering (”no iso fast + filtering”). We give the result times (obtained by a Java program running on a Linux 2.4 Ghz PC), the number of generated T-trees, and the number of calls to the comparison function.

Our first experiments were realized on the floor planning problem illustrated in figure 1. In order to explore the combinatorial properties of the problem, we let vary the following parameters: F (counting the max number of floors in a building) and R (the max count of rooms in a floor). We have written a configurator in Java which generates all the possible solutions of the floor planning problem, according to the parameters F, R.

We do not list any execution result concerning the enumeration of all (non canonical inclusive) solutions, because they are too high. For instance, with $F = 1$ and $R = 3$, it takes 712ms, and with $F = 1$ and $R = 4$ it takes 40s.

We observe in Table 1 that improving the canonicity test reduces by more than one order of magnitude the number of calls to `compareT-Trees` and to significantly reduce execution times. Filtering reduces the number of T-trees considered and further improves the execution times. The best results occur with problems of big size, for which the cumulated impact of both methods divides execution times by more than 2.

We have also resolved two classical configuration problems: the “Rack” problem (problem 031 in CSPLib [4]) and the “Vellino” problem (Hentenryck *et al.* 1999). We added to our structure generation algorithms the constraint tests implied by these problems, which are not directly linked to the structure of the solutions. For the racks: limited number of available racks, power a rack can supply is greater than the sum of powers its connected cards require, etc. For the Vellino’s problem: components compatibility constraints, maximum number of components contained by each type of bin, etc. We only implemented a straightforward approach for those constraints, by testing them after each structure unit extension. Implementing usual filtering techniques (Forward Checking (FC),MAC) could help reducing solving times (with or without isomorphism removal). However, we were only interested in comparing between isomorph aware and classic algorithms. Adding domain filtering would not alter the comparison as those two structures

[4]http://www.csplib.org/prob/prob031

(F, R)	#sol	no iso #trees	no iso #calls	no iso time	no iso fast #calls	no iso fast time	no iso fast + filtering #trees	no iso fast + filtering #calls	no iso fast + filtering time
(1, 6)	54 10^3	111 10^3	2.5 10^6	0.54s	233 10^3	0.43s	84 10^3	180 10^3	0.34s
(1, 7)	170 10^3	358 10^3	9.8 10^6	1.9s	721 10^3	1.5s	262 10^3	541 10^3	1.1s
(1, 8)	490 10^3	1.0 10^6	33 10^6	5.6s	2.0 10^6	4.3s	746 10^3	1.5 10^6	3.2s
(1, 9)	1.3 10^6	2.8 10^6	105 10^6	16s	5.3 10^6	11.6s	2.0 10^6	3.8 10^6	8.2s
(1, 10)	3.2 10^6	7.2 10^6	303 10^6	42s	13 10^6	29s	4.9 10^6	9.3 10^6	20s
(2,3)	334 10^3	645 10^3	15 10^6	3.2s	3.4 10^6	2.7s	530 10^3	3.1 10^6	2.2s
(2, 4)	7.5 10^6	15 10^6	511 10^6	85s	79 10^6	63s	12 10^6	72 10^6	51s
(2, 5)	120 10^6	245 10^6	2.0 10^9	1459s	1.3 10^9	1071s	187 10^6	1.2 10^9	835s

Table 1: number of solutions, T-trees, calls to `compareT-trees` and time obtained when varying the values of F and R with and without incrementality in the canonicity test and with and without filtering.

elimination mechanisms are orthogonal: if the filtering prevents a non canonical structure, it also prevents its canonical counterpart. Each isomorphism class is either completely eliminated by filtering, or left as a possible solution candidate. The gain factor we have obtained should therefore be equivalent with the addition of filtering techniques.

To the best of our knowledge, the most recent experimental results on the rack problem are listed in (Kiziltan & Hnich 2001). The authors connect two dual models of the problem within a classical CSP approach using channelling constraints. They do not solve the problem 4. We have not found any solution of the instance 4 elsewhere. Here it is : cost = 1150 with one R250 containing one C150 and one C100, one R300 containing one C100, two C75 and one C50, one R300 containing three C50, three C40 and one C20, and one R300 containing three C40 and eight C20.

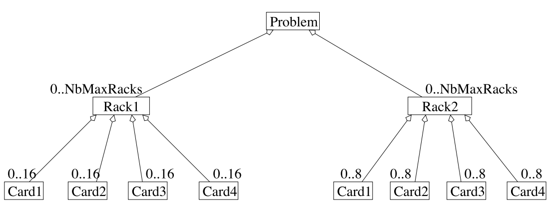

Figure 6: Rack problem component hierarchy (the arrows represent composition relations). A problem instance involves up to `NbMaxRacks` racks of types 1 or 2 needed to connect cards of differents types. Racks of type 1 can be connected up to 16 cards, racks of type 2 up to 8 cards.

Conclusion

This work exploits the incremental nature of canonical configuration generation to both introduce filtering and obtain a significant complexity improvement of the whole procedure. We see that the experimental results here performed on problems having a limited size already yield significant speedups, and that the gain grows with tree size and depths.

Additionally, configuration tree generation is meant to be coupled to a standard configuration search engine, or to a constraint solver. In that case, filtering allows to exploit the fact that the possibility of further connecting objects inside a structure is closed. This can result in added constraint propagation in the rest of the problem. This is ongoing research.

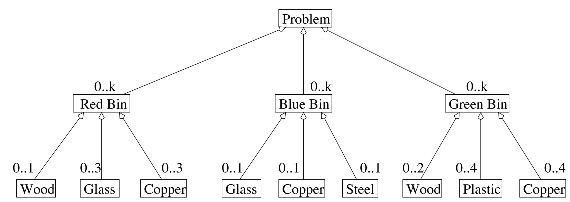

Figure 7: Vellino's problem component hierarchy. k components must be stored in the bins so we know that we need at most k bins all, and at most k of each kind. The object model accounts for some of the constraints : Red bins can't contain plastic or steel, Blue bins can't contain wood or plastic, Green bins can't contain steel or glass. Red bins can contain up to 3 components but at most 1 wooden one. Blue bins can only contain 1 component. Green bins can contain up to 4 components but at most 2 wooden ones.

References

Amilhastre, J.; Fargier, H.; and Marquis, P. 2002. Consistency restoration and explanations in dynamic csps—-application to configuration. *Artificial Intelligence* 135(1-2):199–234.

Backofen, R., and Will, S. 1999. Excluding symmetries in constraint-based search. In *Principles and Practice of Constraint Programming*, 73–87.

Barker, V.; O'Connor, D.; Bachant, J.; and Soloway, E. 1989. Expert systems for configuration at digital: Xcon and beyond. *Communications of the ACM* 32:298–318.

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

cois Puget, J. F. 2006. Dynamic lex constraints. In *In Benhamou, F., ed., Principles and Practice of. Constraint Programming - CP 2006, LNCS 4204*, 453?467.

Crawford, J.; Ginsberg, M. L.; Luck, E.; and Roy, A. 1996. Symmetry-breaking predicates for search problems. In Aiello, L. C.; Doyle, J.; and Shapiro, S., eds., *KR'96: Principles of Knowledge Representation and Reasoning*, 148–159. San Francisco, California: Morgan Kaufmann.

Gelle, E., and Faltings, B. 2003. Solving mixed and condi-

Instance	all		no iso			no iso fast		no iso fast + filtering		
	#trees	time	#trees	#calls	time	#calls	time	#trees	#calls	time
inst1	2424	55ms	1647	6849	51ms	3280	51 ms	1645	3263	51ms
inst2	237 10^6	1732s	4.9 10^3	71 10^3	36s	12.2 10^6	33s	4.8 10^6	11.9 10^6	33s
inst3	683	66ms	683	10 10^3	66ms	1.4 10^3	63ms	683	1.4 10^3	63ms
inst4	821 10^3	5.2 s	237 10^3	1.8 10^6	1.8s	526 10^3	1.6s	237 10^3	525 10^3	1.6s

Table 2: The four instances of the Rack Problem (CSPLib 031).

Instances					all		no iso			no iso fast		no iso fast + filtering		
G	P	S	W	C	#trees	time	#trees	#calls	time	#calls	time	#trees	#calls	time
1	2	1	3	2	32 10^3	462ms	14 10^3	74 10^3	179ms	60 10^3	174ms	12 10^3	55 10^3	166ms
2	2	1	3	2	105 10^3	1.4s	40 10^3	446 10^3	442ms	171 10^3	429ms	33 10^3	156 10^3	403ms
2	4	1	3	2	195 10^3	3.1s	65 10^3	458 10^3	815ms	292 10^3	787ms	55 10^3	266 10^3	732ms
2	4	3	3	2	6.9 10^6	108s	715 10^3	6.1 10^6	7.3s	2.5 10^6	6.7s	517 10^3	2.2 10^6	6.0s
2	4	3	6	2	119 10^6	1958s	4.0 10^6	48 10^6	38s	15 10^6	33s	2.8 10^6	12 10^6	28s
2	4	3	6	4	533 10^6	8615s	13 10^6	177 10^6	151s	48 10^6	124s	9.8 10^6	41 10^6	111s

Table 3: Vellino's Problem results. The first five columns show the demand of Glass, Plastic, Steel, Wood and Copper.

tional constraint satisfaction problems. *Constraints* 8:107–141.

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

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

Henocque, L., and Prcovic, N. 2004. Practically handling configuration automorphisms. In *proceedings of the 16th IEEE International Conference on Tools for Artificial Intelligence*.

Henocque, L.; Kleiner, M.; and Prcovic, N. 2005. Advances in polytime isomorph elimination for configuration. In *proceedings of Principles and Practice of Constraint Programming - CP 2005,*, 301–313. Sitges Barcelona, Spain: Springer.

Hentenryck, P. V.; Michel, L.; Perron, L.; and Rgin, J.-C. 1999. Constraint programming in opl. In *Principles and Practice of Declarative Programming*.

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

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

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

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, 25–32.

Sabin, D., and Freuder, E. C. 1996. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, 153–161.

Soininen, T.; Niemela, I.; Tiihonen, J.; and Sulonen, R. 2001. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming: Towards Efficient and Scalable Knowledge*, 195–201.

Stumptner, M. 1997. An overview of knowledge-based configuration. *AI Communications* 10(2):111–125.