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; Shtompner 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

1Later 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 relations such that:

\[
\forall R_1, R_2 \in R_C, \forall o_1, o_2, o_3 \in O, R_1(o_1, o_2) \Rightarrow R_2(o_1, o_3)
\]

The above condition forbids that an object occurs twice in a structural configuration, which hence is a tree\(^2\). By \(<_O, \prec_T, \prec_R \), 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 \( X \subseteq O \), \( E \subseteq O \times O \) and \( L \subseteq O \times T_C \) 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 (Soisinen 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 is the total order that generalizes to T-trees.

**Definition 4 (The relation)** Given two T-trees and , is recursively defined as follows: if or and and

\[
C \preceq C' \iff T \preceq T' \text{ or } T = T' \text{ and } \text{L is } \preceq \text{lex } L'.
\]

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

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

**Definition 5 (Canonicity of a T-tree)** A T-tree is canonical iff is empty or if is -sorted and each in .

It can be shown that, as defined, canonical T-trees are the 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 . If additionally both 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 (recall that 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 be a T-tree. Let result from unit extension on . We have .

**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 , 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 .

We henceforth know according to Proposition 1 that adding a node may never yield a tree that would be -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.

```plaintext
function CompareT-Trees(C, C')
in : C = (T, L) and C' = (T', L'),
out : EQUAL if C = C',
LESS if C != C' and C \preceq\ C',
GREATER in the other cases,
if T < T' then return LESS
if T' < T then return GREATER
if L = () and L' = () then return EQUAL
if L = () then return LESS
if L' = () then return GREATER
let L = (a_1, \ldots, a_k),
let L' = (b_1, \ldots, b_l),
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 \( \leq \)-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 newly 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.

![Figure 5: Incremental canonicity testing](image)

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 \( 2^{S+d-1} \cdot (S/2^d) \), 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 cumulative 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) and the “Vellino” problem (Hentenryc 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 are canonical, hence internally lexicographically sorted, it is always 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. 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 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 \( 2^{S+d-1} \cdot (S/2^d) \), which converges towards 4S when \( d \) increases.
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 different types.](image)

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.

<table>
<thead>
<tr>
<th>(F, R)</th>
<th>#sol</th>
<th>#trees</th>
<th>no iso</th>
<th>#calls</th>
<th>time</th>
<th>no iso fast</th>
<th>#calls</th>
<th>time</th>
<th>no iso fast + filtering</th>
<th>#calls</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 6)</td>
<td>54*10^7</td>
<td>111*10^7</td>
<td>2.5*10^7</td>
<td>5.4s</td>
<td>2.33*10^4</td>
<td>0.43s</td>
<td>84*10^4</td>
<td>180*10^4</td>
<td>0.34s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 7)</td>
<td>170*10^8</td>
<td>358*10^7</td>
<td>9.8*10^7</td>
<td>1.9s</td>
<td>721*10^4</td>
<td>1.5s</td>
<td>262*10^4</td>
<td>541*10^4</td>
<td>1.1s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 8)</td>
<td>490*10^8</td>
<td>1.0*10^9</td>
<td>33*10^7</td>
<td>5.6s</td>
<td>2.0*10^6</td>
<td>4.3s</td>
<td>746*10^4</td>
<td>1.5*10^7</td>
<td>3.2s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 9)</td>
<td>1.3*10^9</td>
<td>2.8*10^9</td>
<td>105*10^6</td>
<td>16s</td>
<td>5.3*10^6</td>
<td>11.6s</td>
<td>2.0*10^6</td>
<td>3.8*10^6</td>
<td>8.2s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1, 10)</td>
<td>3.2*10^9</td>
<td>7.2*10^9</td>
<td>303*10^6</td>
<td>42s</td>
<td>13*10^9</td>
<td>29s</td>
<td>4.9*10^9</td>
<td>9.3*10^9</td>
<td>20s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 3)</td>
<td>334*10^9</td>
<td>645*10^7</td>
<td>15*10^7</td>
<td>3.2s</td>
<td>3.4*10^9</td>
<td>2.7s</td>
<td>330*10^7</td>
<td>3.1*10^9</td>
<td>2.2s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 4)</td>
<td>7.5*10^9</td>
<td>15*10^9</td>
<td>511*10^6</td>
<td>85s</td>
<td>79*10^9</td>
<td>63s</td>
<td>12*10^9</td>
<td>72*10^9</td>
<td>51s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2, 5)</td>
<td>120*10^9</td>
<td>245*10^9</td>
<td>2.0*10^9</td>
<td>1459s</td>
<td>1.3*10^9</td>
<td>1071s</td>
<td>187*10^6</td>
<td>1.2*10^7</td>
<td>8.35s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 7: Vellino’s problem component hierarchy.](image)

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.

References


Gelle, E., and Faltings, B. 2003. Solving mixed and condi-
Table 2: The four instances of the Rack Problem (CSPLib 031).

<table>
<thead>
<tr>
<th>G</th>
<th>P</th>
<th>S</th>
<th>W</th>
<th>C</th>
<th>Instances</th>
<th>#trees</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>inst1</td>
<td>2424</td>
<td>55ms</td>
<td>1647</td>
<td>6849</td>
<td>51ms</td>
<td>3280</td>
<td>51ms</td>
<td>1645</td>
<td>3263</td>
<td>51ms</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>inst2</td>
<td>827</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>inst3</td>
<td>683</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>inst4</td>
<td>821</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>437</td>
<td>2187</td>
<td>0.6s</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instances</th>
<th>all</th>
<th>G</th>
<th>P</th>
<th>S</th>
<th>W</th>
<th>C</th>
<th>#trees</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
<th>#trees</th>
<th>#calls</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td>321</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td>105</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td>195</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td>6.9</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
<td>0.6s</td>
<td>1647</td>
<td>0.1s</td>
<td>4825</td>
</tr>
</tbody>
</table>


