Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs

Tommi Junttila* Petteri Kaski†

Abstract

The problem of canonically labeling a graph is studied. Within the general framework of backtracking algorithms based on individualization and refinement, data structures, subroutines, and pruning heuristics especially for fast handling of large and sparse graphs are developed. Experiments indicate that the algorithm implementation in most cases clearly outperforms existing state-of-the-art tools.

1 Introduction.

1.1 Background and Motivation. Consider the task of querying a large database of chemical compounds for a compound described in terms of constituent atoms and bonds. To describe the compound, we must be able to name the atoms that bond with each other, and this in general requires associating a unique label to each constituent atom. The compound itself, however, is independent of any labeling. In particular, we are faced with the difficulty that the labeling used to describe the compound in the database may be different from the labeling we use to specify the query. This example serves to illustrate a recurrent problem in combinatorial computation. Given a labeled combinatorial object as input, the task is to label the object in a manner that is independent of the input labeling. Such a labeling is called a *canonical labeling* of the input object.

Because essentially any explicitly given combinatorial object can be concisely represented as a graph for purposes of canonical labeling, in practice the fundamental problem is to efficiently canonically label a given graph. Unfortunately, no polynomial-time algorithm is known for this task. Indeed, a polynomial-time canonical labeling algorithm immediately implies a polynomial-time algorithm for the *graph isomorphism problem*—the problem of deciding whether two given graphs are the same up to a change of labels—which from a theoretical perspective ranks among the most extensively studied computational problems [2, 3, 4, 18, 19, 24, 36, 39]. To briefly summarize current theoretical knowledge, while no polynomial-time algorithm is known in the context of arbitrary graphs, polynomial-time algorithms are known for numerous restricted families of graphs, including [5, 16, 28, 34], and there is considerable theoretical evidence, including [7, 23, 30], that the graph isomorphism problem is not NP-complete; cf. [27, 29, 38]. Currently the best asymptotic running time bound for canonical labeling of graphs remains quasi-exponential in the number of vertices [4, 6].

The unresolved theoretical status withstanding, canonical labeling techniques are required in practice in a wide range of disciplines from chemistry to combinatorics and computer science [1, 10, 17, 21, 22]. In particular, there is a demand for practical software tools that perform well on graphs encountered in applications, even if the worst-case performance can be witnessed to scale exponentially on crafted instances. In this respect the current situation bears a resemblance to the substantial algorithm engineering effort in the last few years aimed at practical algorithms for the propositional satisfiability (SAT) problem [33, 43].

1.2 Contribution and Earlier Work. In this paper we take an algorithm engineering approach to canonical labeling, and develop a backtrack algorithm and data structures appropriate for fast handling of large and sparse graphs. To relate the present contribution to earlier research, the present algorithm belongs to the family of backtrack algorithms based on the technique of partition refinement alternated with individualization of vertices; see, for example, [25, 26, 31]. The present algorithm is most closely related to the algorithm in the celebrated software tool *nauty* [31, 44]. Also related is the tool *saucy* [14, 45], which is especially designed for large and sparse graphs. These two tools constitute the state of the art from the perspective of practical algorithms.

*Laboratory for Theoretical Computer Science, Helsinki University of Technology, P.O.Box 5400, FI-02015 TKK, Finland.
e-mail: Tommi.Junttila@tkk.fi

†Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, P.O.Box 68, FI-00014 University of Helsinki, Finland.
e-mail: Petteri.Kaski@cs.helsinki.fi

Comparing with *nauty*, the main contribution in the present work lies in the design of data structures and subroutines accommodating large graphs and facilitating fast searching. To accommodate large and sparse graphs, the total space usage of the algorithm must be linear in the number of edges in the input graph; this in particular precludes the incidence matrix representation used by *nauty* for the input graph. Storing the input graph in a sparse format is essentially just the beginning, however. For graphs with thousands of vertices, careful attention needs to be paid to the design of basic data structures for ordered partitions and the subroutines that manipulate them. Compared with *nauty*, the present algorithm in many cases avoids a linear time overhead at a search tree node by additional bookkeeping and by accessing only essential parts of the ordered partition and the graph at hand. In addition, the present algorithm design improves the heuristics for eliminating redundancy and facilitates early pruning by means of an incremental leaf certificate.

Comparing *saucy* with both *nauty* and the present algorithm design, there are two essential differences. First, *saucy* does not compute a canonical labeling for the input graph, but only a set of generators for its automorphism group. Second, *saucy* apparently implements a less effective set of heuristics for structuring and pruning the search tree. This design choice pays off as less overhead in situations where the input graph lacks regularity, but—as can be observed in the subsequent experiments—significantly degrades performance on many instances of combinatorial origin.

In more detail, the central contributions over *nauty* and *saucy* are as follows. (a) We develop a novel linear-space data structure for chains of ordered partitions that enables local and at most linear-time solutions to basic tasks such as splitting a cell, color degree computation, and undoing any number of cell splits. (b) We introduce a technique for computing leaf certificates incrementally when descending from the root node to a leaf node. This both amortizes work from the leaves towards the root node and enables early pruning of the search tree compared with existing algorithms that evaluate leaf certificates only at the leaf nodes; a substantial performance gain is obtained by being able to abort an emerging node in certain situations before the partition refinement process completes. (c) We strengthen existing heuristics for pruning redundant subtrees by keeping track of an additional automorphism orbit partition; this achieves additional pruning in certain situations with negligible overhead.

Experiments carried out on an extensive catalogue of benchmark graphs indicate that the present tool in most cases clearly outperforms *nauty* and *saucy* on large and sparse graphs, and exhibits comparable or better performance also on dense and highly regular graphs of combinatorial origin, such as graphs constructed from various finite geometries and Hadamard matrices. Exponential scaling can nevertheless be observed for all tools on specifically crafted families of graphs.

The open-source algorithm implementation and the catalogue of graphs compiled for the experiments are available at [46].

1.3 Organization of the Paper. In §2 we review the basic definitions and notational conventions used throughout this paper. In §3 we present the general individualization and refinement scheme for canonical labeling algorithms; the subsequent sections proceed to describe the implementation of the scheme in the case of the present algorithm. In §4 we describe and justify the techniques employed for detecting automorphisms and pruning redudant subtrees. In §5 we describe the data structures and subroutines for chains of ordered partitions. In §6 we describe the incremental leaf certificate together with an additional pruning technique. In §7 we present the experiments comparing the present algorithm implementation with *nauty* and *saucy*. In §8 we conclude the paper with a brief discussion.

2 Preliminaries.

We assume familiarity with graphs, permutation groups, and group actions [8, 15].

A *graph* is an ordered pair $G = (V, E)$, where V is a finite set and E is a set of 2-element subsets of V. The elements of V are called *vertices* and the elements of E *edges*. We write $\mathcal{G}(V)$ for the set of all graphs with vertex set V. Throughout this paper we assume that $V = \{1, 2, \ldots, n\}$.

We denote by $\mathrm{Sym}(V)$ the group of all permutations of V. The image of $x \in V$ under $\gamma \in \mathrm{Sym}(V)$ is denoted by x^γ. The composition of permutations $\gamma_1, \gamma_2 \in \mathrm{Sym}(V)$ is defined for all $x \in V$ by $x^{(\gamma_1\gamma_2)} = (x^{\gamma_1})^{\gamma_2}$. For example, in cycle notation, $(1\ 2)(2\ 3) = (1\ 3\ 2)$. A permutation acts on a subset $W \subseteq V$ by $W^\gamma = \{x^\gamma : x \in W\}$ and on a graph G by $G^\gamma = (V^\gamma, E^\gamma)$, $V^\gamma = V$, and $E^\gamma = \{\{x^\gamma, y^\gamma\} : \{x, y\} \in E\}$.

A *partition* of V is a set of nonempty pairwise disjoint subsets of V whose union is V. An *ordered partition* of V is a list $\pi = (W_1, W_2, \ldots, W_m)$ such that the set $\{W_1, W_2, \ldots, W_m\}$ is a partition of V. We write $\Pi(V)$ for the set of all ordered partitions of V. Each set W_i is called a *cell* of the partition. A partition is *discrete* if all its cells are singleton sets and *unit* if it has only one cell (the set V).

An ordered partition π associates with each $x \in V$ the index $\pi(x)$ of the cell of π in which x occurs, that is,

$\pi(x) = i$ if and only if $x \in W_i$. If π is discrete, the mapping $\bar{\pi} : x \mapsto \pi(x)$ is a permutation of V. Conversely, a permutation $\gamma \in \mathrm{Sym}(V)$ corresponds to the discrete ordered partition $(\{1^{\gamma^{-1}}\}, \{2^{\gamma^{-1}}\}, \ldots, \{n^{\gamma^{-1}}\})$. We identify discrete ordered partitions with permutations in this manner. For example, if $\pi = (\{3\}, \{1\}, \{2\})$, then the corresponding permutation is $\bar{\pi} = (1\ 2\ 3)$. A permutation $\gamma \in \mathrm{Sym}(V)$ acts on an ordered partition $\pi = (W_1, W_2, \ldots, W_m)$ by $\pi^\gamma = (W_1^\gamma, W_2^\gamma, \ldots, W_m^\gamma)$. In particular, if π is discrete, $\overline{\pi^\gamma} = \gamma^{-1}\bar{\pi}$.

For ordered partitions $\pi_1, \pi_2 \in \Pi(V)$, we say that π_1 is *at least as fine as* π_2 and write $\pi_1 \preceq \pi_2$ if π_2 can be obtained from π_1 by replacing zero or more times two consecutive cells with the union of the cells. Equivalently, $\pi_1 \preceq \pi_2$ if and only if for all $x, y \in V$ it holds that $\pi_1(x) \leq \pi_1(y)$ implies $\pi_2(x) \leq \pi_2(y)$. If $\pi_1 \preceq \pi_2$ and $\pi_1 \neq \pi_2$, we say that π_1 is *finer than* π_2 and write $\pi_1 \prec \pi_2$. The relation $\prec$ is a partial order on $\Pi(V)$ whose minimum elements are the discrete ordered partitions and the unique maximum element is the unit ordered partition.

Two graphs G_1, G_2 are *isomorphic* if there exists a permutation $\gamma \in \mathrm{Sym}(V)$ such that $G_1^\gamma = G_2$. Such a permutation γ is called an *isomorphism* of G_1 onto G_2. We write $G_1 \cong G_2$ to indicate that G_1 and G_2 are isomorphic. An isomorphism of a graph onto itself is an *automorphism*. The *automorphism group* $\mathrm{Aut}(G)$ of a graph G consists of all automorphisms of G with composition as the group operation. We extend these notions of isomorphism and automorphism to ordered tuples of objects on which $\mathrm{Sym}(V)$ acts elementwise. For example, for $G_1, G_2 \in \mathcal{G}(V)$ and $\pi_1, \pi_2 \in \Pi(V)$, we have $(G_1, \pi_1) \cong (G_2, \pi_2)$ if and only if there exists a permutation $\gamma \in \mathrm{Sym}(V)$ with $G_1^\gamma = G_2$ and $\pi_1^\gamma = \pi_2$.

Let Ω be a finite set on which $\mathrm{Sym}(V)$ acts. A function I of Ω is an (*isomorphism*) *invariant* on Ω if for all $X, Y \in \Omega$ it holds that $X \cong Y$ implies $I(X) = I(Y)$. An invariant is a *certificate* if for all $X, Y \in \Omega$ it holds that $I(X) = I(Y)$ implies $X \cong Y$.

Let Σ be an ordered set. Let $a = (a_1, a_2, \ldots, a_k)$ and $b = (b_1, b_2, \ldots, b_\ell)$ be two ordered tuples consisting of elements in Σ. The *lexicographic order* for such tuples is defined by setting $a < b$ if and only if either (i) $k < \ell$ and $a_i = b_i$ for all $1 \leq i \leq k$ or (ii) there exists a $1 \leq j \leq \min(k, \ell)$ such that $a_j < b_j$ and $a_i = b_i$ for all $1 \leq i \leq j - 1$. We say that a is *prefix* of b if either $a = b$ or (i) holds. In particular, lexicographically $a \leq b$ whenever a is a prefix of b.

3 Search Trees and Certificates.

The present algorithm is best described by means of an associated backtrack search tree, which is common to essentially all algorithms relying on the individualization and refinement scheme [25, 26, 31]. Differences in algorithms start to appear when one considers how the data structures and subroutines are implemented and how the basic search tree is traversed and pruned using information discovered during the traversal. These implementation aspects for the present algorithm are discussed in more detail in subsequent sections.

It is technically convenient to work with *colored graphs*, that is, ordered pairs $(G, \pi) \in \mathcal{G}(V) \times \Pi(V)$, the intuition being that π associates a "color" $\pi(x)$ with every $x \in V$. A graph corresponds to a colored graph where π is the unit ordered partition.

A *partition refiner* is a function R that associates with every colored graph (G, π) an ordered partition $R(G, \pi)$ such that (i) $R(G, \pi) \preceq \pi$ and (ii) $R(G^\gamma, \pi^\gamma) = R(G, \pi)^\gamma$ for all $\gamma \in \mathrm{Sym}(V)$.

A *cell selector* is a function S that associates with every colored graph (G, π) with non-discrete π a subset $S(G, \pi) \subseteq V$ such that (i) $S(G, \pi)$ is a non-singleton cell of π and (ii) $S(G^\gamma, \pi^\gamma) = S(G, \pi)^\gamma$ for all $\gamma \in \mathrm{Sym}(V)$.

3.1 Search Trees.

Assuming a partition refiner R and a cell selector S, the *search tree* $\mathcal{T}(G, \pi)$ associated with a colored graph (G, π) is an edge-labeled rooted tree inductively defined by the following two rules:

1. If $R(G, \pi)$ is discrete, then the tree $\mathcal{T}(G, \pi)$ consists of the single leaf node $R(G, \pi)$.
2. Otherwise, assume that
$$\nu = R(G, \pi) = (W_1, W_2, \ldots, W_m)$$
and that
$$S(G, \nu) = W_i = \{x_1, x_2, \ldots, x_k\}.$$
The tree $\mathcal{T}(G, \pi)$ consists of the root node ν that has as its children the trees $\mathcal{T}(G, \pi_j)$, $1 \leq j \leq k$, where
$$\pi_j = (W_1, \ldots, W_{i-1}, \{x_j\}, W_i \backslash \{x_j\}, W_{i+1}, \ldots, W_n).$$
The edge from the root node ν to the root node ν_j of the child tree $\mathcal{T}(G, \pi_j)$ is labeled with the vertex x_j. The node ν_j is called the *x_j-child* of ν, denoted by $\nu \xrightarrow{x_j} \nu_j$.

Obviously, for each path
$$\mu_1 \xrightarrow{x_1} \mu_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{d-1}} \mu_d$$
in a search tree $\mathcal{T}(G, \pi)$ it holds that
$$\mu_1 \succ \mu_2 \succ \cdots \succ \mu_d.$$
Furthermore, the nodes in $\mathcal{T}(G, \pi)$ are distinct ordered partitions; each leaf node is a discrete ordered partition.

THEOREM 3.1. *For all $\gamma \in \mathrm{Sym}(V)$, a node ν is the x-child of the node μ in the search tree $\mathcal{T}(G,\pi)$ if and only if the node ν^γ is the x^γ-child of the node μ^γ in the search tree $\mathcal{T}(G^\gamma,\pi^\gamma)$.*

Proof. Starting from the root nodes, by induction on the levels of the trees $\mathcal{T}(G,\pi)$ and $\mathcal{T}(G^\gamma,\pi^\gamma)$.

It follows that we can define an action of $\mathrm{Sym}(V)$ on the set of all search trees associated with colored graphs by setting $\mathcal{T}(G,\pi)^\gamma = \mathcal{T}(G^\gamma,\pi^\gamma)$ for all $(G,\pi) \in \mathcal{G}(V) \times \Pi(V)$ and $\gamma \in \mathrm{Sym}(V)$. Equivalently, $\mathcal{T}(G,\pi)^\gamma$ is obtained by acting on the nodes, edges, and edge labels of $\mathcal{T}(G,\pi)$ with γ. This group action induces a notion of isomorphism for search trees.

A fundamental observation is now that every isomorphism of colored graphs is also an isomorphism of the associated search trees. In particular, any automorphism of a colored graph maps an associated search tree onto itself. Because only the identity permutation fixes a discrete ordered partition, it follows that the automorphism group of a colored graph acts semiregularly on the leaf nodes of an associated search tree. This observation on one hand enables the discovery of automorphisms while traversing the search tree, and, on the other hand, shows that it is not feasible to traverse the entire tree for graphs with a large automorphism group—indeed, by the orbit-stabilizer theorem, the number of leaf nodes in $\mathcal{T}(G,\pi)$ is a positive multiple of $|\mathrm{Aut}(G,\pi)|$. Thus, a search tree can be very large; as an extreme case, consider the complete graph K_n.

3.2 Leaf Certificates and Canonical Labeling. In practice the entire search tree is not traversed when computing a canonical labeling, however. To give a description of the traversal process, it is convenient to rely on the following abstract notions, whose implementation in the present algorithm is made precise in later sections.

A *node invariant* is an invariant on three-tuples (G,π,ν), where $G \in \mathcal{G}(V)$, $\pi \in \Pi(V)$, and ν is a node of $\mathcal{T}(G,\pi)$. For example, the number of child nodes of a node ν in $\mathcal{T}(G,\pi)$ is a node invariant. A *leaf certificate* is a certificate on three-tuples (G,π,λ), where $G \in \mathcal{G}(V)$, $\pi \in \Pi(V)$, and λ is a leaf node in $\mathcal{T}(G,\pi)$.

An elementary example of a leaf certificate is

$$C(G,\pi,\lambda) = (G^{\bar{\lambda}}, \pi^{\bar{\lambda}}).$$

Indeed, for all $\gamma \in \mathrm{Sym}(V)$ we have

$$\begin{aligned} C(G^\gamma,\pi^\gamma,\lambda^\gamma) &= (G^{\gamma\overline{\lambda^\gamma}}, \pi^{\gamma\overline{\lambda^\gamma}}) = (G^{\gamma\gamma^{-1}\bar{\lambda}}, \pi^{\gamma\gamma^{-1}\bar{\lambda}}) \\ &= (G^{\bar{\lambda}}, \pi^{\bar{\lambda}}) = C(G,\pi,\lambda). \end{aligned}$$

Conversely, we have that $C(G_1,\pi_1,\lambda_1) = C(G_2,\pi_2,\lambda_2)$ implies $G_1^{\bar{\lambda}_1} = G_2^{\bar{\lambda}_2}$ and $\pi_1^{\bar{\lambda}_1} = \pi_2^{\bar{\lambda}_2}$. In other words, for $\gamma = \bar{\lambda}_1\bar{\lambda}_2^{-1}$ it holds that $(G_1^\gamma,\pi_1^\gamma,\lambda_1^\gamma) = (G_2,\pi_2,\lambda_2)$. Thus, $(G_1,\pi_1,\lambda_1) \cong (G_2,\pi_2,\lambda_2)$.

Assuming a leaf certificate, a canonical labeling algorithm is obtained as follows. Because isomorphic colored graphs have isomorphic search trees, the search trees have identical sets of leaf certificate values. We can now arbitrarily declare one value in this set as canonical (say, the minimum value with respect to an appropriate order). It follows that if κ is a leaf node in $\mathcal{T}(G,\pi)$ that has a canonical leaf certificate value, then $(G^{\bar{\kappa}},\pi^{\bar{\kappa}})$ is a canonically labeled version of (G,π), where $\bar{\kappa}$ is a mediating isomorphism. Consequently, a canonical labeling algorithm is obtained by traversing $\mathcal{T}(G,\pi)$ and keeping track of the certificate values at the leaf nodes. To this end, not necessarily all of $\mathcal{T}(G,\pi)$ needs to be traversed; first, it is immediate that one has a lot of freedom in choosing the leaf certificate to use, and a careful choice allows one to restrict the traversal.

3.3 Pruning with a Node Invariant. The present algorithm carries out a depth-first traversal of $\mathcal{T}(G,\pi)$ and makes extensive use of the following prefix pruning scheme, which is essentially due to McKay [31]. Let I be an arbitrary node invariant that assumes values in an ordered set. (The implementation of I in the present algorithm is discussed in detail in §6.) Consider a node ν_ℓ in $\mathcal{T}(G,\pi)$. Associated with ν_ℓ is the unique path $\nu_1 \xrightarrow{x_1} \nu_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{\ell-1}} \nu_\ell$ from the root node ν_1 to ν_ℓ. Define a node invariant $\vec{I}$ by setting

$$\vec{I}(G,\pi,\nu_\ell) = (I_1, I_2, \ldots, I_\ell),$$

where $I_j = I(G,\pi,\nu_j)$ for all $1 \le j \le \ell$, and order the values of $\vec{I}$ lexicographically based on the ordering of the values of I. We assume that the restriction of $\vec{I}$ to leaf nodes is a leaf certificate; an elementary way to meet this assumption is to assume that I is a leaf certificate when restricted to leaf nodes. Selecting the minimum value of $\vec{I}$ at the leaves as the canonical leaf certificate value, it follows—by virtue of lexicographic order—that at each level ℓ of $\mathcal{T}(G,\pi)$, it suffices to traverse only nodes that have the minimum value of $\vec{I}$ among all nodes at level ℓ. This pruning scheme is implemented in the present algorithm by keeping a record of the minimum invariant value so far encountered at a leaf, and disregarding all nodes whose invariant value is lexicographically greater than the current record. To expedite the discovery of automorphisms, this basic scheme is relaxed by allowing the traversal of nodes whose invariant value is a prefix of the invariant value at the first traversed leaf node.

4 Automorphism Discovery and Redundant Subtree Pruning.

To arrive at a practical algorithm for graphs with a large automorphism group, the basic individualization and refinement scheme must be complemented with techniques that enable the discovery of automorphisms during the traversal of $\mathcal{T}(G,\pi)$; also required are techniques that use the discovered automorphisms to prune redundant subtrees of $\mathcal{T}(G,\pi)$. This section provides a brief description of the techniques employed in the present algorithm. For the most part the techniques are similar to those described in [31] and implemented in *nauty* [44], the essential novelty here being that we employ two orbit partitions instead of one to detect redundancy. This achieves additional pruning in certain situations at the negligible cost of maintaining the orbit partition.

4.1 Discovering Automorphisms. The fundamental observation enabling discovery of automorphisms is that any automorphism $\alpha \in \mathrm{Aut}(G,\pi)$ maps the search tree $\mathcal{T}(G,\pi)$ onto itself. In particular, if λ is a leaf node in $\mathcal{T}(G,\pi)$, then so is λ^α, and the two leaf nodes λ, λ^α have equal values of the leaf certificate. Furthermore, $\bar{\lambda}\overline{\lambda^\alpha}^{-1} = \bar{\lambda}(\alpha^{-1}\bar{\lambda})^{-1} = \alpha$. Conversely, if λ_1, λ_2 is a pair of leaf nodes having the same leaf certificate value, then $\alpha = \bar{\lambda}_1\bar{\lambda}_2^{-1} \in \mathrm{Aut}(G,\pi)$ by definition of a leaf certificate. This observation enables us to discover automorphisms while traversing $\mathcal{T}(G,\pi)$.

Fix a reference leaf node ϕ in $\mathcal{T}(G,\pi)$ and record its leaf certificate value; in practice we let ϕ be the first traversed leaf node in a depth-first traversal. When we subsequently traverse a leaf node λ with the same leaf certificate value, we have discovered an automorphism $\bar{\phi}\bar{\lambda}^{-1} \in \mathrm{Aut}(G,\pi)$. It is clear that every $\alpha \in \mathrm{Aut}(G,\pi)$ can be explicitly discovered in this manner. However, in practice we do not want to explicitly discover all the automorphisms, but only a set of generators for $\mathrm{Aut}(G,\pi)$; otherwise we want to avoid traversing redundant parts of the search tree as much as can be achieved in an efficient manner.

4.2 Redundant Subtrees. Let us now make precise what is meant by redundant. Let ϕ be the first traversed leaf node. Suppose that we have either traversed or pruned each node in a subtree of $\mathcal{T}(G,\pi)$ rooted at a node ν, and let $\Phi \leq \mathrm{Aut}(G,\pi)$ be the group generated by all automorphisms discovered so far; in particular, we assume that for every leaf node λ in the subtree rooted at ν it holds that if $\bar{\phi}\bar{\lambda}^{-1} \in \mathrm{Aut}(G,\pi)$, then $\bar{\phi}\bar{\lambda}^{-1} \in \Phi$. We claim that in this situation it is redundant work to traverse the subtree rooted at ν^α for all $\alpha \in \Phi$. To justify this, first observe that because α is an automorphism of (G,π)—and hence an automorphism of $\mathcal{T}(G,\pi)$—it follows that the subtrees rooted at ν and ν^α have identical sets of leaf certificate values at their respective leaf nodes. Thus, by traversing the subtree rooted at ν^α, one would discover no new leaf certificate values, so the traversal is redundant from the perspective of determining the canonical leaf certificate value. Second, for each leaf node λ in the subtree rooted at ν^α we claim that if $\bar{\phi}\bar{\lambda}^{-1} \in \mathrm{Aut}(G,\pi)$, then $\bar{\phi}\bar{\lambda}^{-1} \in \Phi$ holds already. To this end, let λ be an arbitrary leaf node in the subtree rooted at ν^α such that $\bar{\phi}\bar{\lambda}^{-1} \in \mathrm{Aut}(G,\pi)$. Observe that $\lambda^{\alpha^{-1}}$ is a leaf node in the subtree rooted at ν. Because $\mathrm{Aut}(G,\pi)$ is closed under composition, we have

$$\bar{\phi}\overline{\lambda^{\alpha^{-1}}}^{-1} = \bar{\phi}(\alpha\bar{\lambda})^{-1} = \bar{\phi}\bar{\lambda}^{-1}\alpha^{-1} \in \mathrm{Aut}(G,\pi).$$

Thus, $\bar{\phi}\bar{\lambda}^{-1}\alpha^{-1} \in \Phi$ by the assumption on Φ. Since $\alpha \in \Phi$, we have $\bar{\phi}\bar{\lambda}^{-1} \in \Phi$. Thus, traversing the subtree rooted at ν^α is redundant work also from the perspective of discovering generators for $\mathrm{Aut}(G,\pi)$.

4.3 Eliminating Redundancy. Although it is theoretically possible to eliminate all redundancy in the aforementioned sense, this is not done in practice because a complete elimination of redundancy is computationally too expensive compared with the savings gained. The present algorithm relies on three incomplete heuristics to prune redundant subtrees. (Computationally more expensive strategies for eliminating redundancy appear in [25, 26]; see also [37].)

We require some definitions to set up the heuristics. Assuming a reference leaf node ρ, a leaf node $\lambda \neq \rho$ is a *ρ-leaf* with an associated automorphism $\bar{\rho}\bar{\lambda}^{-1}$ if $\bar{\rho}\bar{\lambda}^{-1} \in \mathrm{Aut}(G,\pi)$. Let ϕ be the first traversed leaf node in $\mathcal{T}(G,\pi)$. Call the path from ϕ to the root of $\mathcal{T}(G,\pi)$ the *first path*. Let ψ be the first traversed leaf node whose leaf certificate value is the best candidate so far for the canonical leaf certificate value. Call the path from ψ to the root of $\mathcal{T}(G,\pi)$ the *current best path*. We keep track of two groups, $\Psi \leq \Phi \leq \mathrm{Aut}(G,\pi)$, both initially set to the identity group. The group Φ (respectively, Ψ) is generated by all the automorphisms discovered so far (respectively, by all automorphisms associated with ψ-leaves discovered so far); in particular, Ψ is reset to the identity group whenever ψ changes. For reasons of efficiency, the algorithm stores in memory only the orbit partitions $\{x^\Phi : x \in V\}$ and $\{x^\Psi : x \in V\}$; however, to ease the present exposition it is convenient to track the groups themselves.

First Heuristic: Pruning upon Discovery. The first heuristic occurs in connection with automorphism discovery. In practice, automorphisms are discovered via ρ-leaves with $\rho = \phi$ or $\rho = \psi$. Whenever we traverse a ρ-leaf λ, we first update Φ and Ψ accordingly with

the associated automorphism $\alpha = \bar{\rho}\bar{\lambda}^{-1}$. Let μ be the first common ancestor node of ρ and λ, and let ν_ρ (respectively, ν_λ) be the child of μ whose subtree contains ρ (respectively, λ). The first heuristic now prunes as redundant the subtree rooted at ν_λ; that is, in practice the search backjumps to consider the next child of μ. To justify redundancy, first observe that $\rho^\alpha = \lambda$. Because α is an automorphism of $\mathcal{T}(G, \pi)$, it follows that $\mu^\alpha = \mu$ and that $\nu_\rho^\alpha = \nu_\lambda$. Because of the depth-first traversal order, each node in the subtree rooted at ν_ρ has been either traversed or pruned, so the subtree rooted at ν_λ is redundant.

Second Heuristic: First and Best Path Orbits. The second heuristic is employed at each node μ in the first path and, respectively, at each node in the current best path but not in the first path. When μ is being traversed, the way in which automorphisms are discovered and the depth-first traversal order imply that $\Phi \leq \mathrm{Aut}(G, \mu)$ (respectively, $\Psi \leq \mathrm{Aut}(G, \mu)$). Assume that we have either traversed or pruned each node in the subtree rooted at the x-child of μ for some $x \in V$, and are selecting the next child of μ to traverse. The second heuristic now prunes as redundant all the subtrees rooted at the x^Φ-children (respectively, x^Ψ-children) of μ. To justify this, consider any y-child of μ with $y \in x^\Phi$ (respectively, $y \in x^\Psi$). Observe that $y \in x^\Phi$ and $\Phi \leq \mathrm{Aut}(G, \mu)$ (respectively, $y \in x^\Psi$ and $\Psi \leq \mathrm{Aut}(G, \mu)$) imply that there exists an $\alpha \in \Phi$ (respectively, $\alpha \in \Psi$) with $\mu^\alpha = \mu$ and $x^\alpha = y$. Because α is an automorphism of $\mathcal{T}(G, \pi)$, it follows that α takes the x-child of μ onto the y-child of μ. Thus, the subtree rooted at the y-child is redundant. In certain situations this observation can be used to achieve further pruning in connection with the first heuristic. Indeed, if we discover a ψ-leaf from the subtree rooted at an y-child of a first-path node μ, and the associated automorphism merges the Φ-orbits of y and x for some already traversed or pruned x-child of μ, then we can prune the entire subtree rooted at the y-child as redundant.

Third Heuristic: Stored Automorphisms. The third heuristic keeps track of m of the most recently discovered automorphisms. (In the present algorithm implementation, m is the maximum integer at most 50 such that the storage space required by the automorphisms does not exceed 20 megabytes.) The stored automorphisms now affect which child nodes of a node are traversed in the following manner. Let $\nu_1 \xrightarrow{x_1} \nu_2 \xrightarrow{x_2} \cdots \xrightarrow{x_{\ell-1}} \nu_\ell$ be the path from the root node ν_1 to the current node ν_ℓ, and let A be the set of all stored automorphisms α that satisfy $\alpha(x_i) = x_i$ for all $1 \leq i \leq \ell - 1$. An y-child of ν_ℓ is now traversed only if for every $\alpha \in A$ it holds that either y is fixed by α or y is the minimum element in its cycle in the decomposition of α into disjoint cycles. To justify redundancy, observe that the group generated by A is a subgroup of Φ that fixes ν_ℓ, and it suffices to traverse only one y-child from each orbit of this subgroup on $S(G, \nu_\ell)$.

4.4 Reporting Automorphisms. To limit the number of discovered automorphisms reported to the user, the algorithm reports a discovered automorphism if and only if it merges orbits in the orbit partition of Φ. Thus, at most $n - 1$ automorphisms are reported. To see that the reported automorphisms generate $\mathrm{Aut}(G, \pi)$, observe inductively that whenever we have completed the traversal of a node ν on the first path, the reported automorphisms suffice to generate $\mathrm{Aut}(G, \nu)$. For the inductive step, let ν be the x-child μ, and observe that to obtain a generator set for $\mathrm{Aut}(G, \mu)$ it suffices to report additional automorphisms (if necessary) so that the reported automorphisms move x in the set $x^{\mathrm{Aut}(G,\mu)}$; any necessary automorphisms will be discovered and reported when traversing the children of μ.

5 Data Structures and Subroutines for Ordered Partitions.

This section provides a description of the novel ideas in the low-level implementation of the individualization and refinement scheme in the present algorithm.

5.1 Representing a Chain of Ordered Partitions. Considering a depth-first traversal of the search tree, the main low-level design challenge is presented by the fact that we have to keep track of a chain of ordered partitions $\nu_1 \succ \nu_2 \succ \cdots \succ \nu_\ell$ when descending from the root node towards a leaf node. Since ℓ may be linear in the number of vertices, we are essentially left with constant amortized space for each ordered partition in the chain. Comparing with *nauty* and *saucy*, the present design contribution lies in the careful usage of this space for bookkeeping to avoid costly recomputations and updates when descending and ascending the chain. Also, extra bookkeeping facilitates quick discovery of essential parts of the ordered partition at hand, such as the nonsingleton cells for purposes of refinement and cell selection. In this respect the governing design principle is that we want to avoid the time overhead proportional to the number of vertices that results from accessing the entire ordered partition whenever possible, and rather focus on what is essential to the task at hand, as permitted by the space constraint.

We proceed to describe the low-level data structures in more detail. The basic data structure for representing an ordered partition $\nu_\ell = (W_1, W_2, \ldots, W_m)$ and a chain of its predecessors consists of the following three elements:

(a) An n-element integer array element_array. The elements in each cell are stored in consecutive entries of element_array so that the cells themselves form contiguous subarrays.

(b) An ordered, doubly linked list of cell structures describing the cells. Each cell structure has (i) the integer fields first and length defining the subarray

$$\text{element\_array}[\text{first}, \text{first}+1, \ldots, \text{first}+\text{length}-1]$$

containing the vertices in the cell, and (ii) the integer field in_level containing the level of the chain at which the cell was created.

(c) An n-element array in_cell of pointers to cell structures. For a vertex x, the element in_cell$[x]$ points to the cell structure of the cell that contains x. This information allows one to quickly find the cell in which a neighbor vertex of a vertex belongs to when computing the partition refiner and cell selector functions.

When descending from the root towards the leaves, the basic operation is that of *splitting* a cell W_i into two nonempty subcells Z_1, Z_2, resulting in the finer ordered partition

$$(W_1, W_2, \ldots, W_{i-1}, Z_1, Z_2, W_{i+1}, W_{i+2}, \ldots, W_m).$$

The aforementioned data structure enables the implementation of this operation so that only entries of element_array and in_cell corresponding to the elements in W_i are accessed. Also, the list of cell structures only needs to be accessed at W_i (which becomes Z_1) and its successor cell to insert the new cell structure for Z_2.

When ascending back up the chain, to rapidly undo any number of splits we maintain also a stack data structure refinement_stack of integers recording the values of the first fields of the cells that were split. Also stored is the size of the stack at each level in the chain. The crucial optimization is that when a cell W_i at level ℓ is being recovered, we start with the current cell Y_u whose first value agrees with W_i, and find the first cell Y_v succeeding Y_u in the cell list such that the in_level value of Y_v is at most ℓ. Then, all the cells $Y_u, Y_{u+1}, \ldots, Y_v$ are merged in one pass into W_i. Again observe that only the essential elements of the data structure need to be accessed.

When implementing the partition refiner and the cell selector, for efficiency purposes it is important to access only the nonsingleton cells and not the singleton ones. Therefore, an ordered, doubly linked list of nonsingleton cells is also maintained in conjunction with the data structure for partition chains. When splitting a cell in two, the first fields of the previous and next cell in the list are first stored in a stack (a special value is stored if the split cell was the first or the last in the list), and then the list is updated appropriately (adding the new cell after the split cell if the new cell is nonsingleton, and dropping the split cell if it became singleton). When ascending back up the chain, the stack is consulted to recover the list. We embed the list of nonsingleton cells in the cell structure so that only the cells that change during refinement or backtracking have to be visited or modified in the list.

5.2 Refinement to an Equitable Ordered Partition. A colored graph (G, π) is *equitable* if any two vertices with the same color have an equal number of neighbors of each color. The number of neighbors of a given color is alternatively called the *color degree* of a vertex. Assuming that the graph G is clear from the context, it is a convenient abuse of terminology to call an ordered partition π equitable if (G, π) is an equitable colored graph.

The partition refiner $R(G, \pi)$ used in the present algorithm implementation, as well as in *nauty* and *saucy*, refines a given ordered partition π into a maximum equitable ordered partition ϵ such that $\epsilon \preceq \pi$. (The ordered partition ϵ is unique up to ordering of the cells resulting from splits.) Pseudocode for an equitable partition refiner can be found in [26, 31].

At implementation level, the key operation during refinement is to split nonsingleton cells having neighbors in a cell W into cells of vertices that have the same number of neighbors in W. This is straightforward to achieve by traversing all neighbors of each vertex in W, counting how many times each neighbor is visited, and splitting the neighbor cells according to these counts. To this end, the partition chain data structure and the adjacency list representation for the input graph provide the advantage that only cells including vertices that are neighbors of vertices in W need to be accessed.

To provide a further implementation speedup, for each neighbor cell Y of W, we keep track of the maximum number of neighbors that a vertex in Y has in W together with an occurrence count for the maximum. This allows one to immediately discover whether Y has to be split and also assists in selecting a fast sorting algorithm when preparing element_array for the split. In particular, if the maximum neighbor count is less than 2 (respectively, 256), an ad-hoc one pass sort (respectively, a bucket sort) is applied.

5.3 Cell Selection. Two distinct cells in an equitable colored graph (G, π) are *nonuniformly joined* if each vertex in one cell has both neighbors and non-

neighbors in the other cell. The default cell selector in the present algorithm is identical to the default cell selector employed in *nauty*. Namely, we select the first nonsingleton cell that is nonuniformly joined to the maximum number of other cells.

At implementation level, the main difference to *nauty* is that the present data structures allow for a relatively efficient evaluation of the cell selector. Thus, we can use the cell selector in each node of the search tree and not only at the first few levels.

The present data structures also facilitate the evaluation of other types of cell selectors. For example, the first nonsingleton cell can be found in constant time because we keep track of the nonsingleton cells. Also the first nonsingleton cell of the minimum size can be found relatively efficiently. These two cell selectors introduce less overhead and result in faster operation on many graphs lacking significant regularity, but appear to be less stable in general. The subsequent experiments are carried out using the default cell selector only.

6 Incremental Leaf Certificate and Further Pruning.

This section provides a high-level description of the combined node invariant and leaf certificate in the present algorithm. Also discussed is a further pruning technique.

6.1 Incremental Leaf Certificate. The leaf certificate in the present algorithm is based on essentially three ideas. First, we attempt to extract at each node the maximum amount of invariant information that is available with minor overhead from the basic traversal process—in this respect the present implementation closely parallels *nauty* [31, 44]. Second, we amortize the leaf certificate construction by shifting the computational effort from the leaves towards the root, where the partial certificate provides additional invariant information to assist in pruning. Third, if we know that we can prune the current node and backtrack, we prefer to do so at the earliest possible point of execution in the low-level implementation. Compared with earlier algorithm designs, the last two ideas are the main novelty in the present design. In what follows we give a high-level description of the implementation of these ideas; the actual low-level implementation is best studied from the source code.

The main source of cheap labeling-invariant information during the traversal of the search tree is the partition refiner. Labeling-invariant information that can be extracted from the refinement process includes the ordered tuple of cell sizes at each step of refinement, the index of a cell that splits, the invariant values (e.g. the number of neighbors of each color) of the vertices in a cell together with their multiplicity, and so forth.

Let (G, π) be the colored graph given as input, $G = (V, E)$, and consider a node ν_ℓ at level ℓ in $\mathcal{T}(G, \pi)$. The component node invariant, $I(G, \pi, \nu_\ell)$, consists of two parts,

$$I(G, \pi, \nu_\ell) = (F(G, \pi, \nu_\ell), H(G, \pi, \nu_\ell)).$$

The first part, F, consists of a fragment of a leaf certificate. The second part, H, consists of invariant information accumulated into a hash digest during the refinement process—for the details of this accumulation we refer to the source code.

The fragment F is constructed as follows. For a node ν_ℓ at level ℓ in $\mathcal{T}(G, \pi)$, let $\nu_{\ell-1}$ be the parent node of ν_ℓ; for the root node ν_1, let ν_0 be the unit ordered partition. Let $S \subseteq V$ consist of all vertices that occur in singleton cells in ν_ℓ but not in $\nu_{\ell-1}$. Let λ be any discrete ordered partition with $\lambda \preceq \nu_\ell$. For each $u \in S$ and each $v \in V$ such that $\{u, v\} \in E$, form the ordered pair $(u^{\bar{\lambda}}, v^{\bar{\lambda}})$. Let $F(G, \pi, \nu_\ell)$ be the ordered tuple consisting of these ordered pairs in lexicographic order; for the root node, also include the ordered partition $\pi^{\bar{\lambda}}$ into the tuple. Now observe that because ν_ℓ is equitable, $F(G, \pi, \nu_\ell)$ is independent of the choice of $\lambda \preceq \nu_\ell$. For every leaf node λ it follows that $\vec{I}(G, \pi, \lambda)$ essentially consists of the colored graph $(G^{\bar{\lambda}}, \pi^{\bar{\lambda}})$ interleaved with invariant values from $\vec{H}(G, \pi, \lambda)$. Thus, $\vec{I}(G, \pi, \nu_\ell)$ is a leaf certificate when restricted to the leaf nodes.

Besides amortizing the leaf certificate construction effort towards the root node, the lexicographic organization of the leaf certificate facilitates early pruning already during the refinement process. Indeed, observe that as soon as a singleton cell emerges during refinement, we can determine the edges $(u^{\bar{\lambda}}, v^{\bar{\lambda}})$ associated with that cell. By virtue of lexicographic order, this information allows us to sometimes prune the emerging node without ever completing the refinement process. Since the refinement process accounts for the majority of the execution time, this simple early abortion scheme sometimes results in substantial gains in performance.

6.2 Pruning via Transitive First-Path Subtrees. To give a complete description of the present algorithm, we provide a brief description of the following pruning heuristic, which is identical to one employed in *nauty* [31, 44].

Suppose that $\mathrm{Aut}(G, \pi)$ acts transitively on the leaf nodes in a subtree rooted at a first-path node μ, and note that this can be detected by tracking the orbit partition of Φ when traversing the first-path nodes. Suppose that we are currently traversing a node ν at a level at least that of μ. If the node invariant value of ν is

greater than the current minimum leaf certificate value encountered, and not a prefix of the leaf certificate value at the first leaf node ϕ, then we can backtrack to one level above the level of μ, or to the first level whose node invariant value is a prefix of the current minimum leaf certificate value, whichever level is the maximum (that is, provides the least pruning). To justify this, observe on one hand that the pruned nodes cannot improve the minimum leaf certificate value encountered so far, and, on the other hand, no ϕ-leaf is pruned because by transitivity a subtree containing a ϕ-leaf cannot contain a node with invariant value equal to that of ν.

7 Experiments.

In this section we report on experiments that compare the performance of the present algorithm with *nauty* and *saucy*.

7.1 Benchmark Families. In the experiments we consider the following families of graphs derived from various combinatorial objects, lower bound constructions, and industrial problem settings. Descriptions of the combinatorial objects and related further references can be found in [13].

Affine and projective geometries The benchmark series `ag2-`$\langle q\rangle$ and `pg2-`$\langle q\rangle$ consist of the bipartite point-line incidence graphs of the 2-dimensional affine and projective geometries $\mathrm{AG}_2(q)$ and $\mathrm{PG}_2(q)$ over the field $\mathrm{GF}(q)$.

Cai–Fürer–Immerman construction The series `cfi-`$\langle n\rangle$ consists of 3-regular graphs built from a random 3-regular graph on n vertices by replacing each vertex with a gadget graph and each edge with two edges as described in [11].

Constraint satisfaction problems The series `chnl`, `difp`, `fpga`, `hole`, `s3-3-3`, and `Urq` consists of graphs translated from conjunctive normal form propositional satisfiability instances available at [42] by using the construction described in [14].

Hadamard matrices The series `had-`$\langle n\rangle$ consists of graphs built from $n \times n$ Hadamard matrices obtained using various constructions [40]. The series `had-sw-`$\langle n\rangle$ is derived by repeated application of Orrick's switching operations [35] to the base Hadamard matrices to reduce the symmetry.

Miyazaki construction Miyazaki's construction [32] is based on the Cai–Fürer–Immerman construction applied to a specific 3-regular multigraph. The series `mz-`$\langle n\rangle$ consists of the graphs resulting from the base construction. The series `mz-aug-`$\langle n\rangle$ and `mz-aug2-`$\langle n\rangle$ consists of graphs in which the base construction is reinforced with gadgets to mislead the cell selector.

Projective planes The series `pp16-`$\langle m\rangle$ consists of the bipartite point-line incidence graphs of the known projective planes of order 16 [41].

Random regular graphs The series `rnd-3-reg-`$\langle n\rangle$ consists of random 3-regular graphs on n vertices constructed by rejection sampling using the configuration model [9, §2.4].

Strongly regular graphs The series `latin-`$\langle n\rangle$ consists of Latin square graphs $\mathrm{srg}(n^2, 3(n-1), n, 6)$. The series `lattice-`$\langle n\rangle$ consists of lattice graphs $\mathrm{srg}(n^2, 2(n-1), n-2, 2)$. The series `paley-`$\langle q\rangle$ consists of Paley graphs $\mathrm{srg}(q, (q-1)/2, (q-5)/4, (q-1)/4)$. The series `sts-`$\langle v\rangle$ consists of line graphs $\mathrm{srg}(v(v-1)/6, 3(v-3)/2, (v+3)/2, 9)$ of Steiner triple systems. The series `triang-`$\langle n\rangle$ consists of triangular graphs $\mathrm{srg}(n(n-1)/2, 2(n-2), n-2, 4)$. The series `latin-sw-`$\langle n\rangle$ and `sts-sw-`$\langle v\rangle$ consist of aforementioned families whose base objects are perturbed using repeated Jacobson–Matthews switching operations [12, 20] to reduce the symmetry.

Miscellany The series `k-`$\langle n\rangle$ consists of complete graphs K_n. The series `grid-`$\langle d\rangle$`-`$\langle n\rangle$ and `grid-w-`$\langle d\rangle$`-`$\langle n\rangle$ consist of d-dimensional grid graphs where n is the length of each dimension; the latter series has wrapping boundary.

These families are motivated by the following general considerations. First, to evaluate the efficiency of the basic data structures, graphs with significant regularity but little symmetry (as recorded by the automorphism group) are required. Second, graphs with extensive symmetries are required to evaluate the heuristics for eliminating redundancy. Third, to evaluate the data structures and subroutines on large graphs encountered in industrial contexts, graphs originating from industrial applications are required—while such graphs are arguably not as challenging as graphs of combinatorial origin, their sheer size presents an effective scalability test.

7.2 Experimental Setup. The present tool, which we have chosen to call *bliss*, and the benchmark graphs are available at [46]. The other tools considered in the experiments are *nauty* (version 2.2) [44] and *saucy* (version 1.1) [45].

The experiments were carried out on a network of 105 Dell OptiPlex Linux PCs with 2.8-GHz Intel Pentium 4 CPUs with 512 KB of in-processor cache and 1 GB of main memory. Each tool was compiled with the GNU C Compiler (version 3.3.5) with the compilation flags as configured by each tool.

The experiments consisted of randomly relabeling every graph in every benchmark series 11 times, and executing each tool on each graph with the canonical

labeling option set to both true and false (if appropriate). The $5 \cdot 11 = 55$ experiments associated with each individual graph in the benchmark set were executed in succession on the same physical hardware.

From each experiment we recorded the execution time, as reported by the operating system, and the number of nodes in the corresponding search tree, as reported by each tool. A time limit of 600 seconds for each experiment was imposed, after which the experiment was aborted. The total amount of CPU time consumed by the experiments was about 5.38 million seconds (62 days).

7.3 Results and Observations. Due to space constraints, we can provide here only a relatively succinct summary of the data obtained from the experiments; the reader is encouraged to consult [46] for a more extensive summary, on which some of the subsequent detailed observations for individual benchmark series rely.

To provide a brief summary of the experiments, Fig. 1 consists of 9 scatter plots, each providing a pairwise comparison of the two indicated tools. Each plot displays all the experiments associated with the two tools in question; a logarithmic scale is used for both axes. Experiments that have timed out are assigned the values 600 (time) and 10^9 (traversed nodes) in the plots.

Figures 1(a) and 1(b) compare *bliss* with *nauty* in the situation where only generators for the automorphism group are computed. Figure 1(a) plots the running time for both tools on each experiment, and Fig. 1(b) plots the number of nodes traversed in the respective search trees. It is immediate that *bliss* outperforms *nauty* on the benchmark set in terms of running time. Looking at the number of traversed nodes, however, it appears that the numbers are roughly the same but with some structural differences, for example, associated with the series `latin-sw` and `sts-sw`. The difference in running times appears largely to be explained by the fact that *nauty* suffers from the incidence matrix representation and overhead apparently caused by nonlocality in the data structures; this is observed, for example, in the series `cfi` and `rnd-3-reg`.

Figures 1(c) and 1(d) compare *bliss* with *saucy*. We observe that *bliss* clearly outperforms *saucy* on the benchmark set both in terms of running time and in terms of the traversed search tree nodes. It appears that *bliss* has more effective heuristics for structuring and pruning the search tree (series `ag`, `cfi`, `mz`, `mz-aug` and `pg`). Also the basic traversal process is more efficient (series `rnd-3-reg` in particular and also `latin-sw` and `sts-sw`).

Figures 1(e) and 1(f) compare *saucy* and *nauty*. From Fig. 1(f) it is immediate that *nauty* is clearly superior in terms of heuristics that structure and prune the search tree (series `ag`, `cfi`, `mz`, `mz-aug`, and `pg`); however, in terms of the running times in Fig. 1(e) it appears that *nauty* suffers at least from the incidence matrix representation (series `difp` and `rnd-3-reg`).

Figures 1(g) and 1(h) compare *bliss* and *nauty* in the situation where the canonical labeling option set to true. Comparing with Figs. 1(a) and Figs. 1(b), the same general behavior occurs—*bliss* in general outperforms *nauty* in terms of the running time. However, now there are graphs for which *bliss* becomes unstable already on slightly smaller graphs than *nauty* (series `mz` and `mz-aug`).

Figure 1(i) compares the running times of *bliss* with and without the canonical labeling option. It appears that the additional cost incurred by computing a canonical labeling is in a large number of cases not significant compared with the cost of computing only a set of generators for the automorphism group. Again there are exceptions in which the canonical labeling incurs an additional cost (series `cfi` and `mz`), sometimes quite radically (series `mz-aug`).

To provide a somewhat more detailed analysis within the present paper, Fig. 2 displays 12 more scatter plots that compare *bliss* with *nauty* for selected benchmark series.

Figures 2(a) and 2(b) illustrate for the series `cfi` apparently a polynomial improvement in running time obtained compared with *nauty*; indeed, while the number of traversed nodes in the search trees is roughly the same in Fig. 2(b), based on Fig. 2(a) it appears that *bliss* is faster by a factor that grows roughly polynomially in n. This improvement in running time is most likely explained by more efficient data structures and subroutines.

Figures 2(c) and 2(d) show that the present data structures and heuristics are competitive with *nauty* also for the relatively dense graphs in the series `had`. For example, *nauty* times out more often in situations where *bliss* is still able to produce a canonical labeling within the allocated time. The competitiveness is perhaps even more clearly visible in Figs. 2(e) and 2(f) displaying the series `had-sw`, where the graphs have only little symmetry. Here it is immediate that both the running times and the number of traversed nodes favor *bliss*.

Figures 2(g) and 2(h) illustrate the difference in performance for the series `sts-sw`. Observe in Fig. 2(h) that while *bliss* appears to traverse polynomially more nodes than *nauty*, the running times in Fig. 2(g) are roughly the same. This illustrates the efficiency of the data structures, but also shows that the heuristics have a strong effect on performance. In this particular case, the larger number of traversed nodes results from a tie

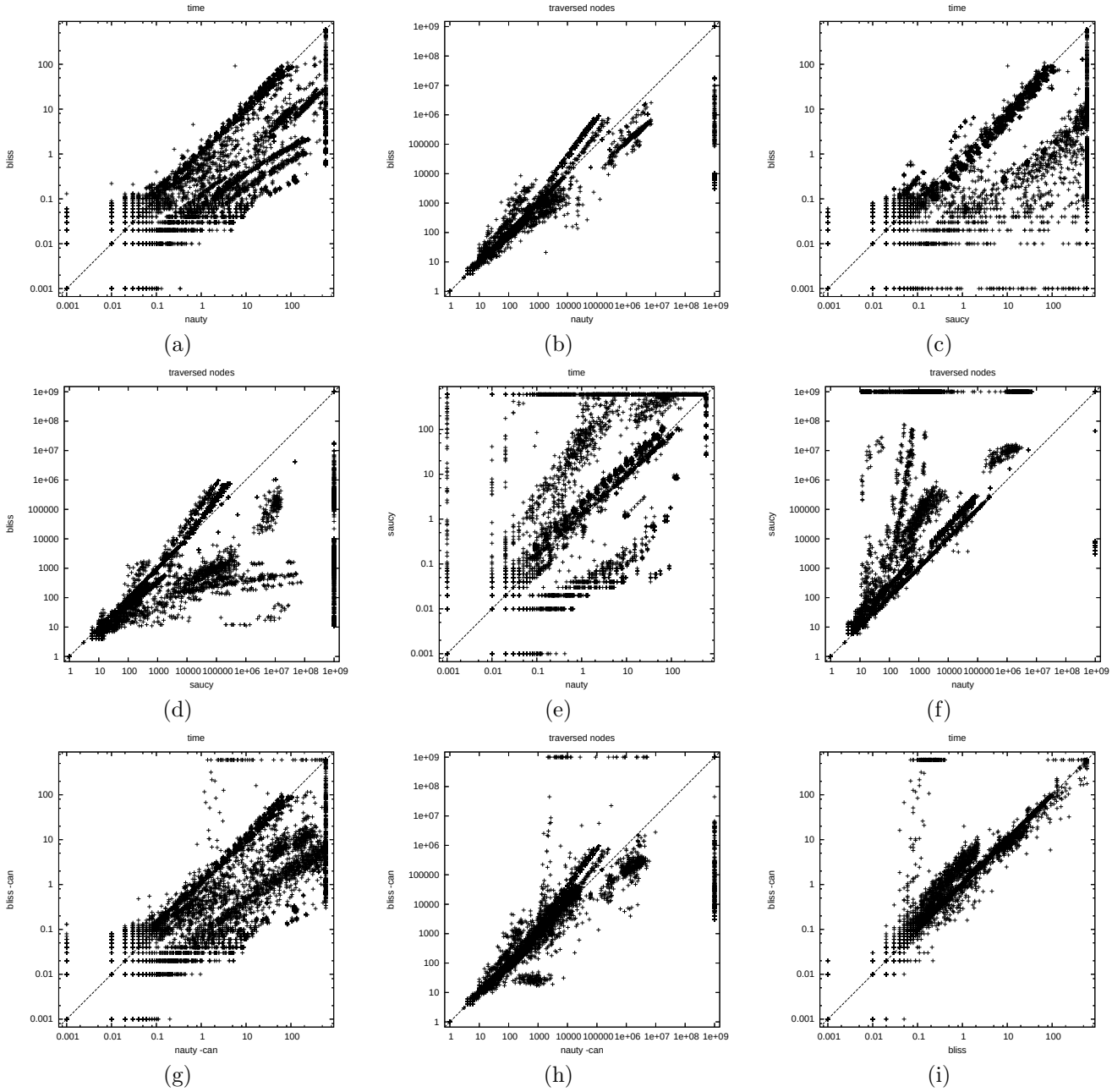
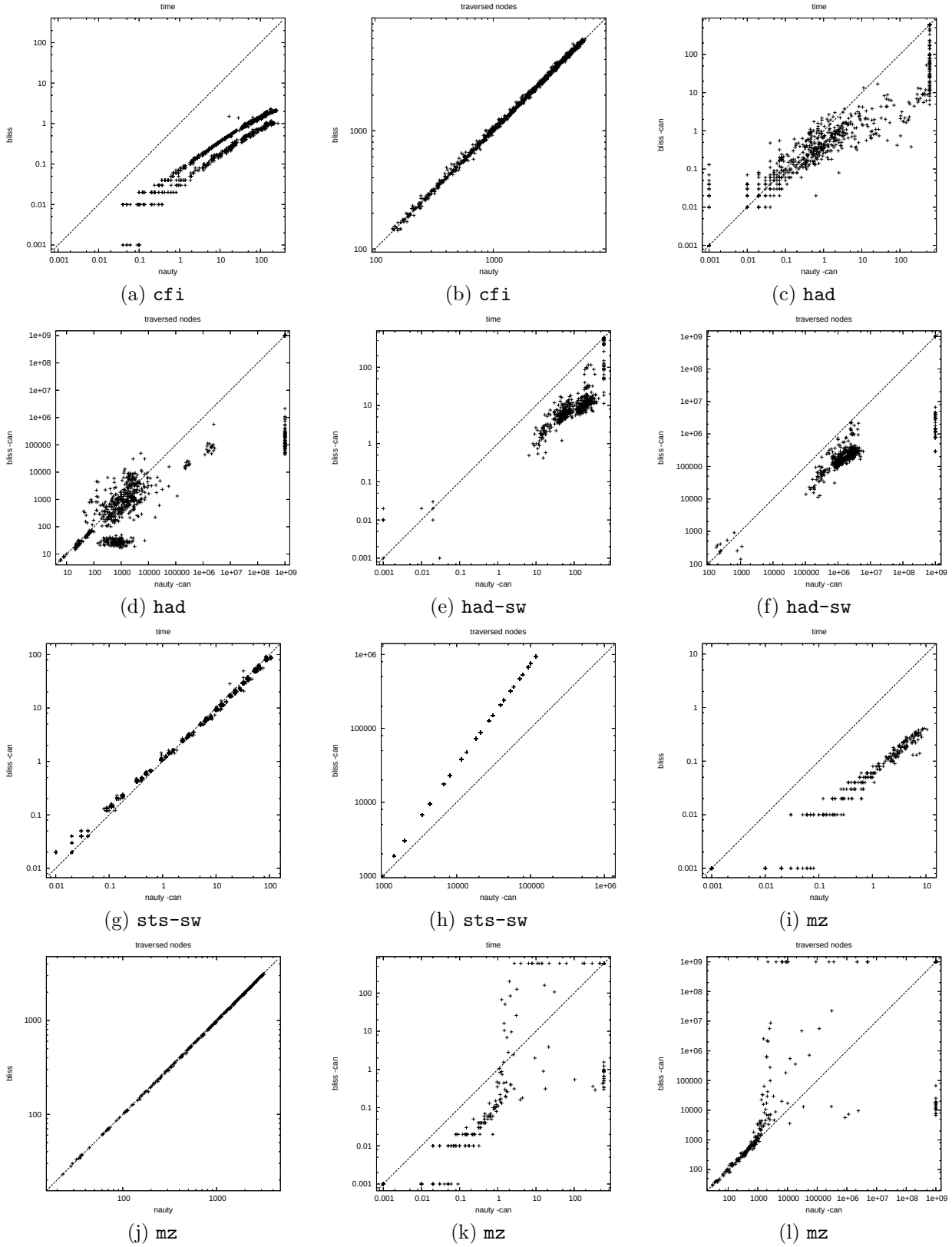


Figure 1: Scatter plots summarizing the experiments

in the cell selector between the two nonsingleton cells in each child node of the root node—apparently *bliss* selects the larger cell and *nauty* the smaller cell due to different ordering of the cells. Similar results are obtained for the series `latin`, `latin-sw`, and `sts`.

Figures 2(i) to 2(l) illustrate the sometimes dramatic additional cost incurred by canonical labeling using the series `mz`. In Figs. 2(i) and 2(j) the canonical labeling option is set to false. We see that *bliss* is faster, and that the numbers of traversed nodes are roughly equal for both tools. In Figs. 2(k) and 2(l) the canonical labeling option is set to true, and a qualitatively different situation emerges. Now neither tool dominates the other, and we see extensive timeouts, apparently due to the fact that "bad" choices made by the cell selector cause the algorithm to traverse an exponentially larger subtree compared with a "good" choice. Similar results are obtained for the series `mz-aug`.



(a) `cfi` (b) `cfi` (c) `had`

(d) `had` (e) `had-sw` (f) `had-sw`

(g) `sts-sw` (h) `sts-sw` (i) `mz`

(j) `mz` (k) `mz` (l) `mz`

Figure 2: Scatter plots for selected benchmark series

Table 1: Results for `mz-aug2` series

graph	*nauty* nodes	*nauty* time	*nauty* -can nodes	*nauty* -can time	*bliss* nodes	*bliss* time	*bliss* -can nodes	*bliss* -can time
mz-aug2-4	133–133	0–0	133–141	0–0	92–92	0–0	92–116	0–0
mz-aug2-6	599–599	0–0	599–631	0–0	310–310	0–0	311–403	0–0
mz-aug2-8	2777–2777	0–0	2777–2905	0–0	1112–1112	0–0	1120–1591	0–0
mz-aug2-10	12931–12931	0–0	12931–13443	0–0	4226–4226	0–0	4234–6105	0–0
mz-aug2-12	61621–61621	1–1	61621–61621	3–3	16564–16564	0–0	16700–24711	0–0
mz-aug2-14	270575–270575	5–5	270575–278767	22–22	65774–65774	1–1	66442–93147	1–2
mz-aug2-16	1212721–1212721	30–30	1212721–1245489	131–133	262448–262448	6–6	269280–390154	7–11
mz-aug2-18	5374331–5374331	147–166	?–?	t.o.–t.o.	1048954–1048954	27–33	1060666–1437802	33–47
mz-aug2-20	?–?	t.o.–t.o.	?–?	t.o.–t.o.	4194764–4194764	126–129	4217180–6290680	148–229
mz-aug2-22	?–?	t.o.–t.o.	?–?	t.o.–t.o.	16777766–?	532–t.o.	?–?	t.o.–t.o.

Table 2: Results for `rnd-3-reg` series

graph	*nauty* nodes	*nauty* time	*saucy* nodes	*saucy* time	*bliss* nodes	*bliss* time
rnd-3-reg-1000-1	1001–1001	9.00–9.08	1001–1001	1.11–1.18	1001–1001	0.06–0.07
rnd-3-reg-2000-1	2001–2001	117.13–120.74	2001–2001	8.57–8.63	2001–2001	0.27–0.30
rnd-3-reg-3000-1	?–?	t.o.–t.o.	3001–3001	28.34–28.44	3001–3001	0.64–0.71
rnd-3-reg-4000-1	?–?	t.o.–t.o.	4001–4001	62.22–64.50	4001–4001	1.09–1.20
rnd-3-reg-5000-1	?–?	t.o.–t.o.	5001–5001	122.23–129.40	5001–5001	1.70–1.93
rnd-3-reg-6000-1	?–?	t.o.–t.o.	6001–6001	208.65–215.22	6001–6001	2.65–2.98
rnd-3-reg-7000-1	?–?	t.o.–t.o.	7001–7001	335.10–365.94	7001–7001	3.61–4.04
rnd-3-reg-8000-1	?–?	t.o.–t.o.	8001–8001	499.67–508.16	8001–8001	4.87–5.49
rnd-3-reg-9000-1	?–?	t.o.–t.o.	?–?	t.o.–t.o.	9001–9001	7.43–8.28
rnd-3-reg-10000-1	?–?	t.o.–t.o.	?–?	t.o.–t.o.	10001–10001	7.90–8.74

To illustrate exponential scaling, Table 1 displays results for the series `mz-aug2`. Displayed in the table are both the number of traversed nodes and the running times for *bliss* and *nauty*, with both settings of the canonical labeling option. Each row in the table corresponds to one input graph; the interval in each column consists of the minimum and maximum values over the 11 random relabelings for each graph. A time out is indicated by "t.o." in the table. As can be seen from the table, the number of nodes traversed for both tools grows roughly exponentially in n, which is explained by the fact that the cell selector is systematically misled into making "bad" choices in the top levels of the search tree (see [32] for a detailed combinatorial analysis).

To illustrate scaling on large and sparse graphs, Table 2 displays partial results for the series `rnd-3-reg`. Displayed in the table are both the number of traversed nodes and the running times for *bliss*, *nauty*, and *saucy*. As can be seen from the table, all tools traverse the same number of nodes, but *bliss* alone handles random 3-regular graphs up to 10000 vertices within the allocated time. Furthermore, it appears that the running time of *bliss* grows roughly as a quadratic function in the number of vertices, whereas the running time of *saucy* is grows roughly as a cubic function in the number of vertices. The poor performance of *nauty* is apparently largely explained by the incidence matrix representation for the input graph, which becomes cumbersome for large and sparse graphs. The difference in running times between *bliss* and *saucy* is apparently explained by differences in implementation of the data structures and subroutines that traverse the search tree.

For more extensive data on the experiments, complete tables and scatter plots covering all the benchmark families are available at [46].

8 Conclusions.

The objective of the present paper has been to improve the practical performance of canonical labeling algorithms relying on the individualization and refinement scheme, especially in the context of large and sparse graphs. The main contributions of the present paper are (a) novel data structures to accommodate large graphs and to facilitate fast traversal of the search tree, (b) amortization of work from the leaves towards the root of the search tree, and (c) improvement of heuristics for pruning the search.

Experiments indicate that the present tool in most cases clearly outperforms existing public tools on large and sparse graphs, and exhibits comparable or better performance also on dense and highly regular graphs of combinatorial origin. In summary, we believe that the present tool is one of the fastest general-purpose tools for canonical labeling of graphs currently in existence.

It is, however, necessary to emphasize that the aforementioned claim concerns only general-purpose tools. Indeed, it is immediate that many restricted families of graphs admit more efficient techniques both in theory and in practice. Furthermore, whenever one considers a specific family of graphs, the basic algorithm design can typically be optimized for performance by

applying tailored isomorphism invariants during the refinement process and by tailoring the cell selector function. Thus, in this respect one can view the present contribution as providing an improved template for the construction of tailored algorithms.

Although this is not analyzed in detail in the reported experiments, to arrive at a balanced algorithm with stable performance on a broad range of graphs, our general observation during the present engineering effort has been that it is necessary to incorporate all of the heuristics discussed in this paper into the algorithm. If a heuristic is omitted, then performance will in general suffer on some benchmark families. This is also why the present algorithm design quite closely parallels *nauty* in terms of the heuristics employed; in our experience, the heuristics in *nauty* are very well balanced for practical performance. If one aims for a tailored algorithm, however, then a careful consideration should be given as to which heuristics to include and which to leave out.

In addition to the algorithm engineering effort, we have also made available an electronic catalogue of graphs to provide a testbed for canonical labeling tools. Many of the benchmark families are arguably among the most difficult families of graphs currently known in the context of canonical labeling and practical performance. Keeping this in mind, it is interesting to observe that the augmented Miyazaki construction provides currently the only family with clear exponential scaling as the size parameter is increased. Further contributions to the catalogue are encouraged.

We conclude the paper with the general observation that there is an apparent gap between theory and practice in the context of canonical labeling. In particular, there is a notable absence of actual implementations of the algorithms with the best theoretical running time bounds, even in many cases where polynomial-time algorithms are known, such as for bounded-degree graphs [6]. From the perspective of practical performance, it appears that the individualization and refinement scheme remains uncontested.

Acknowledgments

This research was supported in part by the Academy of Finland, Grant 117499. The authors thank Jani Jaakkola and Pekka Niklander at the Department of Computer Science of University of Helsinki for providing computing resources for the experiments.

References

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, Solving difficult instances of Boolean satisfiability in the presence of symmetry, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22 (2003), 1117–1137.

[2] V. Arvind and J. Torán, Isomorphism testing: Perspective and open problems, *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS* 86 (2005), 66–84.

[3] L. Babai, Moderately exponential bound for graph isomorphism, in *Fundamentals of Computation Theory* (F. Gécseg, Ed.), Springer-Verlag, Berlin, 1981, pp. 34–50.

[4] L. Babai, Automorphism groups, isomorphism, reconstruction, in *Handbook of Combinatorics* (R. L. Graham, M. Grötschel, and L. Lovász, Eds.), Vol. 2, Elsevier, Amsterdam, 1995, pp. 1447–1540.

[5] L. Babai, D. Yu. Grigoryev, and D. M. Mount, Isomorphism of graphs with bounded eigenvalue multiplicity, in *Proc. 14th ACM Symposium on Theory of Computing*, (San Francisco, May 5–7, 1982), ACM Press, New York, 1982, pp. 310–324.

[6] L. Babai and E. M. Luks, Canonical labeling of graphs, in *Proc. 15th ACM Symposium on Theory of Computing*, (Boston, Apr. 25–27, 1983), ACM Press, New York, 1983, pp. 171–183.

[7] L. Babai and S. Moran, Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes, *J. Comput. System Sci.* 36 (1988), 254–276.

[8] B. Bollobás, *Modern Graph Theory*, Springer-Verlag, New York, 1998.

[9] B. Bollobás, *Random Graphs*, 2nd ed., Cambridge University Press, Cambridge, 2001.

[10] G. Brinkmann, Isomorphism rejection in structure generation programs, in *Discrete Mathematical Chemistry* (P. Hansen, P. Fowler, and M. Zheng, Eds.), Amer. Math. Soc., Providence, R.I., 2000, pp. 25–38.

[11] J.-Y. Cai, M. Fürer, and N. Immerman, An optimal lower bound on the number of variables for graph identification, *Combinatorica* 12 (1992), 389–410.

[12] P. J. Cameron, Random strongly regular graphs? *Discrete Math.* 273 (2003), 103–114.

[13] C. J. Colbourn and J. H. Dinitz, Eds., *Handbook of Combinatorial Designs*, 2nd ed., Chapman & Hall/ CRC, Boca Raton, Fla., 2007.

[14] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, Exploting structure in symmetry detection for CNF, in *Proc. 41th ACM IEEE Design Automation Conference*, (San Diego, June 7–11, 2004), ACM Press, New York, 2004, pp. 530–534.

[15] J. D. Dixon and B. Mortimer, *Permutation Groups*, Springer-Verlag, New York, 1996.

[16] S. A. Evdokimov and I. N. Ponomarenko, Recognition and verification of an isomorphism of circulant graphs in polynomial time (in Russian), *Algebra i Analiz* 15 (2003), 1–34. English translation in *St. Petersburg Math. J.* 15 (2004), 813–835.

[17] J.-L. Faulon, M. J. Collins, and R. D. Carr, The signature molecular descriptor. 4. Canonizing molecules using extended valence sequences, *J. Chem. Inf. Comput. Sci.* 44 (2004), 427–436.

[18] M. Goldberg, The graph isomorphism problem, in

Handbook of Graph Theory (J. L. Gross and J. Yellen, Eds.), CRC Press, Boca Raton, Fla., 2004, pp. 68–78.

[19] C. M. Hoffmann, *Group-Theoretic Algorithms and Graph Isomorphism*, Springer-Verlag, Berlin, 1982.

[20] M. T. Jacobson and P. Matthews, Generating uniformly distributed random Latin squares, *J. Combin. Des.* 4 (1996), 405–437.

[21] T. Junttila, On the symmetry reduction method for Petri nets and similar formalisms, Research Report A80, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2003. Doctoral dissertation.

[22] P. Kaski and P. R. J. Östergård, *Classification Algorithms for Codes and Designs*, Springer-Verlag, Berlin, 2006.

[23] A. R. Klivans and D. van Melkebeek, Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses, *SIAM J. Comput.* 31 (2002), 1501–1526.

[24] J. Köbler, U. Schöning, and J. Torán, *The Graph Isomorphism Problem: Its Structural Complexity*, Birkhäuser, Boston, 1993.

[25] W. Kocay, On writing isomorphism programs, in *Computational and Constructive Design Theory* (W. D. Wallis, Ed.), Kluwer, Dordrecht, the Netherlands, 1996, pp. 135–175.

[26] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press, Boca Raton, Fla., 1999.

[27] A. Lubiw, Some NP-complete problems similar to graph isomorphism, *SIAM J. Comput.* 10 (1981), 11–21.

[28] E. M. Luks, Isomorphism of graphs of bounded valence can be tested in polynomial time, *J. Comput. System Sci.* 25 (1982), 42–65.

[29] E. M. Luks, Permutation groups and polynomial-time computation, in *Groups and Computation* (L. Finkelstein and W. M. Kantor, Eds.), Amer. Math. Soc., Providence, R.I., 1993, pp. 139–175.

[30] R. Mathon, A note on the graph isomorphism counting problem, *Inform. Process. Lett.* 8 (1979), 131–132.

[31] B. D. McKay, Practical graph isomorphism, *Congr. Numer.* 30 (1981), 45–87.

[32] T. Miyazaki, The complexity of McKay's canonical labeling algorithm, in *Groups and Computation, II* (L. Finkelstein and W. M. Kantor, Eds.), Amer. Math. Soc., Providence, R.I., 1997, pp. 239–256.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: Engineering an efficient SAT solver, in *Proc. 38th ACM IEEE Design Automation Conference*, (Las Vegas, June 18–22, 2001), ACM Press, New York, 2001, pp. 530–535.

[34] M. Muzychuk, A solution of the isomorphism problem for circulant graphs, *Proc. London Math. Soc. (3)* 88 (2004), 1–14.

[35] W. P. Orrick, Switching operations for Hadamard matrices, preprint available at ⟨`http://arxiv.org/abs/math.CO/0507515`⟩.

[36] R. C. Read and D. G. Corneil, The graph isomorphism disease, *J. Graph Theory* 1 (1977), 339–363.

[37] Á. Seress, *Permutation Group Algorithms*, Cambridge University Press, Cambridge, 2003.

[38] J. Torán, On the hardness of graph isomorphism, *SIAM J. Comput.* 33 (2004), 1093–1108.

[39] B. Weisfeiler, Ed., *On Construction and Identification of Graphs*, Springer-Verlag, Berlin, 1976.

[40] [Data]: A library of Hadamard matrices maintained by N. J. A. Sloane, ⟨`http://www.research.att.com/~njas/hadamard/`⟩.

[41] [Data]: A library of projective planes of order 16 maintained by G. Royle, ⟨`http://www.csse.uwa.edu.au/~gordon/remote/planes16/index.html`⟩.

[42] [Data]: A library of SAT benchmarks maintained by F. Aloul, ⟨`http://www.eecs.umich.edu/~faloul/benchmarks.html`⟩.

[43] [Link]: The SATLive! page, ⟨`http://satlive.org/`⟩.

[44] [Software]: nauty (version 2.2), ⟨`http://cs.anu.edu.au/~bdm/nauty/`⟩.

[45] [Software]: saucy (version 1.1), ⟨`http://vlsicad.eecs.umich.edu/BK/SAUCY/`⟩.

[46] [Software]: The proposed tool, benchmark graphs, and a summary of the experiments, ⟨`http://www.tcs.hut.fi/Software/benchmarks/ALENEX-2007/`⟩.