A Generic Framework for Engineering Graph Canonization Algorithms

JAKOB L. ANDERSEN and DANIEL MERKLE, University of Southern Denmark

The state-of-the-art tools for practical graph canonization are all based on the individualization-refinement paradigm, and their difference is primarily in the choice of heuristics they include and in the actual tool implementation. It is thus not possible to make a direct comparison of how individual algorithmic ideas affect the performance on different graph classes.

We present an algorithmic software framework that facilitates implementation of heuristics as independent extensions to a common core algorithm. It therefore becomes easy to perform a detailed comparison of the performance and behavior of different algorithmic ideas. Implementations are provided of a range of algorithms for tree traversal, target cell selection, and node invariant, including choices from the literature and new variations. The framework readily supports extraction and visualization of detailed data from separate algorithm executions for subsequent analysis and development of new heuristics.

Using collections of different graph classes, we investigate the effect of varying the selections of heuristics, often revealing exactly which individual algorithmic choice is responsible for particularly good or bad performance. On several benchmark collections, including a newly proposed class of difficult instances, we additionally find that our implementation performs better than the current state-of-the-art tools.

CCS Concepts: • **Mathematics of computing → Combinatorial algorithms**; *Graph theory*; • **Computing methodologies** → *Discrete space search*; • **Software and its engineering** → *Software libraries and repositories*;

Additional Key Words and Phrases: Graph canonization, graph isomorphism, generic programming

ACM Reference format:
Jakob L. Andersen and Daniel Merkle. 2020. A Generic Framework for Engineering Graph Canonization Algorithms. *J. Exp. Algorithmics* 25, 1, Article 1.2 (April 2020), 26 pages.
https://doi.org/10.1145/3356020

J. L. Andersen was affiliated with University of Vienna and Harvard Medical School when this work was done. D. Merkle was affiliated with Harvard Medical School when this work was done.
This work was supported in part by COST-Action CM1304 "Systems Chemistry," by the Independent Research Fund Denmark, Natural Sciences, grant DFF-7014-00041, and by the ELSI Origins Network (EON), which is supported by a grant from the John Templeton Foundation. The opinions expressed in this publication are those of the authors and do not necessarily reflect the views of the John Templeton Foundation. Computation for the work described in this article was supported by the DeiC National HPC Centre, SDU, as well as the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
Authors' addresses: J. L. Andersen and D. Merkle, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense, Denmark; emails: {jlandersen, daniel}@imada.sdu.dk.

1084-6654/2020/04-ART1.2 $15.00
https://doi.org/10.1145/3356020

1 INTRODUCTION

Graph canonization is the process of finding a canonical representation of a graph such that all isomorphic graphs are assigned the same representation. The graph isomorphism problem can thus be reduced to comparing such canonical representations, which is especially useful when we want to test isomorphism against a large collection of graphs (e.g., for database querying). There is rich literature on the complexity of both graph canonization and graph isomorphism. For a longer discussion, we refer to the work of McKay and Piperno [22] and simply note that for general graphs, the problems are not known to be NP-complete, and the best bound for canonization has previously been $e^{O(\sqrt{n \log n})}$ [10, 13]. For the isomorphism problem, a quasi-polynomial bound has been presented [11], and recently it seems that this result can be extended also to the canonization problem [12].

For practical graph canonization, there has also been extensive work, with several competitive tools being published in the past decades. They are all built on the same core idea of a tree search over gradually more refined partitions of the vertex set, also called the *individualization-refinement paradigm*. Their difference is thus essentially in the heuristics for traversing and pruning the search tree, and how partitions are being refined. One of the most successful tools is nauty [21], which not only finds a canonical representation but also computes the automorphism group, which during the canonization is used for pruning the search tree. Later tools, Bliss [16, 17] and Traces [22], also use this technique with the latter introducing a new way to exploit the discovered automorphisms. A related tool is Saucy [14, 18], which only performs computation of the automorphism group, for which it introduced new heuristics to discover them. Similarly there is Conauto [20], which performs isomorphism testing directly without computing a canonical form. Each new tool (and updated versions of tools) has incorporated ideas from the other tools, with further development of heuristics. However, the performance comparisons have been done between the tools in their entirety, making it exceedingly difficult to discover exactly which combination of ideas lead to better or worse performance on particular classes of graphs. As the tools are almost completely independent implementations, it is additionally hard to make a fair comparison, even if individual innovations could be isolated.

To address these problems, we have developed a generic framework for constructing variations of graph canonization algorithms, where new ideas can be implemented as separate plugins and injected into a common core algorithm. Not only does this framework solve the problem of fairly comparing heuristics, but it also significantly lowers the barrier of entry for people to test new ideas in practice. We provide implementations of a core set of heuristics, including new variations of node invariants and a memory-sensitive tree traversal algorithm. Contrary to the established tools, the framework allows for direct canonization of graphs with edge attributes, and we have developed a generalization of the widely used Weisfeiler-Leman (WL) refinement function, which can exploit such attributes.

Using established benchmark graphs, we discover interesting performance differences among combinations of heuristics, including combinations with significantly different scaling behavior and better performance than the established tools. For a recently proposed collection of six difficult graphs classes [25, 26], we perform a detailed benchmark of the effects of node invariants, which suggests why some of them are more difficult than the others.

The framework is implemented as a C++ library, called *GraphCanon*, with heavy use of generic programming [24] with influences from and compatibility with the Boost Graph library [19, 30]. It is available on GitHub [2] along with the accompanying *PermGroup* library [5] for handling permutation groups, and an interactive visualizer for algorithm runs *GraphCanon Visualizer* [3]. In Appendix B, we provide additional details of the framework, including pseudocode with direct links to the corresponding C++ code. The appendix also contains additional visualizations of search trees.

In Sections 2 and 3, we mathematically describe the whole class of individualization-refinement algorithms, although formulated to lend itself to a more direct mapping to algorithmic terms. This is used in Section 4, where we describe the generic algorithm framework where the abstract algorithm is decomposed into six categories of subprocedures. This includes a description of the interplay between the categories and brief examples of concrete algorithms implemented in published tools and for this contribution. The experimental results are presented in Section 5, and a summary with future developments is in Section 6.

2 PRELIMINARY DEFINITIONS

We denote an undirected graph as $G = (V, E)$, with V as the vertices and E as the edges. The goal is to find a canonical representation of G, and we therefore assume the vertices already to have associated IDs. For ease of notation, we assume $V = \{1, 2, \ldots, n\}$. An attributed graph is a tuple $G = (V, E, l_V, l_E)$ of a graph (V, E) and two attribution functions $l_V : V \rightarrow \Omega_V$ and $l_E : E \rightarrow \Omega_E$. We assume that the attribute sets Ω_V and Ω_E are totally ordered sets. We denote the set of all attributed graphs on n vertices as $\mathcal{G}_n$ or simply as $\mathcal{G}$. For the remainder of this contribution, we assume an attributed graph $G = (V, E, l_V, l_E)$ with n vertices is given. Note that in related works (e.g., [16, 22]), they use so-called colored graphs that are equivalent to graphs only with vertex attributes, and they do not consider edge attributes directly. We assume the set of graphs $\mathcal{G}$ is totally ordered. For example, if the graphs are represented as adjacency matrices, we can lexicographically compare the matrices.[1] For graphs with vertex and/or edge attributes, this comparison must also account for those attributes. For two graphs $G_1, G_2 \in \mathcal{G}$ with the same underlying representation, we say they are *representationally equal*, written $G_1 \overset{r}{=} G_2$. When they are not, we may say that one is *representationally smaller*, written $G_1 \overset{r}{<} G_2$.

A canonization algorithm starts with the assumption that all vertices are unordered and then incrementally introduces order. To represent these intermediary partial orders, we use the following construct. An *ordered partition* of V is a sequence $\pi = (W_1, W_2, \ldots, W_r)$ of non-empty sets of vertices that partitions V. Each of the constituent vertex sets is called a *cell* of π. For a vertex v in the j-th cell, we define $cell(v, \pi) = j$. A cell of size 1 is called a *singleton*, and if all cells are singletons, we call the partition *discrete*. If the ordered partition only has one cell (i.e., $\pi = (V)$), it is called the *unit partition*.

The set of all ordered partitions over V is denoted Π. An ordered partition π' is *at least as fine as* π, written $\pi' \preceq \pi$, when $cell(u, \pi) < cell(v, \pi)$ implies

$$cell(u, \pi') < cell(v, \pi') \qquad \forall u, v \in V.$$

In other words, π' can be obtained from π by only subdividing cells.

Ordered partitions are used to represent intermediary states of the canonization procedure, in the sense that for a partition π, the vertices of a cell W_i are said to be ordered before vertices of a cell W_j when $i < j$. The unit partition thus represents no ordering information, whereas each discrete ordered partition is a canonical order candidate.

Let S_n denote the symmetric group on the set of vertices V. For a permutation $\gamma \in S_n$, we denote the image of an element $v \in V$ as v^γ. Composition of permutations is thus written from left to right (i.e., $v^{\gamma_1\gamma_2} = (v^{\gamma_1})^{\gamma_2}$). The inverse of a permutation γ is denoted $\overline{\gamma}$. The transformation of a subset

[1] An illustrated example of comparison using adjacency lists can be found in Appendix A.

of vertices $W \subseteq V$ with $\gamma \in S_n$ is defined as

$$W^\gamma = \{w^\gamma \mid w \in W\},$$

whereas the transformation of a sequence $Q = (q_1, q_2, \ldots, q_k) \in V^k$ is

$$Q^\gamma = (q_1^\gamma, q_2^\gamma, \ldots, q_k^\gamma).$$

Similarly, we extend this scheme to combinations of these structures, all derived from V, which in particular means we can permute ordered partitions and (representations of) graphs. An example of permuting representations can be found in Appendix A.

We interpret a discrete ordered partition π as a permutation in S_n, which maps each cell index to its contained vertex. In other words, if $cell(v, \pi) = j$, then $j^\pi = v$. The inverse permutation $\overline{\pi}$ thus maps vertices to their cell indices. Note that if we use a discrete ordered partition π to represent a candidate for the canonical order, we then have π as a permutation from the candidate canonical indices to the indices in the input graph. Permuting the input graph with the inverse permutation $G^{\overline{\pi}}$ thus gives us the actual candidate for the canonical representation.

Two graphs $G_1, G_2 \in \mathcal{G}$ are isomorphic, denoted $G_1 \cong G_2$ if there exists a permutation $\gamma \in S_n$ such that $G_1^\gamma \overset{r}{=} G_2$. The permutation γ is then called an *isomorphism*, and if G_1 and G_2 refer to the same object, it is further called an *automorphism*. The set of all automorphisms of a graph G, the automorphism group, is denoted $\mathrm{Aut}(G)$.

A canonization algorithm can be seen as a function on graphs, $C : \mathcal{G} \to \mathcal{G}$, with the following properties (C1 and C2 in McKay and Piperno [22]):

$$C(G) \cong G$$
$$C(G^\gamma) \overset{r}{=} C(G) \qquad \forall \gamma \in S_n.$$

In other words, it returns a graph isomorphic to its input, and it is invariant with respect to permutations of its input. The second property is also called *isomorphism invariance*, and we will require this property for most of the procedures we describe in the following sections.

3 THE ABSTRACT ALGORITHM

For a high-level description of the individualization-refinement approach with proofs of correctness we refer to the work of McKay and Piperno [22]. The following description follows the same principles, but we opt for a description that more easily maps to the generic implementation presented in the next section.

The individualization-refinement approach is a tree search starting from the unit partition in the root with each leaf of the search tree corresponding to a discrete ordered partition. The canonical form then corresponds to a minimum leaf, where *minimum* is defined in conjunction by the graph representation comparator $\overset{r}{<}$, and by so-called *node invariants* that will be discussed in the end of this section.

Instead of comparing the vertex attributes in the leaves using $\overset{r}{<}$, we can instead exploit them from the beginning by starting with a specific ordered partition instead of the unit partition. The *initial partition* is then created by partitioning the vertices by their attribute and ordering the cells by the attribute. In other words, the initial partition is $\pi_0 = (V_1, V_2, \ldots, V_k)$ with the property that for all $u \in V_i$ and $v \in V_j$, if $l_V(u) = l_V(v)$, then $i = j$, and otherwise $l_V(u) < l_V(v)$ implies $i < j$.

The algorithm is defined using several abstract functions. The first one is a *refinement function* $R : \mathcal{G} \times \Pi \to \Pi$ with the following properties [22]:

$$R(G, \pi) \preceq \pi$$
$$R(G^\gamma, \pi^\gamma) = R(G, \pi)^\gamma \qquad \gamma \in S_n.$$

In other words, it produces a partition that is finer or equal to its input, and it is isomorphism invariant.

When the refinement function produces non-discrete partitions, we must decide on a cell where we will artificially introduce cell splits. For this, we need a *target cell selector* $T : \mathcal{G} \times \Pi \to 2^V$ that for a non-discrete partition returns one of the non-singleton cells. This function must also be isomorphism invariant—that is, $T(G^\gamma, \pi^\gamma) = T(G, \pi)^\gamma$ for all $\gamma \in S_n$.

The introduction of artificial splits is done by *vertex individualization*. For an ordered partition π with a non-singleton cell W and a vertex $v \in W$, we define $\pi \downarrow v$ as the ordered partition where W is replaced by two cells $\{v\}$ and $W \backslash \{v\}$. In other words, $\pi \downarrow v$ is the partition strictly finer than π obtained by individualizing v to the left (or right) from the rest of its cell. Whether individualization is done to the left or to the right from the remaining vertices is an implementation detail, as long as the same choice is used consistently.

Search tree. A search tree can now be formally defined as follows. Each node τ of the tree is identified by a sequence of vertices, and it implicitly defines an associated ordered partition π_τ defined in the following manner. Let π_0 be the initial ordered partition constructed from vertex attributes as described earlier. The root of the search tree is then the empty sequence $\tau_{root} = ()$, with the associated ordered partition $\pi_{\tau_{root}} = R(G, \pi_0)$.

For a node $\tau = (v_1, v_2, \ldots, v_k)$, if π_τ is discrete, then τ is a leaf. Otherwise, let $W = T(G, \pi_\tau)$ be the target cell selected by T. For each vertex $w \in W$, there is a child node $\tau_{child} = (v_1, v_2, \ldots, v_k, w)$, with the associated ordered partition $\pi_{\tau_{child}} = R(G, \pi_\tau \downarrow w)$. In other words, for each child, we individualize a different vertex of the target cell and then perform refinement on the partition. An example of a search tree with ordered partitions can be found in Figure 1. In the next two sections, we describe techniques for pruning the search tree. Note that although this may leave otherwise internal nodes with no children, we still reserve the term *leaf* to the original leaf nodes (i.e., nodes representing discrete partitions) and let $L(G)$ denote all such leaf nodes for a search tree based on G.

Pruning with automorphisms. Let $\tau_a, \tau_b \in L(G)$ be distinct leaves of the search tree, for which the permuted graphs are representationally equal. In other words, with $\alpha = \overline{\pi_{\tau_a}}$ and $\beta = \overline{\pi_{\tau_b}}$, then $G^\alpha \overset{r}{=} G^\beta$. Permuting both sides with $\overline{\beta}$ gives $G^{\alpha\overline{\beta}} \overset{r}{=} G^{\beta\overline{\beta}} = G$, meaning that $\alpha\overline{\beta}$ is an automorphism of G. During the tree traversal, when finding new leaves that give representations equal to our currently best leaf, we can derive an automorphism. We call this an *explicit automorphism*, and the complete automorphism group can be computed by considering all such pairs of leaf nodes [22]. Sometimes it is possible to deduce automorphisms from internal nodes of the search tree. We call automorphisms found in this manner *implicit automorphisms* [21].

During the canonization procedure, let $\tau = (v_1, v_2, \ldots, v_k)$ be an internal node of the search tree and $u, v \in T(G, \pi_\tau)$ two vertices in the target cell for this node. Further, let $\gamma \in \mathrm{Aut}(G)$ be some known automorphism that fixes all vertices individualized on the path from the root to τ but moves u to v. In other words, $v_i^\gamma = v_i$ for $1 \le i \le k$, and $u^\gamma = v$. As u and v are equivalent under γ, the two subtrees rooted at $(v_1, v_2, \ldots, v_k, u)$ and $(v_1, v_2, \ldots, v_k, v)$ will be isomorphic, and we can safely skip traversal of one of them (Operation P_C in McKay and Piperno [22]). This is illustrated in Figure 1, where grayed-out tree nodes are skipped due to use of automorphisms. An example with automorphisms annotated can be found later in Figure 6.

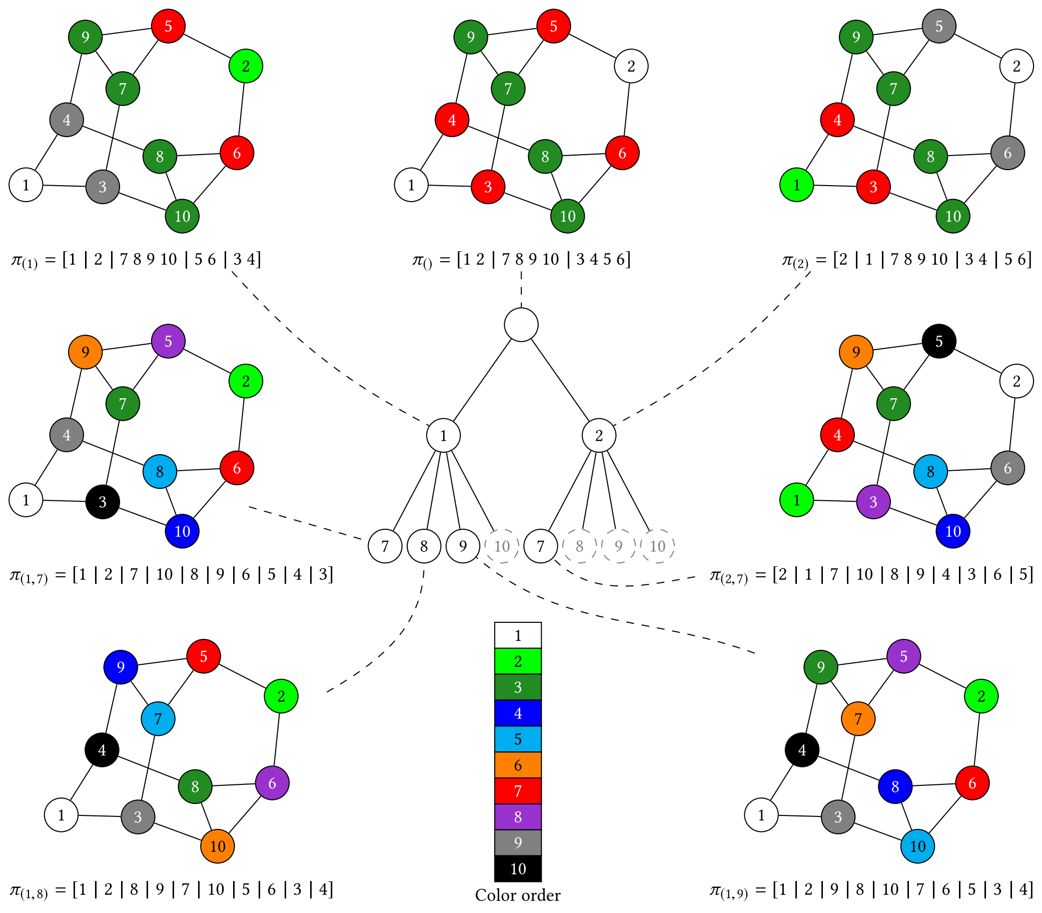

π(1) = [1 | 2 | 7 8 9 10 | 5 6 | 3 4]
π() = [1 2 | 7 8 9 10 | 3 4 5 6]
π(2) = [2 | 1 | 7 8 9 10 | 3 4 | 5 6]
π(1,7) = [1 | 2 | 7 | 10 | 8 | 9 | 6 | 5 | 4 | 3]
π(2,7) = [2 | 1 | 7 | 10 | 8 | 9 | 4 | 3 | 6 | 5]
π(1,8) = [1 | 2 | 8 | 9 | 7 | 10 | 5 | 6 | 3 | 4]
π(1,9) = [1 | 2 | 9 | 8 | 10 | 7 | 6 | 5 | 3 | 4]
Color order

Fig. 1. A search tree starting with the refinement of the unit partition in the root. The refinement function used is the WL-1 algorithm (see Section 4.3), the target cell selector is selecting the first non-singleton cell, and no node invariants are used. Each node in the tree represents a sequence of vertex individualizations, where the latest vertex being individualized is shown in the nodes. For most tree nodes, the corresponding partition is shown along with the colored graph it represents. In the colored graphs, the vertices are labeled with the vertex indices from the input graph and colored with "numbers" corresponding to the potential canonical vertex indices. Note that the colored graphs in the leaves of the left half of the tree (the children of $\tau = (1)$) are all isomorphic. This is also true among the children in the right half of the tree (the children of $\tau = (2)$). However, between the halves of the tree, the graphs are not isomorphic. The dashed gray nodes correspond to nodes pruned from automorphism discovery, when depth-first traversal of the tree is used. The example is heavily inspired by Figure 3 in the work of Piperno [27], although here using different functions for refinement and target cell selection.

Pruning with node invariants. During the construction of a tree node, it is often possible to extract isomorphism-invariant information. The path from the root to a leaf thus has a sequence of such extracted information, which again is isomorphism invariant. We then redefine the canonical form to be the one with the lexicographically smallest of such information sequences.

Formally, let $\mathcal{T}$ denote the set of all search tree nodes and Ω an arbitrary totally ordered set. An *invariant function* $\phi : \mathcal{G} \times \mathcal{T} \to \Omega$ then assigns a value to each tree node. The function must be isomorphism invariant (i.e., $\phi(G^\gamma, \tau^\gamma) = \phi(G, \tau)$). For convenience, we define the *path-invariant* *function* $\vec{\phi} : \mathcal{G} \times \mathcal{T} \to \Omega^*$ as follows. For a tree node $\tau = (v_1, v_2, \ldots, v_k)$, let its ancestors here be denoted as $\tau_i = (v_1, v_2, \ldots, v_i)$ for $i = 0, 1, \ldots k-1$ (e.g., with the root being $\tau_0 = ()$). We then define the path invariant of τ as the sequence of node invariants from the root downward:

$$\vec{\phi}(G, \tau) = (\phi(G, \tau_0), \phi(G, \tau_1), \ldots, \phi(G, \tau_k)).$$

We compare such sequences lexicographically.

Canonization. We can now finally define the canonical form constructed by the abstract algorithm. Recall that the associated ordered partition of each leaf τ is a discrete partition π_τ, which represents a candidate canonical ordering of the vertices. Specifically, the permutation $\overline{\pi_\tau}$ maps the input vertices to their new index, and the graph $G^{\overline{\pi_\tau}}$ is thus a candidate for the canonical form. We first restrict our choice to the leaves that obtain the minimum path invariant, and then define the canonical form $C(G)$ to be the permuted graph indicated by the leaf node with the representationally smallest permuted graph. In other words,

$$\begin{aligned} \phi^* &= \min_{\tau \in L(G)} \vec{\phi}(\tau) \\ \tau_{canon} &= \arg\min_{\tau \in L(G)} \left\{ G^{\overline{\pi_\tau}} \middle| \vec{\phi}(\tau) = \phi^* \right\}. \\ C(G) &= G^{\overline{\pi_{\tau_{canon}}}} \end{aligned}$$

Note that pruning with node invariants is entirely optional but if done may alter which leaf is selected for the canonical form.

4 A GENERIC ALGORITHM FRAMEWORK

From the abstract canonization algorithm, we see that each concrete algorithm is defined by specific choices of subprocedures for the six categories in the following table:

Extension Category	Abstract Function	Section
Tree traversal	—	4.1
Target cell selection	$T : \mathcal{G} \times \Pi \to 2^V$	4.2
Refinement	$R : \mathcal{G} \times \Pi \to \Pi$	4.3
Pruning with automorphisms	—	4.4
Detection of implicit automorphisms	—	4.5
Node invariants	$\phi : \mathcal{G} \times \mathcal{T} \to \Omega$	4.6

The goal of the present framework is to provide an implementation of functionality common to all algorithms, as well as provide a facility for attaching extensions for the six categories. Note that in a given algorithm, there must be exactly one extension for target cell selection and tree traversal, but for the remaining categories it can be beneficial to use multiple independent extensions in conjunction.

Using generic programming [24], we have designed a single common extension infrastructure, based on the idea of a *visitor*, similar to those used in the Boost Graph library [19, 30]. In general, a visitor object can be seen as a set of callbacks that the main algorithm will invoke at various points during its execution. The callbacks in a visitor thus cooperate to implement an extension. A *concept* is a collection of requirements for a type, and the framework here defines a specific *GraphCanonVisitor* concept that details which callbacks must be implemented.[2] The core canonization procedure is given a visitor object vis, which for example must implement the following methods:

[2]The details of the *GraphCanonVisitor* concept can be found in Appendix B.1.

- `vis.isomorphicLeaf`(τ)
  Called when a new candidate leaf τ has the same representation as the current best candidate.
- `vis.implicitAutomorphism`(γ)
  Called from visitors when they discover a new automorphism γ.
- `vis.beforeDescend`(τ)
  Called by tree traversal algorithms before deciding which child to create next of a node τ.

The first two callbacks can for example be used to record automorphisms, which be used in the third callback to prune children.[3]

A visitor must also specify two types of data structures that it can use to store auxiliary data: one that will be instantiated per search tree and one instantiated for each node in each search tree.

We provide a compound visitor, which for a sequence of individual visitors aggregates the associated data structures and dispatches method calls to all of the contained visitors. The compound visitor enforces that exactly one visitor has implemented a tree traversal algorithm and exactly one has a target cell selector. For the following sections, we assume the object visis such a compound visitor aggregating the sequence $(\mathsf{vis}_1, \mathsf{vis}_2, \ldots, \mathsf{vis}_t)$ of visitors. As the framework is implemented in C++ using templating, the function calls involving visitors are all resolved statically, giving the compiler the opportunity to perform appropriate function inlining.

The common functionality of the framework further consists of data structures for tree nodes and ordered partitions, along with methods for creation and destruction of tree nodes, including vertex individualization and invocation of target cell selection and refinement through the visitors. The framework methods will additionally take care of comparing leaves by their permuted graphs, including comparison of vertex/edge attributes if relevant, thus ensuring a functioning algorithm even if no visitors are given.[4]

In the following sections, we provide further details for each of the six extension categories. Outside those categories, we provide two additional visitors: a debug visitor for collecting detailed logs and a stats visitor (e.g., for counting the number of tree nodes created and even for creating an annotated visualization of the search tree).[5]

4.1 Tree Traversal and Automatic Garbage Collection

Most of the published algorithms and tools, including nauty [21, 22] and Bliss [16, 17], use depth-first traversal of the search tree. Bliss notably exploits this traversal order to use a more efficient data structure for ordered partitions. The tool Traces [22] uses a different traversal scheme that combines a breadth-first traversal with so-called experimental paths to find leaves early. As noted in the work of Neuen and Schweitzer [25], this means that Traces can consume much more memory than tools using depth-first traversal.

The framework directly allows for arbitrary tree traversal algorithms to be used by defining appropriate visitors. The lifetime of tree nodes is managed using reference counting, where each node has a owning reference to its parent and the parent has non-owning references to its children. Each visitor is thus responsible for keeping owning references to nodes in which it is interested. The creation of new children is done through a framework method, whereas discovered leaf nodes must be reported through another framework method that handles comparison of permuted graphs and potentially updating the current best leaf. To facilitate pruning, a specific visitor method should be invoked before deciding which child node to create next.

[3] Pseudocode for an automorphism pruning visitor can be found in Algorithm 7 in the appendix.
[4] Pseudocode for the framework methods can be found in Figure 8 in the appendix.
[5] Examples of search tree visualizations can be found in Appendix C.

We provide an implementation of the classical depth-first traversal (DFS)[6] and a traversal similar to Traces (BFSExp). We have also developed a memory-sensitive hybrid of those two traversals (BFSExpM), which trades time for guaranteed memory usage. Based on a given memory limit, it uses BFSExp when the number of tree nodes is low and uses DFS when above the limit. It may therefore switch back to BFSExp if a sufficient amount of the search tree is deallocated. With the provided debug visitor, it is directly possible to investigate how the number of currently allocated tree nodes develops through the course of the algorithm.

4.2 Target Cell Selection

A large variety of target cell selectors are available in Bliss, Traces, and nauty, with the simplest selecting the first either smallest or largest cell. A property used in more advanced target cell selectors is the following [16]: for two cells $U, W \in \pi$, we say that U is *non-uniformly joined* to W if for all vertices $u \in U$ there are two vertices $w_1, w_2 \in W$ such that $(u, v_1) \in E$ and $(u, v_2) \notin E$. In other words, all vertices of U have both neighbors and non-neighbors in W.

In the present framework, the target cell selection is done during construction of each tree node using a dedicated visitor method. Exactly one visitor must indicate that it implemented this method.

We provide three target cell selectors: select the first non-singleton cell (F), select the first largest cell (FL), and select the first largest cell that is non-uniformly joined to the most cells (FLM).

4.3 Refinement

The basic refinement function used in most tools is the 1-dimensional Weisfeiler-Leman algorithm (WL-1) [31], which iteratively separates vertices in a cell depending on their degree with respect to other cells (see Algorithm 1). This algorithm can be generalized to higher dimensions, and Traces implements the 2-dimensional variant [27]. Besides the WL algorithm, the tool nauty also includes a selection of additional refinement functions [22].

Refinement is invoked during the construction of a tree node through the refinement visitor method. Multiple visitors may do refinement, so the formal refinement function R is derived from the composition $R_t \circ \cdots \circ R_2 \circ R_1$, where R_i is the refinement function implemented by visitor vis_i. The compound visitor coordinates the invocation using returned status codes—for example, indicating whether any refinement was performed or if it was aborted due to node-invariant pruning. To support calculation of node invariants and discovery of implicit automorphisms, there are multiple visitor methods that refinement algorithms, especially WL-1, can call—the simplest being the method called for each cell split performed.

We provide the WL-1 algorithm for refinement that, based on the observations of Junttila and Kaski [16], uses custom implementations of counting sort and array partitioning to perform fast sorting for low-degree cases. After WL-1 has run, an ordered partition π is said to be *equitable*, meaning it fulfills the following property: every pair of vertices in the same cell has the same number of edges to every cell (including itself). In other words, when $d : V \times 2^V \to \mathbb{N}_0$ is the degree function counting the number of edges between a given single vertex and a given subset of vertices, then the property is

$$\forall X, W \in \pi, \forall u, v \in X : d(u, W) = d(v, W).$$

Some techniques requires this property to hold to be valid (e.g., the scheme for detecting implicit automorphisms described in the following and in Lemma 2–25 in the work of McKay [21]).

The framework directly allows for canonization of graphs with edge attributes by comparing them in the permuted graphs in the leaves of the search tree. Such attributes can potentially be

[6]Pseudocode for the implementation of DFS is shown in Algorithm 6 in the appendix.

exploited already during refinement, so we have generalized our WL-1 implementation in the following manner. The algorithm refines a cell by distinguishing the vertices by their degree to other cells. In other words, for a cell to be refined $X \subseteq V$ and a cell to refine with $W \subseteq V$, the degrees $d(x, W)$ for each $x \in X$ are considered. The generalization to exploit edge attributes is achieved essentially by abstracting the degree function. We generalize it to return a map from each possible edge attribute to the number of edges with that attribute. In other words, instead of $d : V \times 2^V \rightarrow \mathbb{N}_0$, we change the signature to $d : V \times 2^V \rightarrow (\Omega_E \rightarrow \mathbb{N}_0)$. Such mappings are totally ordered, as we assume Ω_E to be totally ordered.

In practice, we delegate the edge handling to a given `edgeHandler` object such that the user can decide the most efficient way to count edges and sort vertices. A high-level description of the WL-1 algorithm without edge attribute handling is shown in Algorithm 1. In this formulation, the delegation to the `edgeHandler` object happens in addition to lines 4, 7, 9, and 10.

ALGORITHM 1: WL-1 Refinement ▷C++

```
   Input: $\pi$, a reference to an ordered partition
   Input: $Q$, a non-empty FIFO queue of cells of $\pi$
   Data: C , an associative array $V \rightarrow \mathbb{N}_0$ for counting neighbors
1  while $Q$ not empty and $\pi$ not discrete do
2    $W \leftarrow$Q.popFront()
3    foreach $v \in V$ do
4      C[$v$] $\leftarrow 0$
5    foreach vertex $w \in W$ do
6      foreach edge $(w, x) \in E$ do
7        C[$x$] $\leftarrow$ C[$x$] $+ 1$
8    foreach cell $X \in \pi$ do
9      Sort the vertices of $X$ according to the counters in $C$.
10     Split $X$ into cells $X_1, X_2, \ldots, X_k$ according to common values of $C$.
11     Report each cell split using vis.newCell.
12     if $X \in Q$ then
13       $Q$.remove($X$)
14       foreach $X_i \in \{X_1, X_2, \ldots, X_k\}$ do
15         $Q$.pushBack($X_i$)
16     else
17       $X_{max} \leftarrow$ the first cell of $X_1, X_2, \ldots, X_k$ of maximum cardinality
18       foreach $X_i \in \{X_1, X_2, \ldots, X_k\} \backslash \{X_{max}\}$ do
19         $Q$.pushBack($X_i$)
```

Note that even though an input graph $G = (V, E, l_E)$ has edge labels that should be respected by the canonization algorithm, it is still be possible to use the standard WL-1 algorithm that operates on the underlying graph $G' = (V, E)$, as edge labels are taken into account when comparing permuted graphs stemming from the leaves of the search tree. However, this of course means that the resulting ordered partitions are only equitable with respect to G' and not the input graph G, meaning that Lemma 2–25 [21] cannot be applied.

As an example of using edge attributes during refinement, we can consider graphs modeling molecules [7] where each edge has an attribute indicating which one of four possible chemical bonds the edge models. The degree function can thus return 4-vectors of integers and vertex sorting can be done with radix sort. For example, this scheme is used in the MØD software package [4, 7, 9]. A very recent publication [28] describes a similar extension of the Traces tool, although specialized to weighted graphs.

4.4 Pruning with Automorphisms

Let A be the list of discovered automorphisms at some stage in the algorithm. As automorphisms are closed under composition, we can consider the subgroup $\Phi \leq \mathrm{Aut}(G)$ generated from A. For a tree node $\tau = (v_1, v_2, \ldots, v_k)$, we are then interested in pruning with automorphisms that fix all individualized vertices on the path down to τ. In other words, we consider the stabilizer $\mathrm{Stab}_\Phi(\tau) = \{\gamma \in \Phi \mid v_i^\gamma = v_i, 1 \leq i \leq k\}$. Such a stabilizer can be computed using methods based on the Schreier-Sims algorithm [29]. As the full computation can be expensive, there are randomized Monte Carlo versions that may compute only a subgroup of the stabilizer. For example, this is done in Traces and newer versions of nauty [22]. Another approximation that is even less computationally intensive is used in Bliss and earlier versions of nauty. In this scheme, which we here call the *basic scheme*, there is no composition of permutations during computation. Instead, only the subset $A_\tau = \{\gamma \in A \mid v_i^\gamma = v_i, 1 \leq i \leq k\}$ is considered. In other words, the stored automorphisms are directly filtered. However, in previous tools, only a fixed number of permutations are stored at a time to conserve memory [16, 21]. In Section 5, we provide an illustration of how a Schreier scheme may increase pruning power.

The framework provides explicit automorphisms to the visitors, and they may report implicit automorphisms to each other. The actual pruning is done in the visitor method that the tree traversal visitor is expected to invoke before deciding which child to create next.

We have implemented a visitor for automorphism pruning which itself is parameterized such that it can work with different implementations of permutations, groups, and stabilizer chains. For each tree node, the visitor maintains the orbit partition of the target cell, using a union-find data structure. In the present version, we provide two parameterizations: (1) the basic scheme, which maintains A_τ in each node, similar to the strategy in Bliss and earlier versions of nauty but without a limit on the number of stored automorphisms, and (2) a deterministic Schreier scheme, which computes the full stabilizer in each tree node.[7] It should be noted that there are many variations of the Schreier-Sims algorithm (e.g., see [29]), each with different space, time, and accuracy characteristics, and each of them can be adapted in multiple ways to design an automorphism pruning scheme.

4.5 Detecting Implicit Automorphisms

There are several known methods for finding additional automorphisms during the tree search. For example, Lemma 2–25 in the work of McKay [21] describe three cases where all refinements of certain ordered partitions lead to isomorphic leaf nodes. However, note that the lemma only applies if the partition is equitable with respect to the input graph (see also Section 4.3). The simplest, and most common, of those cases in this lemma is where the partition only has singleton cells or cells of size 2.[8] Saucy [14, 18] introduced another heuristic for finding primarily sparse automorphisms, where for each non-leaf node a guess for an automorphism is made. Traces [22] reportedly generalizes this heuristic, although the details have not been described. Traces also has heuristics for finding automorphisms when all vertices in non-singleton cells have certain degrees, but it is not clear what those heuristics are.

We have implemented two visitors in this category: one for the most common case of Lemma 2–25 [21], described earlier, and one for refining cells with only degree 1 vertices and reporting the implicit automorphisms discovered in the process.

[7]Pseudocode for the generic automorphism pruner visitor can be found in Algorithm 7 in the appendix.
[8]Pseudocode for the simplest case of Lemma 2-25 [21] can be found in Algorithm 8 in the appendix.

4.6 Pruning with Node Invariants

In the abstract algorithm, we used a single node-invariant function, although in practice we may want to use multiple functions. For example, one source of invariant data is from the sequence of cell splits performed by the refinement function, introduced in Traces [22]. Another important example is the *partial leaf* invariant introduced in Bliss [16], calculated when new singleton cells arise in the refinement procedure. Third, the WL-1 algorithm in general computes many different counts of edges, and this sequence of numbers can also be used as an invariant. Importantly, both Bliss and nauty use hashing during node-invariant computation, which may diminish the pruning power when collisions occur.

In the framework, the calculation of node invariants and pruning with node invariants can be implemented entirely as visitors, where separate invariant functions are implemented as separate visitors. Each of them produces a sequence of invariant values that are then interleaved to form the overall node invariant. This interleaving is coordinated by another visitor provided in the framework for ensuring correct pruning but also to minimize the implementation overhead for new invariants. Let Ω_i be the domain of invariant values produced by vis_i, then the overall node invariant values for a given tree node is a sequence of pairs $(\langle i_1, \omega_1\rangle, \langle i_2, \omega_2\rangle, \ldots, \langle i_k, \omega_k\rangle)$, where i_j is a visitor index and $\omega_j \in \Omega_{i_j}$. The first component of each pair is stored in the coordinating visitor, whereas the second component is stored in the corresponding visitor. The coordinating visitor handles invalidation of the current best leaf when a better path invariant is found, and handles pruning of children of nodes that has already been created, but where a better invariant was found later.

We implement three invariants: the cell splitting trace introduced in Traces (T), a trace of values for the quotient graph also introduced in Traces [22] (Q), and the partial leaf invariant introduced in Bliss (PL). Note that we do not hash the information in any of these visitors.

5 EXPERIMENTAL RESULTS

Although the framework is capable of handling both vertex and edge attributes, unfortunately there are no comprehensive collections with such graphs. We have therefore benchmarked our implementation using both the collections of unattributed graphs from McKay and Piperno [23] and a proposed collection of difficult graphs, `cfi-rigid` [25, 26]. All executions were repeated five times with random permutations of the input graph, always with a maximum of 8 GB of memory and a 1,000-second time limit. Of the repetitions that succeeded, we plot the average time spent, as well as markers if at least one execution ran out of memory (OOM) or out of time (OOT). We have also recorded the number of tree nodes created, but as the elapsed time to a large degree is proportional to the number of nodes, we show time plots unless specified otherwise. The full set of benchmark results is available online [1] with an interactive visualizer [6]. For comparison of absolute performance, we have also benchmarked the default configurations of Bliss v0.73, as well as v26r10 of nauty (dense and sparse) and Traces. All experiments were run on compute nodes with two Intel Xeon E5-2695v2 CPUs (48 cores), using in total approximately 12,000 compute node hours. First, we investigate the effect of the tree traversal algorithm and target cell selector, and second, we focus on the `cfi-rigid` collections where we also investigate the effect of different subsets of node invariants. Third, we illustrate how the BFSExpM traversal provides a memory-safe alternative to BFSExp at the expense of time, and fourth, we illustrate the effect of using the Schreier-Sims-based pruning scheme compared to the basic pruning scheme.

Tree traversal and target cell selector. For all graph collections from McKay and Piperno [23], we benchmarked the set $\{\text{BFSExp}, \text{DFS}\} \times \{\text{F}, \text{FL}, \text{FLM}\}$ of algorithm configurations, with all node invariants enabled and with the basic automorphism pruning scheme. Overall, we found that no

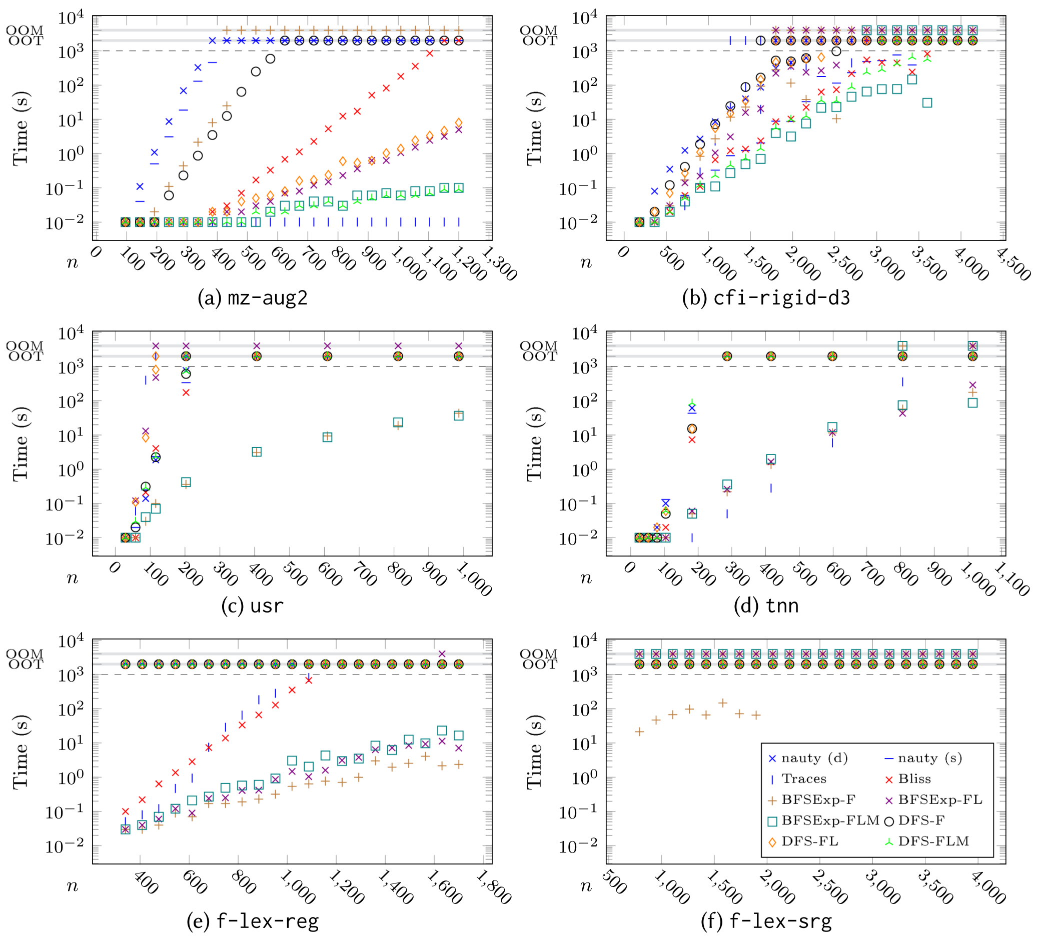


Fig. 2. Selected benchmark results for all combinations {BFSExp, DFS} × {F, FL, FLM}, with all node invariants enabled.

single configuration is the best performing on all instances, although BFSExp-FLM is often performing well. In the following, we focus on a selection of graph collections, where we find that different subsets of algorithm configurations have significantly different performance behaviors, and where some perform significantly better than the established tools.

On several of the (augmented) Miyazaki graphs, we see that the performance largely is determined by the target cell selector. Figure 2(a) shows the result for the `mz-aug2` collection, where the three pairs of F, FL, and FLM configurations have widely different performance. From the number of tree nodes explored (see [2]), we see that the FLM target cell selector scales similar to Traces (subexponential), whereas F and FL both have exponential behavior similar to Bliss and the two nauty modes. This behavior we also see in the `cmz` collection, whereas for `mz-aug`, only the F configurations scale exponentially. In both `mz-aug` and `mz` , the BFSExp-FLM configuration, for $n > 400$, will hit the memory limit for some executions but not all. This can be attributed to the automorphism pruning scheme, which is illustrated later.

On other collections, such as the non-disjoint union of tripartite graphs (`tnn`, Figure 2(d)), the algorithm configurations are separated by the tree traversal algorithms. For `tnn` , the performance

Table 1. Summary of Results for the `cfi-rigid` Collections

Col.	Group	Reduction	Best Algorithm	Invariants Matter	FLM Sep.	Max. Solved n
`d3`	D_3	—	BFSExp-FLM	Yes (any)	Yes	3,600
`z3`	$\mathbb{Z}_3$	—	BFSExp-FLM	Yes (any)	Yes	3,780
`z2`	$\mathbb{Z}_2$	—	Bliss, nauty (s)	Yes (any)	No	2,992
`r2`	$\mathbb{Z}_2$	R^*	Bliss, nauty (s)	No	No	1,584
`s2`	$\mathbb{Z}_2$	B^*	FLM, Bliss, nauty (s)	Yes (PL or Q)	No	2,496
`t2`	$\mathbb{Z}_2$	$R^* \circ B^*$	FLM, Bliss, nauty (s)	Yes (PL or Q)	Yes	1,056

The right-most column is the largest instance that any of the configurations or the tools solved.

of the BFSExp configurations is similar to Traces (which is also breadth first based), whereas for the union of strongly regular graphs (`usr`, Figure 2(c)), the BFSExp-F and BFSExp-FLM configurations perform distinctly better than all other algorithms. Surprisingly, the FL counterpart is one of the worst-performing algorithms on the same collection.

The original collection of product graphs `f-lex` contains two types of graphs where the algorithms perform differently, so we have split it into the two groups, `f-lex-reg` and `f-lex-srg`, shown in Figure 2(e) and (f). For `f-lex-reg`, we again see a separation by the tree traversal algorithm, with the DFS configurations not being able to solve any instances. However, Bliss also uses DFS but still solves many of the instances. For the `f-lex-srg` part of the collection, only `BFSExp-F` of all algorithms solve instances, although only for some executions.

The `cfi-rigid` *collections.* This package of six collections of graphs [26] was recently proposed [25] explicitly as difficult instances for graph isomorphism, and by construction they have very little symmetry. Each collection (Table 1) is constructed using a group: the dihedral group on three points (D_3), or the cyclic group on two or three points ($\mathbb{Z}_2$, $\mathbb{Z}_3$). For $\mathbb{Z}_2$ there are three further variations where the instances have gone through either a single or both of two reduction techniques (here denoted as R^*, B^*, and $R^* \circ B^*$). We have benchmarked all combinations, including subsets of node invariants, on all instances.—that is, the 48 combinations $\{\text{BFSExp}, \text{DFS}\} \times \{\text{F}, \text{FL}, \text{FLM}\} \times 2^{\{\text{PL}, \text{Q}, \text{T}\}}$. For all of them, we have used only the basic automorphism pruning scheme, as very few automorphisms are expected.

For the four $\mathbb{Z}_2$-based collections, Bliss and nauty (sparse) perform well, although both FLM configurations with all invariants have similar performance on `s2` and `t2`. On the two other collections, `d3` and `z3`, the BFSExp-FLM configuration with all invariants is the best-performing algorithm, with the corresponding DFS-FLM configuration slightly behind. However, there is a significant separation up to F and FL configurations (Figure 2(b)). We do not see this separation in the plain $\mathbb{Z}_2$-based collection (`z2`) or in those with just one reduction applied, but interestingly the separation occurs when both reductions are applied at the same time, `t2`.

In the investigation of the effect of node invariants, we first of all found that the relative effect on performance is independent of the tree traversal and target cell selector within the same graph collection. For `d3`, `z3`, and `z2` (i.e., the collections without reductions), it improves performance when enabling any one invariant. However, the effect of enabling additional invariants is minor or non-existent. Intriguingly, for `r2` where the R^* reduction was used to create the instances, the node invariants do not seem to have any effect at all. When the other reduction, B^*, has been applied (`s2`), we find that the cell splitting invariant (T) has no effect, but either PL, Q, or both have the same improving effect. Surprisingly, we also see that pattern of effect on `t2` , which has undergone both reductions. It is our hope that future in-depth studies on these graphs and node invariants may lead to new insights, both for developing better invariants and potentially for creating even more difficult benchmark graphs.

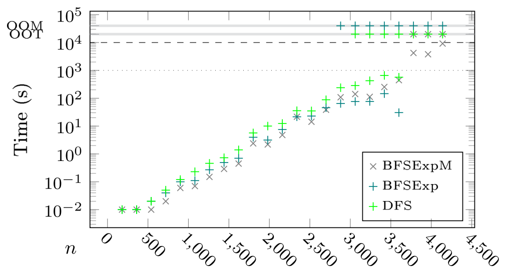


Fig. 3. Comparison of tree traversal algorithms on `cfi-rigid-d3` using FLM for target cell selection and all three node invariants.

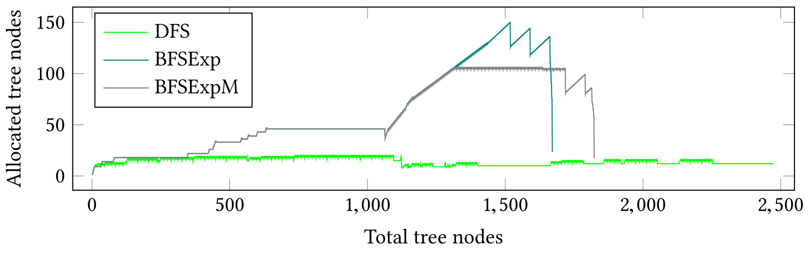
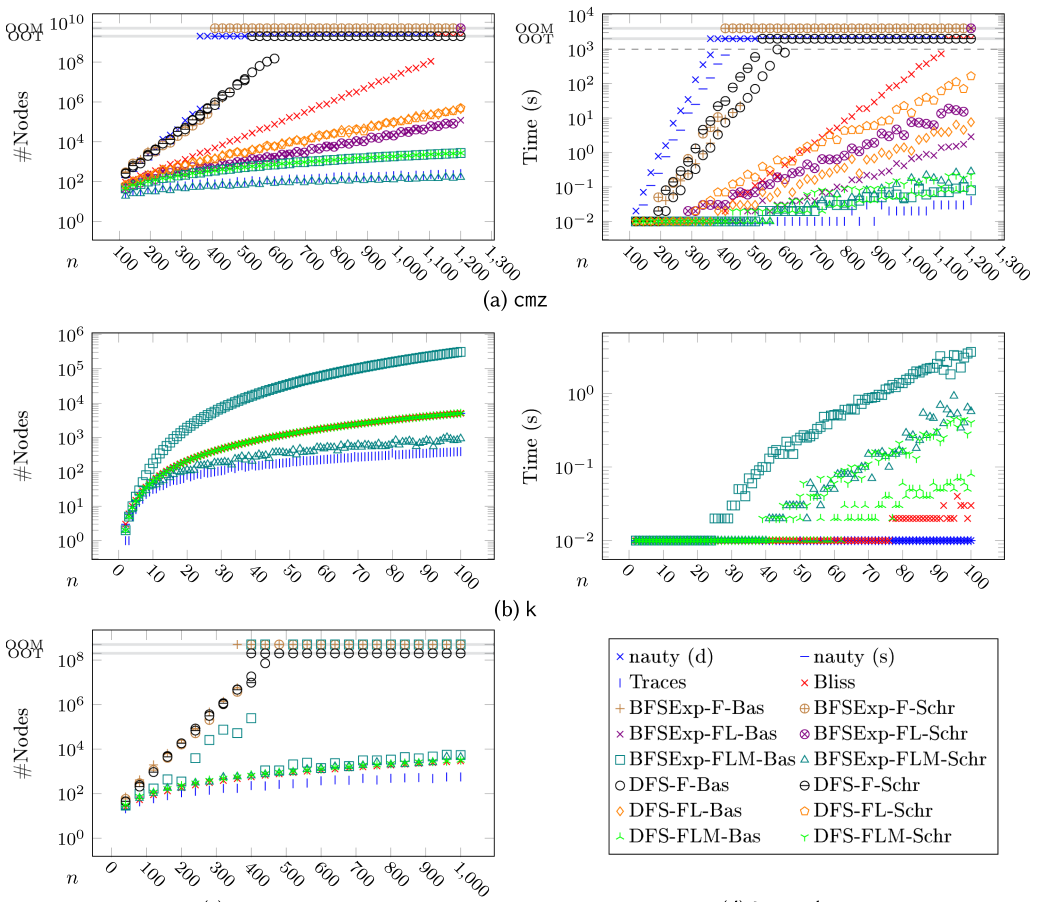


Fig. 4. Trace of how many tree nodes were allocated every time a new tree node was created. We clearly see BFSExpM hitting the limit of 104 tree nodes. As a large amount of nodes is later deallocated, it switches back from DFS into BFSExp mode. For this choice of memory bound, we see a clear trade-off between memory and time, as BFSExpM finishes later than BFSExp. However, it is still faster than DFS.

Memory-sensitive BFSExp. In some contexts it is highly undesirable to run out of memory, and we have therefore developed the BFSExpM tree traversal as described earlier. We have tested it using FLM as the target cell selector on `cfi-rigid-d3` with a limit to ensure the whole process does not hit the 8-GB hard-cap. Varying the memory limit (down to 1 GB) did not result in different performances. In Figure 3, a comparison is shown with the BFSExp and DFS configurations. We clearly see that for the instance sizes where BFSExp goes out of memory on some executions, the BFSExpM configuration increases in average time spent. Notably, it still performs better than DFS, thereby being a viable alternative when a memory limit must be honored. For this experiment, we let all executions run for 10,000 seconds, and we see that BFSExpM is the only configuration to solve the largest instance, although with some executions exceeding the time limit.

A more detailed view of how the number of allocated tree nodes develops during a single run can be obtained using the provided debug visitor. It keeps track of the number of tree nodes allocated, and we can thus immediately visualize how the number of allocated nodes develops. As an example, we illustrate this for a relatively small graph (`cfi-rigid-d3-1260-04-2`, 1,260 vertices). We ran the same input permutation with DFS, BFSExp, and BFSExpM limited to 2 MB (Figure 4).

The current implementation of BFSExpM estimates the memory usage conservatively from the number of arrays of size n in each tree node and the selected integer type. Currently, we use 32-bit integers and there are four arrays per tree node, thereby allowing BFSExpM to have 104 active tree nodes before it goes into DFS mode. Here, the ordinary DFS memory usage is added, which in this case is at most seven tree nodes extra (the maximum distance from the root for this search tree). More precise memory estimates would be possible by allowing visitors to specify their memory usage.



(a) cmz

(b) k

(c) mz-aug

(d) Legend

Fig. 5. Selected benchmark results for all combinations {BFSExp, DFS} × {F, FL, FLM} × {Bas, Schr} with all node invariants enabled. To reduce clutter, not all configurations are shown in all plots. For collection k, only the FLM configurations are shown. For mz-aug, the two nauty types are left out and only the F and FLM configurations are shown.

Deterministic Schreier-Sims for pruning with automorphisms. The basic automorphism pruning scheme is relatively cheap but may leave symmetric tree nodes to be explored. However, a deterministic version of the Schreier-Sims algorithm can compute the exact stabilizers, and therefore provide maximal pruning, but is computationally expensive. Many stochastic variations exist [29] that may provide a better trade-off between stabilizer computation and tree exploration. As a first step in this investigation, we implemented a pruning scheme based on the deterministic Schreier-Sims algorithm and benchmarked it to see the full potential for automorphism pruning. The benchmarks were done for the set {BFSExp, DFS} × {F, FL, FLM} of algorithm configurations, with all node invariants enabled. As expected, the number of nodes explored is either lower or the same as the Schreier scheme compared to the corresponding configuration with the basic scheme. However, even when the node count is lower, the actual performance is often either worse or the same, due to the computational overhead of computing accurate stabilizers.

In Figure 5, a few of the more interesting results are shown, where results for the cmz collection (Figure 5(a)) illustrate the overhead. Only the FLM configuration benefits from the Schreier scheme

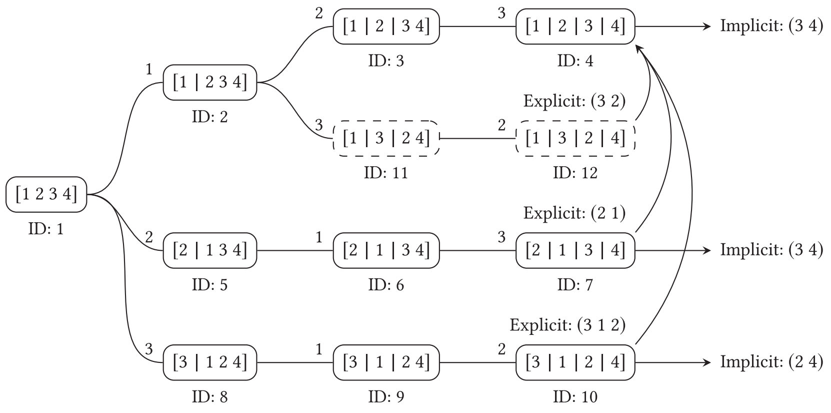


Fig. 6. Depiction of search trees using BFSExp on a clique of size 4. When using the Schreier scheme, the resulting tree appears as node IDs 1 through 10, whereas the basic scheme results in the slightly larger tree with the additional two dashed nodes (IDs 11 and 12). Tree nodes pruned in both configurations are not shown. Each node contains its ordered partition, and the non-root nodes have the individualized vertex annotated. The sequence ID for each tree node is also shown, which indicates the order of creation. The discovered implicit and explicit automorphisms are additionally shown, using cycle notation. The BFSExp traversal means that an experimental path is created from each child of the root node first. As the implicit automorphism (3 4) is found early, only three of the root children are actually explored. The difference between the basic scheme and Schreier scheme appears when the traversal revisits node 2 to create experimental paths out of each child. The basic scheme can only use automorphisms that fix the vertices individualized on the path to the root (i.e., vertex 1). It therefore does not find out that vertex 2 and vertex 3 are equivalent. In contrast, the Schreier scheme computes the complete stabilizer, which contains the permutation $(3\ 1\ 2) \cdot (2\ 1) = (2\ 3)$. It thus realizes that vertex 2 and vertex 3 are in the same orbit and can skip the exploration also of the two dashed nodes (IDs 11 and 12). Note that figures similar to this one can be obtained using the provided debug visitor and the online interactive *GraphCanon Visualizer* [3].

in terms of node count, but they all have a performance penalty, even such that FLM-Schr is slower than FLM-Bas. For the `mz-aug` collection (Figure 5(c)), the time performance is more in line with the node counts. Although the FL (not shown) and F configurations are unaffected by the pruning scheme, the BFSExp-FLM configuration benefits spectacularly from the Schreier scheme. With the basic scheme, it runs out of memory on some permutations of larger instances but otherwise performs as its Schreier counterpart. This indicates that the basic scheme, for the BFSExp-FLM configuration, to a larger degree relies on finding "good" automorphisms that can immediately be used for pruning (i.e., permutations with few points moved). Such an effect is illustrated consistently with the collection of cliques (Figure 5(b)). The BFSExp configurations exhibit the behavior, and as the choice of target cell selector is irrelevant for this graph class, we only show the FLM configurations. With the basic scheme, the number of tree nodes for the BFSExp configurations grows to several orders of magnitude higher than the BFSExp-Bas configurations, and even the DFS configurations. In this case, the difference in node count is enough to offset the overhead, and the BFSExp-Schr are faster than their BFSExp-Bas counterparts.

To better illustrate the effect of using pruning methods based on the Schreier-Sims algorithm, we now consider the simple case of using BFSExp on the clique of size 4. An overlay of the two search trees is shown in Figure 6, where also discovered implicit and explicit automorphisms are annotated. Following the sequence IDs of each vertex, indicating their order of creation, we see that the Schreier configuration can skip exploration of two more nodes, the dashed ones with IDs 11 and 12, compared to the basic configuration.

6 CONCLUDING REMARKS

We have presented a versatile framework for fast graph canonization algorithms that makes it easy to implement and compare heuristics. Not only does it perform better than the established tools on several graph classes, but we find interesting performance separations between different choices of heuristics that are not immediately possible with the established tools. In the future, we will expand the set of available heuristics to approach a full algorithm library for graph canonization. The library is available online [2], and an upcoming version will additionally support directed graphs.

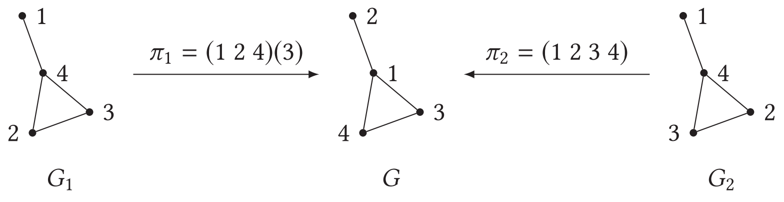
Although the established tools can handle graphs with vertex attributes, the presented framework can also directly handle edge attributes and even exploit them for refinement. However, as for previous algorithms, the support is limited to vertex and edge attributes where one can define an intrinsic total order (e.g., integers and strings), which for example is not general enough to handle attributes used for encoding stereo-chemistry in molecule graphs [8, 15], where attributes are only partially ordered. In our preliminary investigations, we find that it is possible to lift this restriction, and that a novel type of node invariant can be introduced to exploit complex attributes for pruning.

APPENDIX

A COMPARISON OF GRAPH REPRESENTATIONS

The canonization algorithm relies on a total order to exist on the set of all graph representations. For adjacency matrices, we can find the order among two graphs such as by lexicographic comparison of the matrices. When edge attributes are present, we can imagine them being stored in the matrix and then require the attribute domain to be totally ordered. With vertex attributes, we can simply modify the comparison such that we first lexicographically compare vertex attributes in order of the vertex index and then the matrices.

For adjacency lists, a similar comparison can be defined. Assuming the vertex indices are defined by the position of the vertices in the data structure, we still have freedom to order each of the lists of incident edges. We can say that an adjacency list is *globally ordered* if each edge list is sorted by the neighboring vertex index, and with edge attributes and multigraphs, we further require parallel edges to be ordered by the edge attribute. Two globally ordered adjacency lists can then be compared lexicographically in the natural manner. Vertex attributes can be trivially incorporated in this procedure. An example of globally ordered adjacency lists are shown in Figure 7, along with



G_1	G	G_2
1: 4	1: 2, 3, 4	1: 4
2: 3, 4	2: 1	2: 3, 4
3: 2, 4	3: 1, 4	3: 2, 4
4: 1, 2, 3	4: 1, 3	4: 1, 2, 3

Fig. 7. Three isomorphic graphs represented as adjacency lists. The underlying indices of the vertices are shown, and the permutations π_1 and π_2 (in cycle notation) describe the relationship between the indices of the graphs. From the adjacency lists, we see that G_1 is representationally equal to G_2, and representationally different from G. The permutation $\pi' = \pi_1\overline{\pi_2} = (2\ 3)(1)(4)$ thus represents an isomorphism from G_1 to G_2.

a visualization of how automorphisms can be detected by comparing graph representations from different permutations of the input indices.

B FRAMEWORK DETAILS

B.1 Visitor Concept

The following is an outline of the *GraphCanonVisitor* concept, omitting technical details and several methods (e.g., for callbacks during refinement). The concept is not explicitly codified in the implementation, but the compound visitor (▷C++) in practice enforces it.

A type Vis satisfies the *GraphCanonVisitor* concept if the following requirements are fulfilled.

Associated types. Vis must have the following nested types:

- TreeData: The type of a data structure to be instantiated for each search tree.
- NodeData: The type of a data structure to be instantiated for each search tree node.
- CanSelectTargetCell: A type convertible to TrueType or FalseType indicating whether the visitor implements target cell selection.
- CanTraverseTree: A type convertible to TrueType or FalseType indicating whether the visitor implements tree traversal.

Syntax.

- vis, an object of type Vis
- τ_{root}, a tree node representing the root of a search tree
- τ, an arbitrary tree node
- γ, a non-trivial permutation in S_n
- position, an integer indicating the start of a cell

Valid expressions.

- `vis.traverseTree(`τ_{root}`)`, if CanSelectTargetCell is convertible to TrueType
  Must implement a tree traversal algorithm. Called from `canon`, line 5 of Algorithm 2.
- `vis.selectTargetCell(`τ`)`, if CanSelectTargetCell is convertible to TrueType
  Must implement a target cell selector, T. Called from `makeTreeNode`, line 9 of Algorithm 4.
- `vis.isomorphicLeaf(`τ`)`
  Called from `addLeaf`, line 12 of Algorithm 3.
- `vis.implicitAutomorphism(`γ`)`
  May be called at any time by visitors.
- `vis.treeNodeCreateBegin(`τ`)`
  Called from `makeTreeNode`, line 5 of Algorithm 4.
- `vis.treeNodeCreateEnd(`τ`)`
  Called from `makeTreeNode`, line 13 of Algorithm 4.
- `vis.treeNodeDestroy(`τ`)`
  Called from `destroyTreeNode` , line 19 of Algorithm 4.
- `vis.refine(`τ`)`
  Must implement a refinement function R, but in-place. Must call `refineAbort` on the overall visitor object if it returns due to the tree node becoming pruned. Called from `makeTreeNode`, line 7 of Algorithm 4.
- `vis.refineAbort(`τ`)`
  Called by refinement functions.

- vis.beforeDescend(τ)
  Should be called by tree traversal algorithms before deciding which child to create next.
- vis.newCell(τ, *position*)
  Must be called by refinement functions for each new cell split.

B.2 Pseudocode for Framework Methods

The pseudocode is shown in Figure 8.

ALGORITHM 2: Canonization Function

```
1  def canon($G$, vis , vComp , edgeHandler):                          ▷C++
     // We implicitly assume that references to
     // the following variables are passed
     // recursively to all functions:
     // $G$, vis, edgeHandler, and canonLeaf.
2    canonLeaf ← nil
3    $\pi_0$ ← The ordered partition $(V_1, V_2, \ldots, V_k)$ as described
     in Sec. 3, but using vComp for vertex comparison. ▷C++
4    $\tau_{root}$ ← makeTreeNode(nil, $\pi_0$)
5    vis .traverseTree($\tau_{root}$)
6    $\pi_{canon}$ ← canonLeaf .$\pi$
     // Return just the permutation. It is then
     // up to the user to permute the graph.
7    return $\overline{\pi_{canon}}$
```

ALGORITHM 3: Tree Traversal Support

```
1  def addLeaf($\tau_{leaf}$):                                           ▷C++
2    if canonLeaf = nil then
3      canonLeaf ← $\tau_{leaf}$
4      return
5    $\pi_{canon}$ ← canonLeaf.$\pi$
6    $\pi_{leaf}$ ← $\tau_{leaf}.\pi$
7    $G_{canon}$ ← $G^{\overline{\pi_{canon}}}$
8    $G_{leaf}$ ← $G^{\overline{\pi_{leaf}}}$
9    if $G_{leaf} \overset{r}{<} G_{canon}$ then
10     canonLeaf ← $\tau_{canon}$
11   else if $G_{leaf} \overset{r}{=} G_{canon}$ then
12     vis .isomorphicLeaf($\tau_{leaf}$)

13 def makeChildNode($\tau_{parent}$, $w$):                              ▷C++
14   $\pi_{child}$ ← $\tau_{parent}.\pi \downarrow w$
15   $\tau_{child}$ ←makeTreeNode($\tau_{parent}$, $\pi_{child}$)
16   if $\tau_{child}$ = nil then
17     $\tau_{parent}$.childPruned[$w$]← true
18   else
19     $\tau_{child}$.individualizedVertex ← $w$
20     $\tau_{parent}$.child[$w$]← nonOwningRef($\tau_{child}$)
21   return $\tau_{child}$
```

ALGORITHM 4: Tree Node

```
1  def makeTreeNode($\tau_{parent}$, $\pi$):                             ▷C++
2    $\tau$.parent ← $\tau_{parent}$
3    $\tau.\pi$ ← $\pi$
4    $\tau$.isPruned ← false
5    vis .treeNodeCreateBegin($\tau$)
6    if not $\tau$.isPruned then
7      vis .refine($\tau$)
8      if not $\tau$.isPruned and not $\tau.\pi$ discrete then
9        $\tau$.targetCell ←vis .selectTargetCell($\tau$)
         // Initialize children references
         // and pruning status.
10       foreach $w \in \tau$.targetCell do
11         $\tau$.child[$w$] ← nil
12         $\tau$.childPruned[$w$]← false
13   vis .treeNodeCreateEnd($\tau$)
14   if $\tau$.isPruned then
15     return nil
16   else
17     return $\tau$

18 def destroyTreeNode($\tau$):                                          ▷C++
     // Automatically called when
     // the reference count for $\tau$ goes to 0.
19   vis .treeNodeDestroy($\tau$)
20   if $\tau$.parent ≠ nil then
21     $w$ ← $\tau$.individualizedVertex
22     $\tau$.parent .child[$w$]← nil
23     $\tau$.parent .childPruned[$w$]← true
```

ALGORITHM 5: Tree Pruning Support

```
1  def pruneTree($\tau$):                                                ▷C++
2    if $\tau.\pi$ is discrete then
3      if $\tau$ = canonLeaf then
4        canonLeaf ← nil
5      return
6    foreach $w \in \tau$.targetCell do
7      $\tau$.childPruned[$w$]← true
8      $\tau_{child}$ ← $\tau$.child[$w$]
9      if $\tau_{child}$ ≠ nil then
10       pruneTree($\tau_{child}$)
```

Fig. 8. The core of the canonization algorithm framework. Algorithm 2: The entry point for canonization, called with the input graph, a compound visitor, and objects for incorporating vertex and edge attributes. Algorithm 3: The supporting methods for tree traversal algorithms, with makeChildNode for making new child nodes specified by the vertex to individualize, and the method addLeaf for initiating leaf comparison. Algorithm 4: The constructor and destructor methods for tree nodes. Note that the destructor is automatically called when the reference count of a node reaches zero. Algorithm 5: The method for marking a subtree as pruned. Note that each core method calls specific methods on the visitors to facilitate injection of code at appropriate points. In addition, each tree node reference is an owning reference, except the one created with nonOwningRef in line 20 of Algorithm 4. Additionally, the C++ implementation on GitHub can be reached via the "▷C++" hyperlinks in the right margin.

ALGORITHM 6: Visitor: DFS ▷C++

```
@ traverseTree($\tau$):
  vis .beforeDescend($\tau$)
  if $\tau$.isPruned then
    return
  if $\tau.\pi$ is discrete then
    addLeaf($\tau$)
    return
  foreach $w \in \tau$.targetCell do
    vis .beforeDescend($\tau$)
    if $\tau$.isPruned then
      return
    if $\tau$.childPruned[$w$] then
      continue
    $\tau_{child}$←makeChildNode($\tau$, $w$)
    if $\tau_{child} \neq$ nil then
      traverseTree($\tau_{child}$)
```

ALGORITHM 7: Visitor: Pruning With Automorphisms ▷C++

```
Tree data: $A = \{\}$, a set of permutations, generating the
           group $\langle A \rangle$.
Node data: $k = 0$, number of permutations used for last
           pruning.
Node data: stab = {}, (a subset of) the stabilizer of $\langle A \rangle$ with
           respect to the tree node.
def pruneChildren($\tau$):
  if $\tau$.parent = nil then
    $k_p \leftarrow |A|$
  else
    pruneChildren($\tau$.parent)
    $k_p \leftarrow \tau$.parent .$k$
  if $\tau.k = k_p$ then
    return
  Update $\tau$.stab and $\tau.k$.
  Prune children of $\tau$, using the permutations of $\tau$.stab,
  while preserving canonLeaf.

@ beforeDescend($\tau$):
  pruneChildren($\tau$)

@ isomorphicLeaf($\tau_a$):
  $\tau_c \leftarrow$ canonLeaf
  $\pi_c \leftarrow \tau_c.\pi$
  $\pi_a \leftarrow \tau_a.\pi$
  $\gamma \leftarrow \overline{\pi_c}\pi_a$
  $A \leftarrow A \cup \{\gamma\}$
  $\tau_p \leftarrow$ The ancestor of $\tau_a$ for which its parent is lca($\tau_c$, $\tau_a$)
  pruneTree($\tau_p$)

@ implicitAutomorphism($\gamma$):
  $A \leftarrow A \cup \{\gamma\}$
```

ALGORITHM 8: Visitor: Implicit Automorphisms, Cell Size 2 ▷C++

```
Tree data: $\gamma = (1)$, a permutation, initial the identity.
Node data: fulfilled = false, whether the partition of the
           tree node fulfills the conditions of the lemma.
@ treeNodeCreateBegin($\pi$):
  if $\pi$.parent = nil then
    return
  if not $\pi$.parent .fulfilled then
    return
  $\pi$.fulfilled ← true
  // All cells in the parent have size 1 or 2, so
     the cell with the individualized vertex has a
     singleton neighbor. Add a swap of those two
     vertices to the permutation.
  $u \leftarrow \pi$.individualizedVertex
  $v \leftarrow$ vertex(pos($u$, $\pi$)+1, $\pi$)
  $\gamma \leftarrow \gamma \cdot (u\ v)$          // Permutation composition.

@ newCell($\pi$, $p$):
  if not $\pi$.fulfilled then
    return
  // All cells have size 1 or 2, so the new cell
     at position $p$ has a neighbor which is also a
     singleton. Add a swap of those two vertices
     to the permutation.
  $u \leftarrow$ vertex($p - 1$, $\pi$)
  $v \leftarrow$ vertex($p$, $\pi$)
  $\gamma \leftarrow \gamma \cdot (u\ v)$

@ treeNodeCreateEnd($\pi$):
  if $\pi$.isPruned then
    if $\pi$.fulfilled then
      $\gamma \leftarrow (1)$
    return
  // Report the automorphism or check if we now
     fulfill the lemma.
  if $\pi$.fulfilled then
    vis .implicitAutomorphism($\gamma$)
    $\gamma \leftarrow (1)$
  else if not $\pi$ discrete then
    if all cells of $\pi$ have size 1 or 2 then
      $\pi$.fulfilled ← true
  // Perform pruning.
  if $\pi$.fulfilled and not $\pi$ discrete then
    // The target cell only has two vertices.
    $u, v \leftarrow \pi$.targetCell
    // Other visitors may have already pruned
       children.
    if not $\pi$.childPruned[$u$] and not $\pi$.childPruned[$v$] then
      $\pi$.childPruned[$v$] ← true
```

Fig. 9. Pseudocode for a selection of visitors. The visitor for automorphism pruning (Algorithm 7) is in the implementation parameterized such that it can be used with different implementations of permutation group constructs. Algorithm 8 is an implementation of one of the cases in Lemma 2–25 in the work of McKay [21], where an automorphism can be deduced when a WL algorithm of any dimension is used to refine an ordered partition containing only cells of size 1 or 2.

B.3 Tree Traversal

The tree traversal visitor uses the methods `makeChildNode` and `addLeaf` (Figure 8) to create new tree nodes and report leaf nodes. Before deciding which child to create next, it is expected to call the visitor method `beforeDescend` to allow for pruning of children. Pruning of tree nodes is done by visitors by calling the `pruneTree` procedure (line 1 of Algorithm 4), which does not remove the designated subtree but simply sets a flag isPruned on each node. Visitors are then responsible for checking this flag before inspecting a tree node. An example of the DFS implementation is shown in Algorithm 6.

B.4 Automorphisms

Explicit automorphisms are provided by the core through the visitor method `isomorphicLeaf`, where some pruning is immediately done. Implicit automorphisms are reported from visitors through `implicitAutomorphism`. Pruning of children takes place in the `beforeDescend` visitor method. Pseudocode for a generic automorphism pruner is shown Figure 9, Algorithm 7, whereas pseudocode for a visitor for deducing implicit automorphisms in specific cases is shown in Algorithm 8.

C EXAMPLES OF SEARCH TREE VISUALIZATION

The provided stats visitor (▷C++) makes it trivial to create visualizations of the explored search tree, in addition to obtaining statistics. An example of search tree visualization for a small graph is shown in Figure 10 and for a larger graph in Figure 11. Additional examples can be found in the online interactive *GraphCanon Visualizer* [3], where logs from the debug visitor can be uploaded.

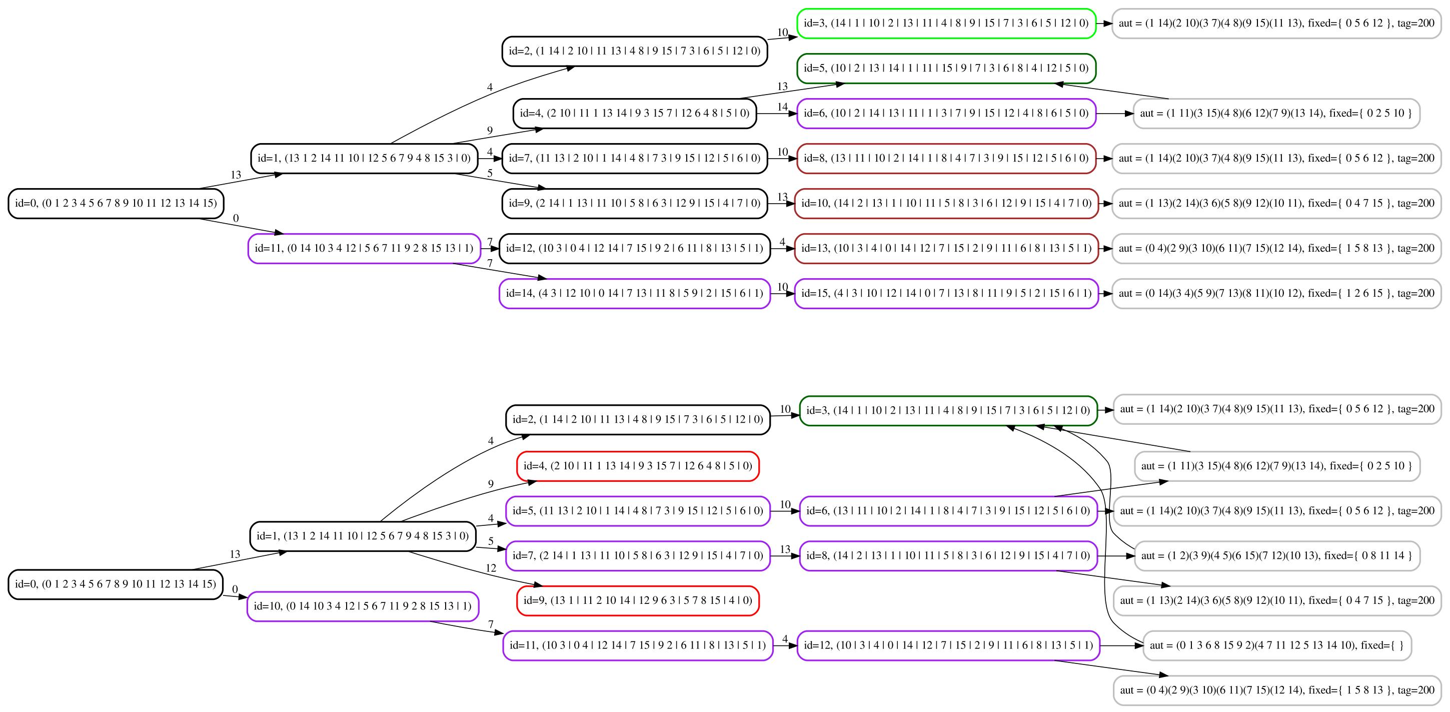

id=0, (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
id=1, (13 1 2 14 11 10 | 12 5 6 7 9 4 8 15 3 | 0)
id=11, (0 14 10 3 4 12 | 5 6 7 11 9 2 8 15 13 | 1)
id=2, (1 14 | 2 10 | 11 13 | 4 8 | 9 15 | 7 3 | 6 | 5 | 12 | 0)
id=4, (2 10 | 11 1 13 14 | 9 3 15 7 | 12 6 4 8 | 5 | 0)
id=7, (11 13 | 2 10 | 1 14 | 4 8 | 7 3 | 9 15 | 12 | 5 | 6 | 0)
id=9, (2 14 | 1 13 | 11 10 | 5 8 | 6 3 | 12 9 | 15 | 4 | 7 | 0)
id=12, (10 3 | 0 4 | 12 14 | 7 15 | 9 2 | 6 11 | 8 | 13 | 5 | 1)
id=14, (4 3 | 12 10 | 0 14 | 7 13 | 11 8 | 5 9 | 2 | 15 | 6 | 1)
id=3, (14 | 1 | 10 | 2 | 13 | 11 | 4 | 8 | 9 | 15 | 7 | 3 | 6 | 5 | 12 | 0)
id=5, (10 | 2 | 13 | 14 | 1 | 11 | 15 | 9 | 7 | 3 | 6 | 8 | 4 | 12 | 5 | 0)
id=6, (10 | 2 | 14 | 13 | 11 | 1 | 3 | 7 | 9 | 15 | 12 | 4 | 8 | 6 | 5 | 0)
id=8, (13 | 11 | 10 | 2 | 14 | 1 | 8 | 4 | 7 | 3 | 9 | 15 | 12 | 5 | 6 | 0)
id=10, (14 | 2 | 13 | 1 | 10 | 11 | 5 | 8 | 3 | 6 | 12 | 9 | 15 | 4 | 7 | 0)
id=13, (10 | 3 | 4 | 0 | 14 | 12 | 7 | 15 | 2 | 9 | 11 | 6 | 8 | 13 | 5 | 1)
id=15, (4 | 3 | 10 | 12 | 14 | 0 | 7 | 13 | 8 | 11 | 9 | 5 | 2 | 15 | 6 | 1)
aut = (1 14)(2 10)(3 7)(4 8)(9 15)(11 13), fixed={ 0 5 6 12 }, tag=200
aut = (1 11)(3 15)(4 8)(6 12)(7 9)(13 14), fixed={ 0 2 5 10 }
aut = (1 14)(2 10)(3 7)(4 8)(9 15)(11 13), fixed={ 0 5 6 12 }, tag=200
aut = (1 13)(2 14)(3 6)(5 8)(9 12)(10 11), fixed={ 0 4 7 15 }, tag=200
aut = (0 4)(2 9)(3 10)(6 11)(7 15)(12 14), fixed={ 1 5 8 13 }, tag=200
aut = (0 14)(3 4)(5 9)(7 13)(8 11)(10 12), fixed={ 1 2 6 15 }, tag=200
13
0
4
9
4
5
7
7
10
13
14
10
13
4
10
id=0, (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
id=1, (13 1 2 14 11 10 | 12 5 6 7 9 4 8 15 3 | 0)
id=10, (0 14 10 3 4 12 | 5 6 7 11 9 2 8 15 13 | 1)
id=2, (1 14 | 2 10 | 11 13 | 4 8 | 9 15 | 7 3 | 6 | 5 | 12 | 0)
id=4, (2 10 | 11 1 13 14 | 9 3 15 7 | 12 6 4 8 | 5 | 0)
id=5, (11 13 | 2 10 | 1 14 | 4 8 | 7 3 | 9 15 | 12 | 5 | 6 | 0)
id=7, (2 14 | 1 13 | 11 10 | 5 8 | 6 3 | 12 9 | 15 | 4 | 7 | 0)
id=9, (13 1 | 11 2 10 14 | 12 9 6 3 | 5 7 8 15 | 4 | 0)
id=11, (10 3 | 0 4 | 12 14 | 7 15 | 9 2 | 6 11 | 8 | 13 | 5 | 1)
id=3, (14 | 1 | 10 | 2 | 13 | 11 | 4 | 8 | 9 | 15 | 7 | 3 | 6 | 5 | 12 | 0)
id=6, (13 | 11 | 10 | 2 | 14 | 1 | 8 | 4 | 7 | 3 | 9 | 15 | 12 | 5 | 6 | 0)
id=8, (14 | 2 | 13 | 1 | 10 | 11 | 5 | 8 | 3 | 6 | 12 | 9 | 15 | 4 | 7 | 0)
id=12, (10 | 3 | 4 | 0 | 14 | 12 | 7 | 15 | 2 | 9 | 11 | 6 | 8 | 13 | 5 | 1)
aut = (1 14)(2 10)(3 7)(4 8)(9 15)(11 13), fixed={ 0 5 6 12 }, tag=200
aut = (1 11)(3 15)(4 8)(6 12)(7 9)(13 14), fixed={ 0 2 5 10 }
aut = (1 14)(2 10)(3 7)(4 8)(9 15)(11 13), fixed={ 0 5 6 12 }, tag=200
aut = (1 2)(3 9)(4 5)(6 15)(7 12)(10 13), fixed={ 0 8 11 14 }
aut = (1 13)(2 14)(3 6)(5 8)(9 12)(10 11), fixed={ 0 4 7 15 }, tag=200
aut = (0 1 3 6 8 15 9 2)(4 7 11 12 5 13 14 10), fixed={ }
aut = (0 4)(2 9)(3 10)(6 11)(7 15)(12 14), fixed={ 1 5 8 13 }, tag=200
13
0
4
9
4
5
12
7
10
13
4
10

Fig. 10. Example of search trees generated from the same graph (`latin-4`), with the same random permutation of the input. DFS was used for tree traversal and FLM for target cell selection. In the top tree, no node invariants were enabled, whereas all were enabled for the bottom tree. Each tree node has a label with an ID, indicating the order of node creation, and the associated ordered partition. Each tree edge is labeled with the vertex being individualized. The right-most gray nodes (with a label “aut = ...”) are not tree nodes but represent discovered automorphisms. Two types exist: explicit automorphisms, having an in-edge and out-edge from/to leaves indicating which permuted graphs were used to discover the automorphism, and implicit automorphisms, having only an in-edge from the tree node where some visitor (implicitly indicated by the “tag”) discovered it. In this case, all implicit automorphisms happened to be discovered in leaf nodes. A red node was pruned during creation, here because of node invariants. Purple means that the node has been pruned (indirectly) through the `pruneTree` method. A brown leaf node was found to be a worse permuted graph when compared to the one represented by the (at that time) current best leaf. A light green leaf node was once the best leaf but was later discarded, whereas the dark green leaf is the resulting canonical form.

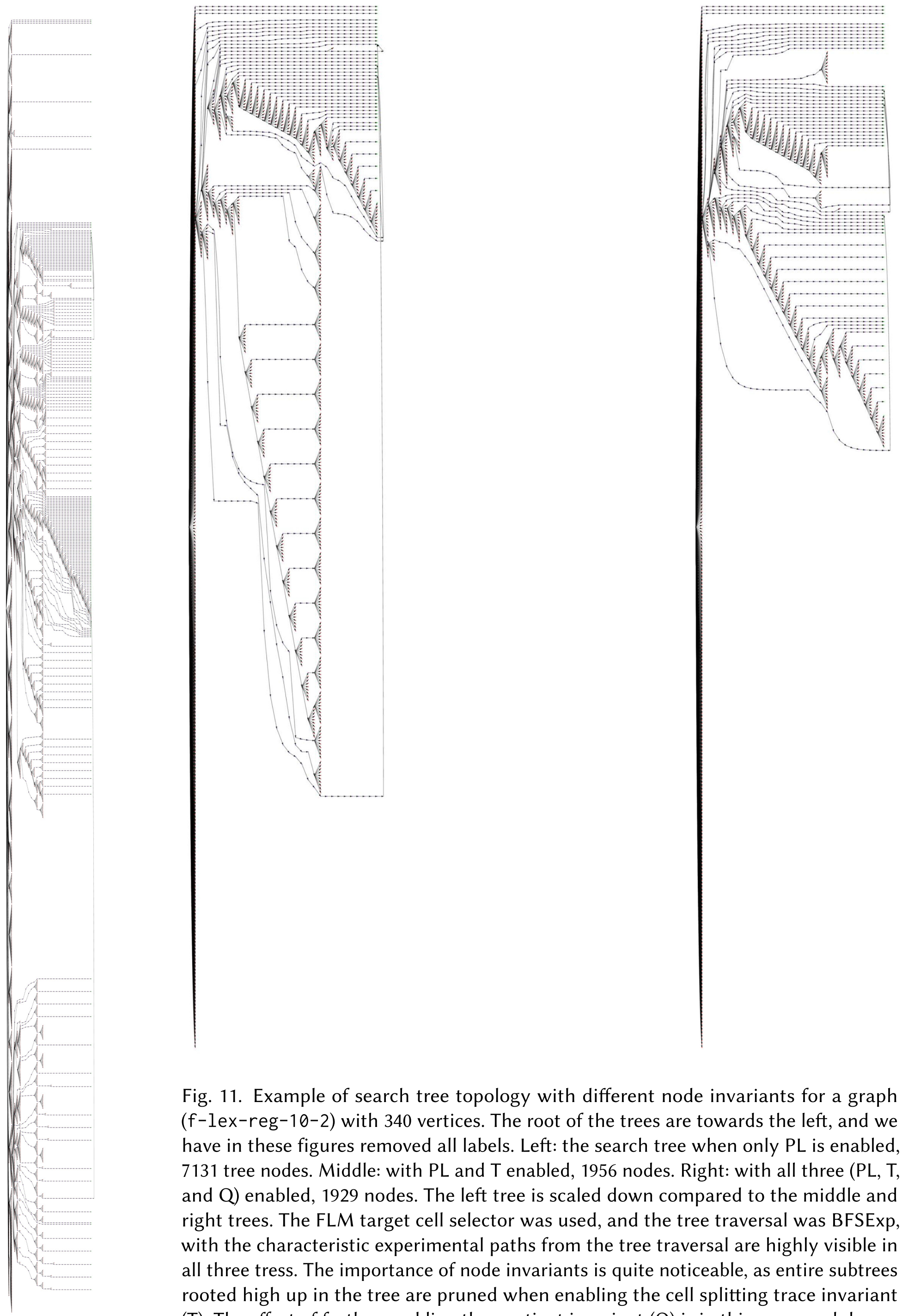


Fig. 11. Example of search tree topology with different node invariants for a graph (`f-lex-reg-10-2`) with 340 vertices. The root of the trees are towards the left, and we have in these figures removed all labels. Left: the search tree when only PL is enabled, 7131 tree nodes. Middle: with PL and T enabled, 1956 nodes. Right: with all three (PL, T, and Q) enabled, 1929 nodes. The left tree is scaled down compared to the middle and right trees. The FLM target cell selector was used, and the tree traversal was BFSExp, with the characteristic experimental paths from the tree traversal are highly visible in all three tress. The importance of node invariants is quite noticeable, as entire subtrees rooted high up in the tree are pruned when enabling the cell splitting trace invariant (T). The effect of further enabling the quotient invariant (Q) is in this case much lower.

REFERENCES

[1] Jakob L. Andersen. 2019. Benchmark Results on GitHub. Retrieved February 20, 2020 from https://github.com/jakobandersen/graph_canon_res.

[2] Jakob L. Andersen. 2019. GraphCanon Repository on GitHub. Retrieved February 20, 2020 from http://github.com/jakobandersen/graph_canon.

[3] Jakob L. Andersen. 2019. GraphCanon Visualizer on GitHub. Retrieved February 20, 2020 from http://jakobandersen.github.io/graph_canon_vis.

[4] Jakob L. Andersen. 2019. MedØlDatschgerl (MØD). Retrieved February 20, 2020 from http://mod.imada.sdu.dk.

[5] Jakob L. Andersen. 2019. PermGroup Repository on GitHub. Retrieved February 20, 2020 from http://github.com/jakobandersen/perm_group.

[6] Jakob L. Andersen. 2019. GraphCanon Results. Retrieved February 20, 2020 from https://jakobandersen.github.io/graph_canon_res.

[7] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. 2016. A software package for chemically inspired graph transformation. In *Graph Transformation*, R. Echahed and M. Minas (Eds.). Lecture Notes in Computer Science, Vol. 9761. Springer, 73–88. DOI:https://doi.org/10.1007/978-3-319-40530-8_5

[8] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. 2017. Chemical graph transformation with stereo-information. In *Proceedings of the 10th International Conference on Graph Transformation (ICGT'17), Held as Part of STAF 2017*. 54–69. DOI:https://doi.org/10.1007/978-3-319-61470-0_4

[9] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. 2019. Chemical transformation motifs—Modelling pathways as integer hyperflows. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16, 2 (March 2019), 510–523. DOI:https://doi.org/10.1109/TCBB.2017.2781724

[10] László Babai. 1996. Automorphism groups, isomorphism, reconstruction. In *Handbook of Combinatorics*. Vol. 2. MIT Press, Cambridge, MA, 1447–1540.

[11] László Babai. 2016. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the F48th Annual ACM Symposium on Theory of Computing (STOC'16)*. ACM, New York, NY, 684–697. DOI:https://doi.org/10.1145/2897518.2897542

[12] László Babai. 2019. Canonical form for graphs in quasipolynomial time: Preliminary report. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC'19)*. ACM, New York, NY, 1237–1246. DOI:https://doi.org/10.1145/3313276.3316356

[13] László Babai and Eugene M. Luks. 1983. Canonical labeling of graphs. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC'83)*. ACM, New York, NY, 171–183. DOI:https://doi.org/10.1145/800061.808746

[14] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. 2008. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. ACM, New York, NY, 149–154. DOI:https://doi.org/10.1145/1391469.1391509

[15] Stephen R. Heller, Alan McNaught, Igor Pletnev, Stephen Stein, and Dmitrii Tchekhovskoi. 2015. InChI, the IUPAC international chemical identifier. *Journal of Cheminformatics* 7, 1 (May 2015), 23. DOI:https://doi.org/10.1186/s13321-015-0068-4

[16] Tommi Junttila and Petteri Kaski. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the 2007 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. 135–149.

[17] Tommi Junttila and Petteri Kaski. 2011. *Conflict Propagation and Component Recursion for Canonical Labeling*. Springer, Berlin, Germany, 151–162. DOI:https://doi.org/10.1007/978-3-642-19754-3_16

[18] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. 2010. Symmetry and satisfiability: An update. In *Theory and Applications of Satisfiability Testing—SAT 2010*. Springer, 113–127.

[19] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. 1999. The generic graph component library. *ACM SIGPLAN Notices* 34, 10 (Oct. 1999), 399–414. DOI:https://doi.org/10.1145/320385.320428

[20] José Luis López-Presa and Antonio Fernández Anta. 2009. *Fast Algorithm for Graph Isomorphism Testing*. Springer, Berlin, Germany, 221–232. DOI:https://doi.org/10.1007/978-3-642-02011-7_21

[21] Brendan D. McKay. 1981. Practical graph isomorphism. In *Congressus Numerantium*. Vol. 30. Utilitas Mathematica Publishing Inc., Winnipeg, Manatoba, Canada, 45–97. http://cs.anu.edu.au/~bdm/papers/pgi.pdf.

[22] Brendan D. McKay and Adolfo Piperno. 2014. Practical graph isomorphism II. *Journal of Symbolic Computation* 60 (2014), 94–112.

[23] Brendan D. McKay and Adolfo Piperno. 2017. Nauty and Traces. Retrieved February 20, 2020 from http://pallini.di.uniroma1.it/Graphs.html.

[24] David R. Musser and Alexander A. Stepanov. 1988. Generic programming. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. 13–25.

[25] Daniel Neuen and Pascal Schweitzer. 2017. Benchmark graphs for practical graph isomorphism. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17)*. Article 60, 14 pages. DOI:https://doi.org/10.4230/LIPIcs.ESA.2017.60

[26] Daniel Neuen and Pascal Schweitzer. 2017. Benchmark Graphs for Practical Graph Isomorphism. Retrieved February 20, 2020 from https://www.lii.rwth-aachen.de/research/95-benchmarks.html.

[27] Adolfo Piperno. 2008. Search space contraction in canonical labeling of graphs (preliminary version). arXiv:0804.4881.

[28] Adolfo Piperno. 2018. Isomorphism test for digraphs with weighted edges. In *17th International Symposium on Experimental Algorithms (SEA 2018)*. Leibniz International Proceedings in Informatics, Vol. 103, G. D'Angelo (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, Article 30, 13 pages. DOI:https://doi.org/10.4230/LIPIcs.SEA.2018.30

[29] Ákoss Seress. 2003. *Permutation Group Algorithms*. Cambridge University Press.

[30] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *Boost Graph Library: The User Guide and Reference Manual*. Pearson Education. http://www.boost.org/libs/graph/.

[31] Boris Weisfeiler. 2006. *On Construction and Identification of Graphs*. Lecture Notes in Mathematics, Vol. 558. Springer.

Received August 2018; revised June 2019; accepted August 2019