A Generic Framework for Engineering Graph Canonization Algorithms

Jakob L. Andersen[*†‡] Daniel Merkle[†]

Abstract

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 behaviour 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.

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 especially is useful when we want to test isomorphism against a large collection of graphs, e.g., for database querying. There is a rich literature on the complexity of both graph canonization and graph isomorphism. For longer discussion we refer to [15], and simply note that for general graphs the problems are not known to be NP-complete, and the best bound for canonization is currently $e^{O(\sqrt{n \log n})}$ [4, 6], while a quasi-polynomial bound for isomorphism was recently presented [5].

For practical graph canonization there has also been extensive work, with several competitive tools being published in the last decades. They are all build 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 [14], 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 [9, 10] and Traces [15], also use this technique with the latter introducing a new way to exploit the discovered automorphisms. A related tool is Saucy [7, 11] which only performs computation of the automorphism group, for which it introduced new heuristics to discover them. Similarly there is Conauto [13] 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 refinement function which can exploit such attributes.

[*]Research group Bioinformatics and Computational Biology, Faculty of Computer Science, University of Vienna, 1090 Vienna, Austria

[†]Department of Mathematics and Computer Science, University of Southern Denmark, Odense M DK-5230, Denmark {daniel,jlandersen}@imada.sdu.dk

[‡]Earth-Life Science Institute, Tokyo Institute of Technology, Tokyo 152-8550, Japan

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 [18, 19] we perform a detailed benchmark of the effects of node invariants which suggests why some of them are more difficult than the others. Plots for all benchmarks can be found in the GitHub repository [1].

The framework is implemented as a C++ library, called **GraphCanon**, with heavy use of generic programming [17] with influences from and compatibility with the Boost Graph library [12, 22]. It is available on GitHub [1] along with the accompanying **PermGroup** library [2] for handling permutation groups. In App. C 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 and data from experiments.

In Sec. 2 and 3 we lay out the mathematical description of the individualization framework, setting the stage for the description of the framework in Sec. 4. The experimental results are presented in Sec. 5, and a summary with future developments is in Sec. 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 $V = \{1, 2, \dots, n\}$. An attributed graph is a tuple $G = (V, E, l_V, l_E)$ of a graph (V, E) and two attribution functions $l_V : V \to \Omega_V$ and $l_E : E \to \Omega_E$. 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., [9] and [15], 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 $G_1 \overset{r}{<} G_2$.

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

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 $\pi = (W_1, W_2, \dots, W_r)$ 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 $cell(u, \pi') < cell(v, \pi')$ for all $u, v \in V$. That is, 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, while 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 permutation of a subset of vertices $W \subseteq V$ with $\gamma \in S_n$ is defined as $W^\gamma = \{w^\gamma \mid w \in W\}$, while the permutation of a sequence $Q = (q_1, q_2, \dots, q_k) \in V^k$ is $Q^\gamma = (q_1^\gamma, q_2^\gamma, \dots, q_k^\gamma)$. Similarly we extend permutation to combinations of these structures, all derived from V, which in particular means we can permute ordered partitions and (representations of) graphs.

We interpret a discrete ordered partition π as a permutation in S_n, that maps each cell index to its contained vertex. That is, 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, $C\colon \mathcal{G} \to \mathcal{G}$, with the following properties, [15, C1 and C2]: $C(G) \cong G$ and $C(G^\gamma) \overset{r}{=} C(G)$, for all $\gamma \in S_n$. That is, 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 [15]. 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 $\pi_0 = (V_1, V_2, \ldots, V_k)$ with the property that for all $u \in V_i$ and $v \in V_j$, if $l_V(u) = l_V(v)$ then $i = j$, otherwise $l_V(u) < l_V(v)$ implies $i < j$. That is, the vertices are partitioned by their attribute, and the cells are ordered by the attribute.

The algorithm is defined using several abstract functions. The first one is a *refinement function* $R\colon \mathcal{G} \times \Pi \to \Pi$, with the properties [15]: $R(G, \pi) \preceq \pi$ and $R(G^\gamma, \pi^\gamma) = R(G, \pi)$ for all $\gamma \in S_n$. That is, 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* $T\colon \mathcal{G} \times \Pi \to 2^V$ that for a non-discrete partition returns one of the non-singleton cells. This function must also be isomorphism invariant, i.e., $T(G^\gamma, \pi^\gamma) = T(G, \pi)^\gamma$, 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 that order. That is $\pi \downarrow v$ is the partition strictly finer than π obtained by individualizing v to the left from the rest of its cell.

Canonization as a Tree Search 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 above. 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 $\pi_{\tau_{child}} = R(G, \pi_\tau \downarrow w)$. That is, for each child we individualize a different vertex of the target cell, and then perform refinement on the partition.[2]

We now define the canonical form of the abstract algorithm. Note though that we will slightly alter this definition later in order to facilitate pruning of the search tree. 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. Let $L(G)$ denote all leaf nodes of the search tree starting from the graph G. The canonical form $C(G)$ is then the permuted graph indicated by the leaf node with the representationally smallest permuted graph. That is $C(G) = G^{\overline{\pi_{\tau_{canon}}}}$ with $\tau_{canon} = \arg\min_{\tau \in L(G)} \left\{ G^{\overline{\pi_\tau}} \right\}$.

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. That is, 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 [15]. Sometimes it is possible to deduce automorphisms from internal nodes of the search tree. We call automorphisms found in this manner *implicit automorphisms* [14].

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. That is, $v_i^\gamma = v_i$ for $1 \leq i \leq k$, and $u^\gamma = v$. As u and

[2] An example of search tree with ordered partitions can be found in App. B.

v are equivalent under γ the two subtrees rooted at $(v_1, v_2, \dots, v_k, u)$ and $(v_1, v_2, \dots, v_k, v)$ will be isomorphic, and we can safely skip traversal of one of them [15, Operation P_C].

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* $\phi\colon \mathcal{G} \times \mathcal{T} \to \Omega$ 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* $\vec{\phi}\colon \mathcal{G} \times \mathcal{T} \to \Omega^*$. For a tree node $\tau = (v_1, v_2, \dots, v_k)$ the value is $\vec{\phi}(G, \tau) = (\phi(G, v_1), \phi(G, v_2), \dots, \phi(G, v_k))$. That is, it is the sequence of all node invariants from the root down to τ. We compare such sequences lexicographically. Finally the canonical form is then redefined to be the one with the smallest permuted graph among the leaves with the smallest path invariant:

$$\phi^* = \min_{\tau \in L(G)} \vec{\phi}(\tau)$$

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

$$C(G) = G^{\overline{\pi_{\tau_{canon}}}}$$

4 A Generic Algorithm Framework

From the abstract canonization algorithm we see that each concrete algorithm is defined by specific choices of sub-procedures for the six categories in the table below.

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

The goal of the present framework is to provide an implementation of functionality common to all algorithms, and provide a facility for attaching extensions for the six categories. Note that while only one option for target cell selection and tree traversal must be chosen, it can be beneficial to use multiple algorithms in conjunction for the remaining categories.

Using generic programming [17] we have designed a single common extension infrastructure, based on the idea of a *visitor*, similar to those used in the Boost Graph library [12, 22]. The core canonization procedure is given a visitor object vis which must fulfill a collection of requirements, i.e., it must model a specific *Visitor* concept. The concept specifies that a concrete visitor must implement a collection of callbacks that will be invoked at various points during algorithm. Each visitor must additionally specify two data structures; one that will be instantiated per search tree, and one instantiated for each node in each search tree.[3]

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 vis is such a compound visitor aggregating the sequence $(\mathsf{vis}_1, \mathsf{vis}_2, \dots, \mathsf{vis}_t)$ of visitors.

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.[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 [14, 15] and Bliss [9, 10] 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 [15] uses a different traversal scheme which combines a breadth-first traversal with so-called experimental paths to find leaves early. As noted in [18], 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 a non-owning reference to its children. Each visitor is thus responsible to keep owning references to nodes they

[3] The details of the *Visitor* concept can be found in App. C.1.

[4] Pseudocode for the framework methods can be found in the appendix, Fig. 5.

[5] Examples of search tree visualizations can be found in App. D.

are interested in. The creation of new children is done through a framework method, while 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.

We provide an implementation of the classical depth-first traversal (DFS) 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.[6]

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 [9]: 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 $(u, v_2) \notin E$. That is, all vertices of U have both neighbours and non-neighbours in W.

In the present framework the target cell selection is done during construction of each internal 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)[23], that iteratively separates vertices in a cell depending on their degree with respect to other cells. Traces [15] can also use the 2-dimensional variant. The tool nauty includes a selection of additional refinement functions [15].

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, e.g., 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, which based on the observations in [9], uses custom implementations of counting sort and array partitioning to perform fast sorting for low degree cases. As the framework directly allows for canonization of edge attributes, we have also generalized the WL-1 implementation to exploit the attributes for even stronger refinement.[7]

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 all permutations in the stabilizer of Φ with respect to the individualized vertices in τ, i.e., $\mathrm{Stab}_\Phi(\tau) = \{\gamma \in \Phi \mid v_i^\gamma = v_i, 1 \leq i \leq k\}$. This is for example done in Traces and newer versions of nauty [15], using random Schreier methods [21]. A computational less intensive method is used in Bliss and earlier versions of nauty, where only the subset $A_\tau = \{\gamma \in A \mid v_i^\gamma = v_i, 1 \leq i \leq k\}$ is considered, i.e., a direct filtering of the found automorphisms without composition. However, only a fixed number of permutations are stored at a time to conserve memory [9, 14].

The framework provides explicit automorphisms to the visitors, and they may report implicit automorphisms to each other. The actual pruning can be 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 a parameterization that 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.

[6] An example of investigating the number of tree nodes allocated can be found in App. E.

[7] The details of the generalized WL-1 algorithm can be found in App. C.3.

4.5 Detecting Implicit Automorphisms There are several known methods for finding additional automorphisms during the tree search. For example, Lemma 2-25 [14] describes three cases where all refinements of certain ordered partitions lead to isomorphic leaf nodes. The simplest, and most common, of those cases is where the partition has singleton cells or cells of size 2. Saucy [7, 11] introduced another heuristic for finding primarily sparse automorphisms, where for each non-leaf node a guess for an automorphism is made. Traces [15] reportedly generalizes this heuristic, though 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 [14], described above, and one for refining cells with only degree 1 vertices and reporting the implicit automorphisms discovered in the process.

4.6 Pruning With Node Invariants In the abstract algorithm we used a single node invariant function, though 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 [15]. Another important example is the *partial leaf* invariant introduced in Bliss [9], 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 and pruning with node invariants can be implemented entirely as visitors. A visitor for coordinating multiple invariants is provided both for ensuring correct pruning, but also to minimize the overhead of implementing a new invariant. Let Ω_i be the domain of invariant values produced by vis_i, then the node invariant values for a given tree node is a sequence of pairs $(\langle i_1, \omega_1\rangle, \langle i_2, \omega_2\rangle, \dots, \langle i_k, \omega_k\rangle)$ 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, while 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.

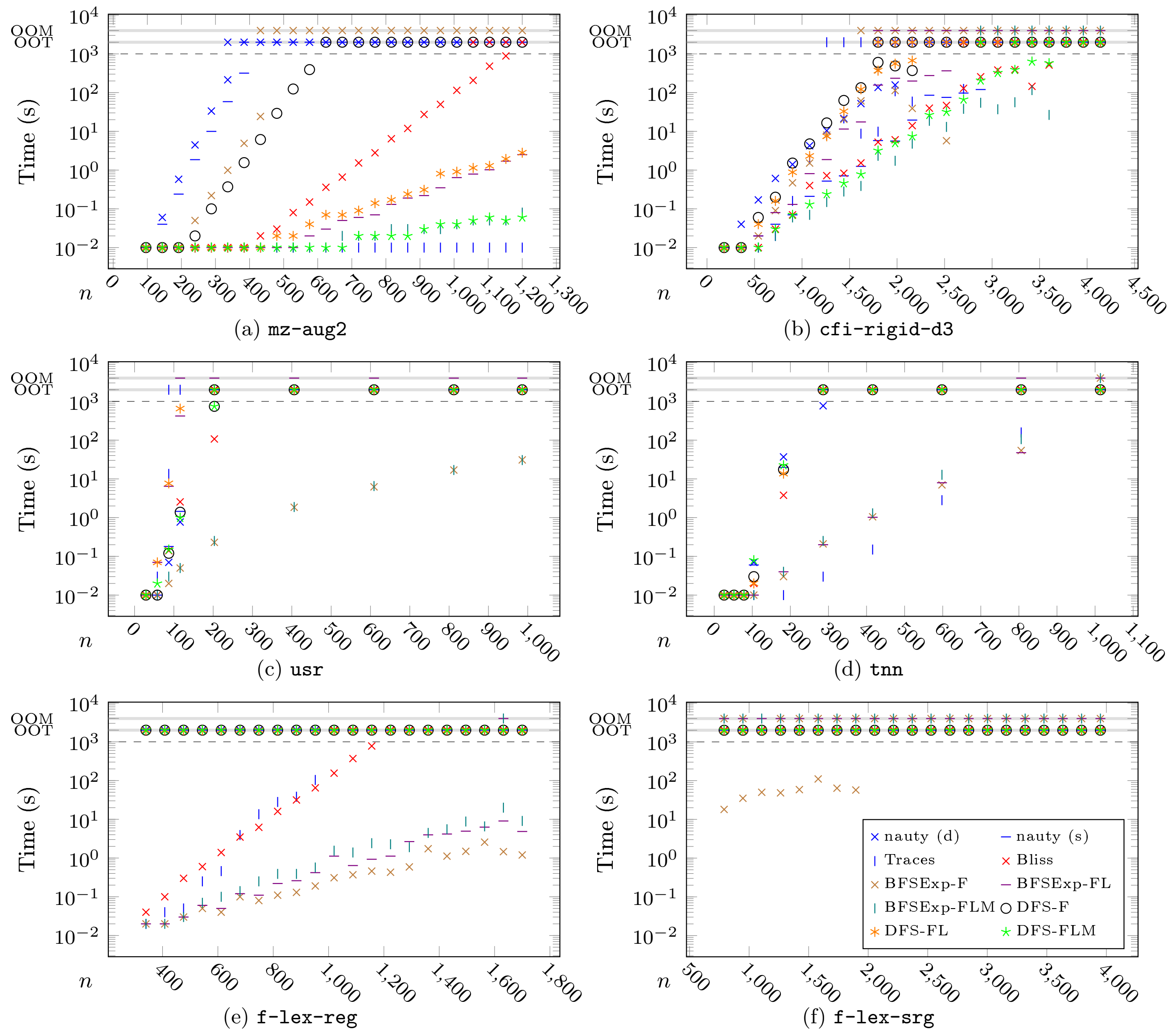
We implement three invariants: the cell splitting trace introduced in Traces (T), a trace of values for the quotient graph also introduced in Traces [15] (Q), and the partial leaf invariant introduced in Bliss (PL). Note that we do not hash the information in any of these visitors.

5 Experimental Results

While the framework is capable of handling both vertex and edge attributes there are unfortunately no comprehensive collections with such graphs. We have therefore benchmarked our implementation using both the collections of unattributed graphs from [16], and using a proposed collection of difficult graphs, `cfi-rigid` [18, 19]. All executions were repeated 5 times with random permutations of the input graph, always with a maximum of 8 GB memory and a 1000 second time limit. Of the repetitions that succeeded we plot the average time spend, 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 only show time plots. The full set of plots can be found at [1]. For comparison of absolute performance we also benchmarked the default configurations of Bliss v0.73 as well as v26r6 of nauty (dense and sparse) and Traces. All experiments were run on compute nodes with two Intel E5-2680v3 CPUs (24 cores), using in total 12,000 compute node hours. In the first part we investigate the effect of tree traversal algorithm and target cell selector, while we focus in the second part on the `cfi-rigid` collections where we also investigate the effect of different subsets of node invariants. In the third part we illustrate how the BFSExpM traversal provides a memory-safe alternative to BFSExp at the expense of time.

Tree Traversal and Target Cell Selector For all graph collections from [16] we benchmarked the set $\{\text{BFSExp}, \text{DFS}\} \times \{\text{F}, \text{FL}, \text{FLM}\}$ of algorithm configurations, with all node invariants enabled. Overall we found that no single configuration is the best performing on all instances, though 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. Fig. 1a 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 [1]) we see that the FLM target cell selector scales similar to Traces (sub-exponential), while F and FL both have exponential behavior similar to Bliss and the two nauty modes. This behavior we also see



(a) `mz-aug2` (b) `cfi-rigid-d3` (c) `usr` (d) `tnn` (e) `f-lex-reg` (f) `f-lex-srg`

Figure 1: Selected benchmark results for all combinations $\{\text{BFSExp}, \text{DFS}\} \times \{\text{F}, \text{FL}, \text{FLM}\}$, with all node invariants enabled.

in the `cmz` collection, while 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. We attribute this to the automorphism pruning where we do not perform composition via random Schreier methods. Thus, if the BFSExp tree traversal finds explicit automorphisms with few fixed vertices it may have fewer chances of using them for pruning.

On other collections, such as the non-disjoint union of tripartite graphs (`tnn`, Fig. 1d), the algorithm configurations are separated by the tree traversal algorithms. For `tnn` the performance of the BFSExp configurations is similar to Traces (which is also breadth-first-based), while for the union of strongly regular graphs (`usr`, Fig. 1c) 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`, Fig. 1e and 1f. For `f-lex-reg` we again see a separation by 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, though only for some executions.

The `cfi-rigid` Collections This package of six collections of graphs [19], was recently proposed [18] explicitly as difficult instances for graph isomorphism, and by construction they have very little symmetry. Each collection (see Tab. 1) is constructed using a group, D_3, $\mathbb{Z}_3$, or $\mathbb{Z}_2$. For $\mathbb{Z}_2$ there are further 3 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 $\{\mathrm{BFSExp}, \mathrm{DFS}\} \times \{\mathrm{F}, \mathrm{FL}, \mathrm{FLM}\} \times 2^{\{\mathrm{PL},\mathrm{Q},\mathrm{T}\}}$.

For the four $\mathbb{Z}_2$-based collections Bliss and nauty (sparse) perform well, though 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. There is however a significant separation up to F and FL configurations (Fig. 1b). 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 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. Though the effect of enabling additional invariants is minor or non-existing. Intriguingly, for `r2` where the R^* reduction were 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. We hope that future in-depth studies on these graphs and node invariants may lead to new insights, both for developing better invariants but potentially also for creating even more difficult benchmark graphs.

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 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 Fig. 2 a comparison

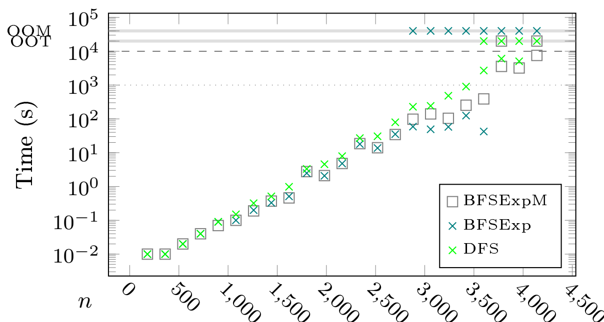


Figure 2: Comparison of tree traversal algorithms on `cfi-rigid-d3`, using FLM for target cell selection and all three node invariants.

is shown with the BFSExp and DFS configurations. We clearly see that for the instance sizes where BFSExp go out of memory on some executions, the BFSExpM configuration increase in average time spend. 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, though with some executions exceeding the time limit.

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 which is 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. While the established tools can handle graphs with vertex attributes, the presented framework can also directly handle edge attributes, and even exploit them for refinement. Though, the attribute sets are currently limited to totally ordered sets, which is not general enough to for example handle attributes used for encoding stereo-chemistry in molecule graphs [3, 8]. 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.

Acknowledgements

This work was supported in part by the 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

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

Table 1: Summary of results for the `cfi-rigid` collections. The right-most column is the largest instance that any of the configurations or the tools solved.

Templeton Foundation. Computation for the work described in this paper was supported by the DeiC National HPC Centre, SDU. The opinions expressed in this publication are those of the authors and do not necessarily reflect the views of the John Templeton Foundation.

References

[1] Jakob L. Andersen. GraphCanon repository on GitHub, 2017. `http://github.com/jakobandersen/graph_canon`.

[2] Jakob L. Andersen. PermGroup repository on GitHub, 2017. `http://github.com/jakobandersen/perm_group`.

[3] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Chemical graph transformation with stereo-information. In *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*, pages 54–69, 2017.

[4] László Babai. Automorphism groups, isomorphism, reconstruction. In *Handbook of combinatorics (vol. 2)*, pages 1447–1540. MIT Press, 1996.

[5] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 684–697, New York, NY, USA, 2016. ACM.

[6] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 171–183, New York, NY, USA, 1983. ACM.

[7] Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA, 2008. ACM.

[8] Stephen R. Heller, Alan McNaught, Igor Pletnev, Stephen Stein, and Dmitrii Tchekhovskoi. Inchi, the iupac international chemical identifier. *Journal of Cheminformatics*, 7(1):23, May 2015.

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

[10] Tommi Junttila and Petteri Kaski. *Conflict Propagation and Component Recursion for Canonical Labeling*, pages 151–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

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

[12] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. *SIGPLAN Not.*, 34(10):399–414, October 1999.

[13] José Luis López-Presa and Antonio Fernández Anta. *Fast Algorithm for Graph Isomorphism Testing*, pages 221–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[14] Brendan D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–97. Utilitas Mathematica Pub. Incorporated, 1981. `http://cs.anu.edu.au/~bdm/papers/pgi.pdf`.

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

[16] Brendan D. McKay and Adolfo Piperno. nauty and traces, 2017. `http://pallini.di.uniroma1.it/Graphs.html`.

[17] David R Musser and Alexander A Stepanov. Generic programming. In *International Symposium on Symbolic and Algebraic Computation*, pages 13–25. Springer, 1988.

[18] Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 60:1–60:14, 2017.

[19] Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism, 2017. `https://www.lii.rwth-aachen.de/research/95-benchmarks.html`.

[20] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR, abs/0804.4881*, 2008.

[21] Á. Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003.

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

[23] Boris Weisfeiler. *On construction and identification of graphs*, volume 558. Springer, 2006.

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 for example 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 Fig. 3, along with a visualization of how automorphisms can be detected by comparing graph representations from different permutations of the input indices.

B Search Tree Example

An example of a search tree is shown in Fig. 4.

C Framework Details

C.1 Visitor Concept The following is an outline of the *Visitor* concept, ommitting 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 *Visitor* 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`, Alg. 1, line 5.
- vis.`selectTargetCell`(τ), if CanSelectTargetCell is convertible to TrueType
  Must implement a target cell selector, T. Called from `makeTreeNode`, Alg. 3, line 9.

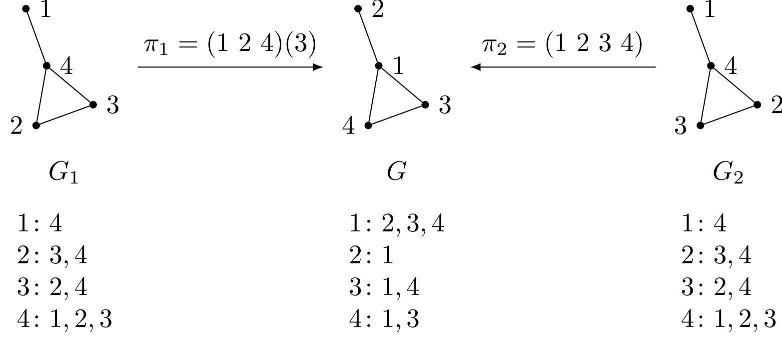


Figure 3: Three isomorphic graphs represented as adjacency lists. The underlying indices of the vertices are shown, and the permutations π_1 and π_2 (in cycle notation) describe the relationship between the indices of the graphs. From the adjacency lists we see that G_1 is representationally equal to 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.

- `vis.isomorphicLeaf`(τ)
  Called from `addLeaf`, Alg. 2, line 12.
- `vis.implicitAutomorphism`(γ)
  May be called at any time by visitors.
- `vis.treeNodeCreateBegin`(τ)
  Called from `makeTreeNode`, Alg. 3, line 5.
- `vis.treeNodeCreateEnd`(τ)
  Called from `makeTreeNode`, Alg. 3, line 13.
- `vis.treeNodeDestroy`(τ) Called from `destroyTreeNode`, Alg. 3, line 19.
- `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 becomming pruned. Called from `makeTreeNode`, Alg. 3, line 7.
- `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.

C.2 Pseudocode for Framework Methods The pseudocode is shown in Fig. 5.

C.3 Weifeler-Leman Refinement and Edge Attributes The WL-1 algorithm refines a cell by distinguishing the vertices by their degree to other cells. That is, for a cells 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. We have generalized the WL-1 algorithm to exploit edge attributes, essentially by abstracting the degree function. Ordinarily it is assumed to have the signature $d\colon V \times 2^V \to \mathbb{N}_0$, but we generalize it to return a map from each possible edge attribute to the number of edges with that attribute. That is d has the signature $V \times 2^V \to (\Omega_E \to \mathbb{N}_0)$. Such mappings are totally ordered, as we assume Ω_E is to be totally ordered.

In practice we delegate the edge handling to the given eHandler 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 Alg. 5. In this formulation the delegation to the eHandler object happens in addition to line 4, 7, 9, and 10.

Algorithm 5: WL-1 Refinement ▷C++

Input: π, a reference to an ordered partition
Input: Q, a non-empty FIFO queue of cells of π
Data: C, an associative array $V \to \mathbb{N}_0$, for counting neighbours

1 **while** *Q not empty* **and** *π 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** $X_i \in \{X_1, X_2, \ldots, X_k\} \setminus \{X_{max}\}$ **do**
19   Q.pushBack(X_i)

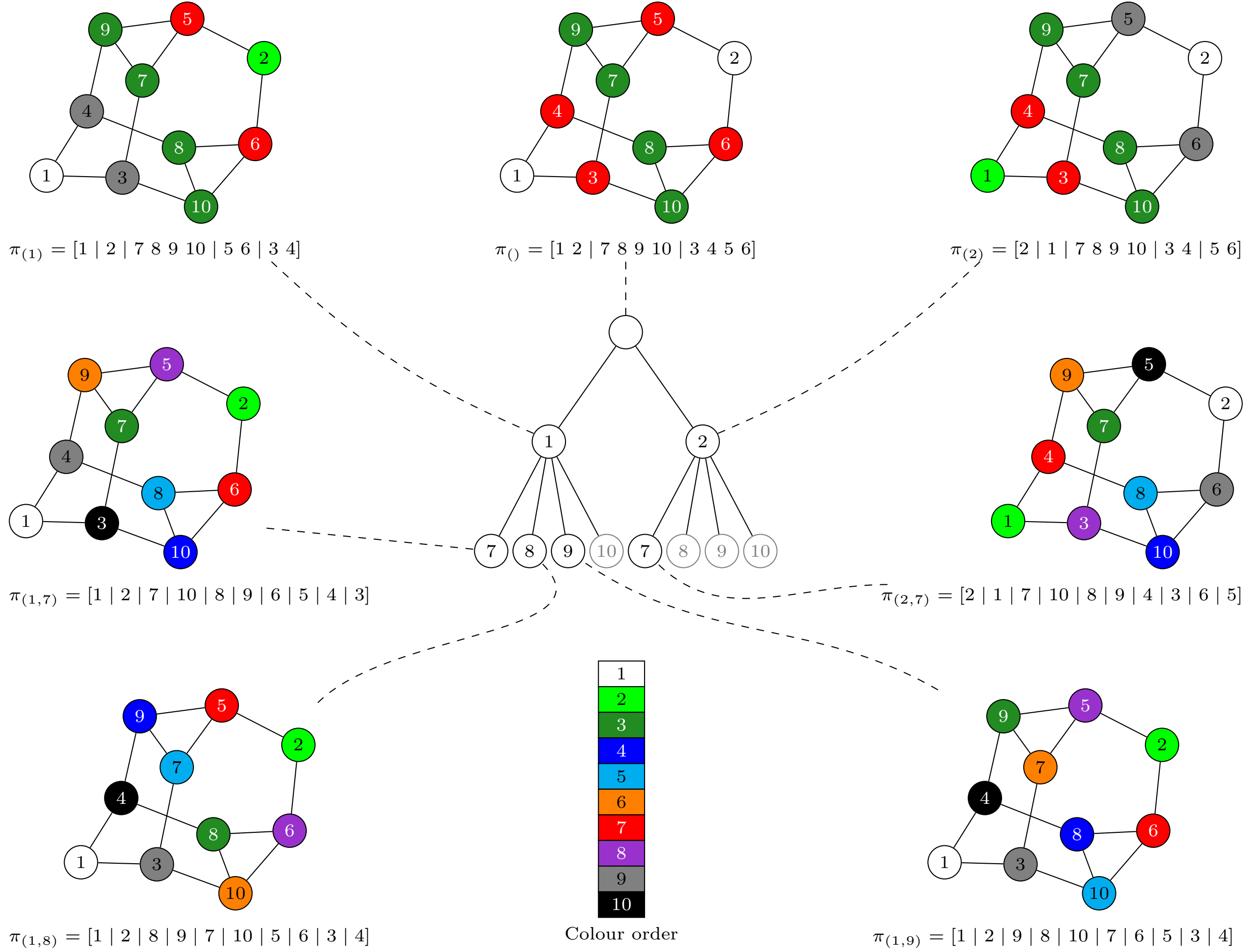


Figure 4: A search tree starting with the refinement of the unit partition in the root. The refinement function used is the WL-1 algorithm, the target cell selector is selecting the first non-singleton cell, and no node invariants are used. Each node in the tree represents a sequence of vertex individualizations, 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 grayed out nodes correspond to nodes pruned from automorphism discovery, when depth-first traversal of the tree is used. The example is heavily inspired by [20, Figure 3], though here using different functions for refinement and target cell selection.

C.4 Tree Traversal The tree traversal visitor uses the methods `makeChildNode` and `addLeaf`, Fig. 5, 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 (Alg. 3, line 1), 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 inspected a tree node.

C.5 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.

Algorithm 1: Canonization Function

```
1  def canon(G, vis, vComp, eHandler):                         ▷C++
     // We implicitly assume that references to
     // the following variables are passed
     // recursively to all functions:
     // G, vis, eHandler, and canonLeaf.
2    canonLeaf ← nil
3    π_0 ← The ordered partition (V_1, V_2, ..., V_k) as described
        in Sec. 3, but using vComp for vertex comparison.
        ▷C++
4    τ_root ← makeTreeNode(nil, π_0)
5    vis.traverseTree(τ_root)
6    π_canon ← canonLeaf.π
     // Return just the permutation.  It is then
     // up to the user to permute the graph.
7    return inverse(π_canon)
```

Algorithm 2: Tree Traversal Support

```
1  def addLeaf(τ_leaf):                                         ▷C++
2    if canonLeaf = nil then
3      canonLeaf = τ_leaf
4      return
5    π_canon ← canonLeaf.π
6    π_leaf ← τ_leaf.π
7    G_canon ← G^(inverse(π_canon))
8    G_leaf ← G^(inverse(π_leaf))
9    if G_leaf <^r G_canon then
10     canonLeaf ← τ_canon
11   else if G_leaf =^r G_canon then
12     vis.isomorphicLeaf(τ_leaf)

13 def makeChildNode(τ_parent, w):                              ▷C++
14   π_child ← τ_parent.π ↓ w
15   τ_child ← makeTreeNode(τ_parent, π_child)
16   if τ_child = nil then
17     τ_parent.childPruned[w] ← true
18   else
19     τ_child.individualizedVertex ← w
20     τ_parent.child[w] ← nonOwningRef(τ_child)
21   return τ_child
```

Algorithm 3: Tree Node

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

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

Algorithm 4: Tree Pruning Support

```
1  def pruneTree(τ):                                            ▷C++
2    if τ.π is discrete then
3      if τ = canonLeaf then
4        canonLeaf ← nil
5      return
6    foreach w ∈ τ.targetCell do
7      τ.childPruned[w] ← true
8      τ_child ← τ.child[w]
9      if τ_child ≠ nil then
10       pruneTree(τ_child)
```

Figure 5: The core of the canonization algorithm framework. Alg. 1: the entry point for canonization, called with the input graph, a compound visitor, and objects for incorporating vertex and edge attributes. Alg. 2: 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. Alg. 3: the constructor and destructor methods for tree nodes. Note that the destructor is automatically called when the reference count of a node reaches zero. Alg. 4: 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. Also, each tree node reference is an owning reference, except the one created with `nonOwningRef` in Alg. 3, line 20. Additionally, the C++ implementation on GitHub can be reached via the '▷C++' hyperlinks in the right margin.

D Example 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 Fig. 6.

E Investigating the Number of Allocated Nodes

The reson for developing the memory sensitive tree traversal BFSExpM is that the plain BFSExp may have a very large amount of tree nodes allocated at the same time. For investigating how that number develops we can use the provided debug visitor (▷C++), which facilitates printing of log messages. It additionally keeps track of the number of tree nodes allocated, using the `treeNodeCreateBegin` and `treeNodeDestroy` methods. We can thus immediately visualize how the number of allocated nodes develop. As an example we illustrate this for a relatively small graph (`cfi-rigid-d3-1260-04-2`, 1260 vertices). We ran the same input permutation with DFS, BFSExp, and BFSExpM limited to 2 MB, see Fig. 7. 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.

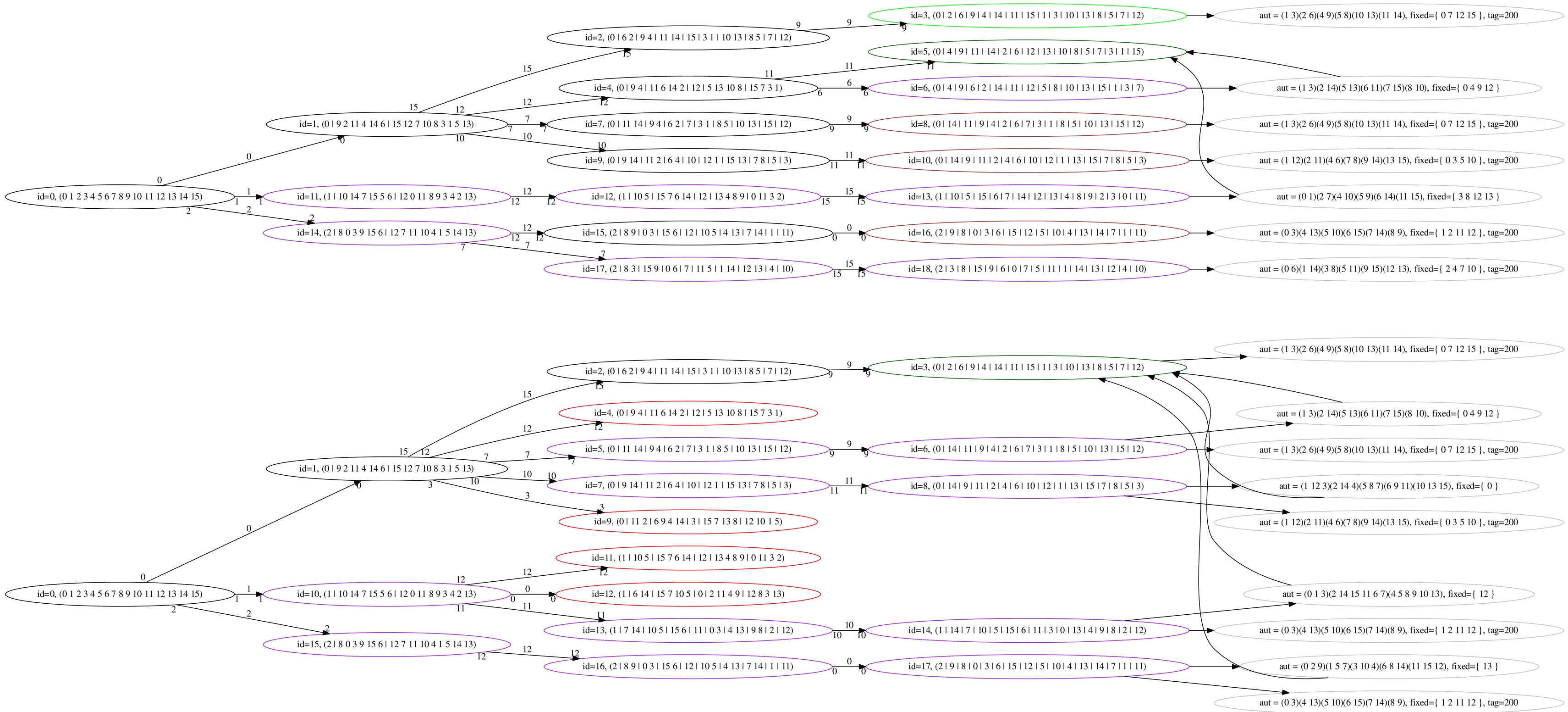


Figure 6: Example of search trees generated from the same graph (`latin-4`), with the same random permutation of the input. DFS was used for tree traversal and FLM for target cell selection. In the top tree no node invariants were enabled, while all were enabled for the bottom tree. Each tree node has a label with an ID, indicating the order of node creation, and the associated ordered partition. Each tree edge is labelled with the vertex being individualilzed. 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 worse by a comparison with the current best leaf. A light green leaf node was once the best leaf, but was later discarded, while the dark green leaf is the resulting canonical form.

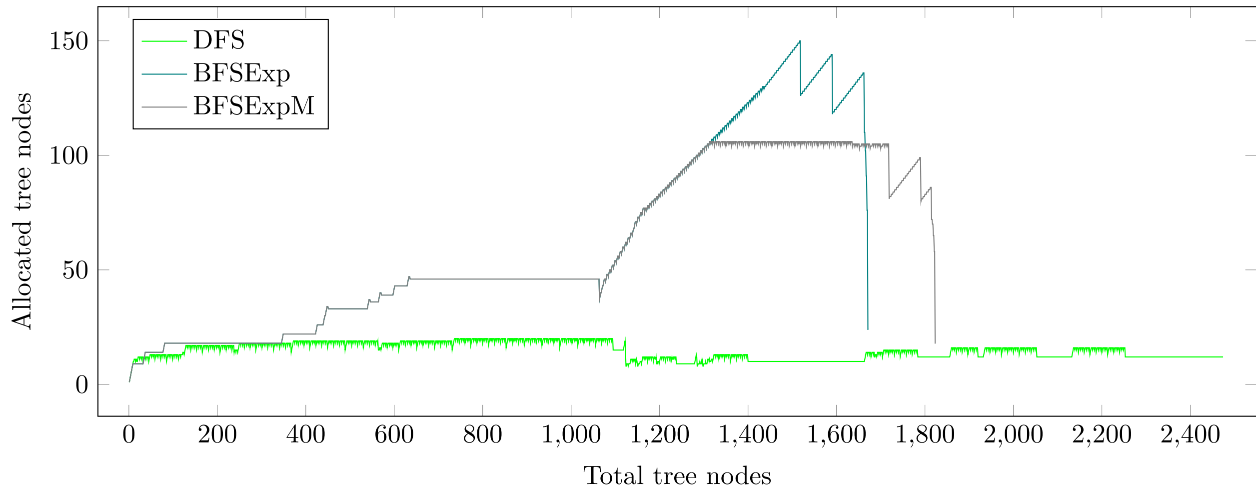


Figure 7: Trace of how many tree nodes were allocated every time a new tree node was created. We clearly see BFSExpM hitting the limit of 104 tree nodes. A large amount of nodes are later deallocated it switches back from DFS into BFSExp mode. For this choice of memory bound we see a clear trade-off between memory and time, as BFSExpM finishes later than BFSExp. It is however still faster than DFS.

Currently we use 32 bit integers and there are 4 arrays per tree node, thereby allowing BFSExpM 104 tree nodes before it goes into DFS mode. Here the ordinary DFS memory usage is added, which in this case is at most 7 tree nodes extra (the maximum distance from the root for this search tree).