Practical graph isomorphism, II ☆

Brendan D. McKay [a], Adolfo Piperno [b]

[a] *Research School of Computer Science, Australian National University, Canberra ACT 0200, Australia*
[b] *Dipartimento di Informatica, Sapienza Università di Roma, Via Salaria 113, I-00198 Roma, Italy*

ARTICLE INFO

ABSTRACT

We report the current state of the graph isomorphism problem from the practical point of view. After describing the general principles of the refinement-individualization paradigm and proving its validity, we explain how it is implemented in several of the key implementations. In particular, we bring the description of the best known program `nauty` up to date and describe an innovative approach called `Traces` that outperforms the competitors for many difficult graph classes. Detailed comparisons against `saucy`, `Bliss` and `conauto` are presented.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

An *isomorphism* between two graphs is a bijection between their vertex sets that preserves adjacency. An *automorphism* is an isomorphism from a graph to itself. The set of all automorphisms of a graph G form a group under composition called the *automorphism group* Aut(G).

The graph isomorphism problem (GI) is that of determining whether there is an isomorphism between two given graphs. GI has long been a favorite target of algorithm designers—so much so that it was already described as a "disease" in 1976 (Read and Corneil, 1977).

Though it is not the focus of this paper, we summarize the current state of the theoretical study of graph isomorphism. It is obvious that GI ∈ NP but unknown whether GI ∈ co-NP. As that implies, no polynomial time algorithm is known (despite many published claims), but neither is GI known to be NP-complete. NP-completeness is considered unlikely since it would imply collapse of the polynomial-time hierarchy (Goldreich et al., 1991). The fastest proven running time for GI has stood for three decades at $e^{O(\sqrt{n \log n})}$ (Babai et al., 1983).

On the other hand, polynomial time algorithms are known for many special classes of graphs. The most general such classes are those with a forbidden minor (Ponomarenko, 1988; Grohe, 2010) and

---

those with a forbidden topological minor (Grohe, 2012). These classes include many earlier classes such as graphs of bounded degree (Luks, 1982), bounded genus (Filotti and Mayer, 1980; Miller, 1980; Kawarabayashi and Mohar, 2008) and bounded tree-width (Bodlaender, 1990). (The correctness of Filotti–Mayer and Miller's algorithms has been questioned by Myrvold and Kocay, 2011.) The algorithms resulting from this theory are most unlikely to be useful in practice. Only for a very few important graph classes, such as trees (Aho et al., 1974), interval graphs and planar graphs (Colbourn and Booth, 1981) are there practical approaches which are sure to outperform general methods such as described in this paper.

Testing two graphs for isomorphism directly can have the advantage that an isomorphism might be found long before an exhaustive search is complete. On the other hand, it is poorly suited for the common problems of rejecting isomorphs from a collection of graphs or identifying a graph in a database of graphs. For this reason, the most common practical approach is "canonical labelling", a process in which a graph is relabelled in such a way that isomorphic graphs are identical after relabelling. When we have an efficient canonical labelling procedure, we can use a sorting, hashing or balanced tree algorithm for removing isomorphs from a large collection and standard data structures for database retrieval.

It is impossible to comprehensively survey the history of this problem since there are at least a few hundred published algorithms. However, a clear truth of history is that the most successful approach has involved fixing of vertices together with refinement of partitions of the vertex set. This "individualization-refinement" paradigm was introduced by Parris and Read (1969) and developed by Corneil and Gotlieb (1970) and Arlazarov et al. (1974). However, the first program that could handle both structurally regular graphs with hundreds of vertices and graphs with large automorphism groups was that of McKay (1978b, 1980), that later became known as `nauty`. The main advantage of `nauty` over earlier programs was its innovative use of automorphisms to prune the search. Although there were some worthy competitors (Leon, 1990; Kocay, 1996), `nauty` dominated the field for the next several decades.

This situation changed when Darga et al. (2004) introduced `saucy`, which at that stage was essentially a reimplementation of the automorphism group subset of `nauty` using sparse data structures. This gave it a very large advantage for many graphs of practical interest, prompting the first author to release a version of `nauty` for sparse graphs. Saucy has since introduced some important innovations, such as the ability to detect some types of automorphism (such as those implied by a locally tree-like structure) very early (Darga et al., 2008). Soon afterwards Junttila and Kaski (2007, 2011) introduced `Bliss`, which also used the same algorithm but had some extra ideas that helped its performance on difficult graphs. In particular, it allowed refinement operations to be aborted early in some cases. The latter idea reached its full expression in `Traces` (Piperno, 2008), which we introduce in this paper. More importantly, `Traces` pioneered a major revision of the way the search tree is scanned, which we will demonstrate to produce great efficiency gains.

Another program worthy of consideration is `conauto` (López-Presa and Fernández Anta, 2009; López-Presa et al., 2011). It does not feature canonical labelling, though it can compare two graphs for isomorphism.

In Section 2, we provide a description of algorithms based on the individualization-refinement paradigm. It is sufficiently general to encompass the primary structure of all of the most successful algorithms. In Section 3, we flesh out the details of how `nauty` and `Traces` are implemented, with emphasis on how they differ from each other and from other programs. In Section 4, we compare the performance of `nauty` and `Traces` with `Bliss`, `saucy` and `conauto` when applied to a variety of families of graphs ranging from those traditionally easy to the most difficult known. Although none of the programs is the fastest in all cases, we will see that `nauty` is generally the fastest for small graphs and some easier families, while `Traces` is better, sometimes in dramatic fashion, for most of the difficult graph families.

For the software and its manual see McKay and Piperno (2012a, 2012b).

2. Generic algorithm

In this section, we give formal definitions of colourings (partitions), invariants, and group actions. We then define the search tree which is at the heart of most recent graph isomorphism algorithms and explain how it enables computation of automorphism groups and canonical forms. This section is intended to be a self-contained introduction to the overall strategy. It does not contain new features, though it is more complete than previous expositions.

Let $\mathcal{G} = \mathcal{G}_n$ denote the set of graphs with vertex set $V = \{1, 2, \ldots, n\}$.

2.1. Colourings

A *colouring* of V (or of $G \in \mathcal{G}$) is a surjective function π from V onto $\{1, 2, \ldots, k\}$ for some k. The number of colours, i.e. k, is denoted by $|\pi|$. A *cell* of π is the set of vertices with some given colour; that is, $\pi^{-1}(j)$ for some j with $1 \leqslant j \leqslant |\pi|$. A *discrete colouring* is a colouring in which each cell is a singleton, in which case $|\pi| = n$. Note that a discrete colouring is a permutation of V.

If π, π' are colourings, then π' is *finer than or equal to* π, written $\pi' \preccurlyeq \pi$, if $\pi(v) < \pi(w) \Rightarrow \pi'(v) < \pi'(w)$ for all $v, w \in V$. (This implies that each cell of π' is a subset of a cell of π, but the converse is not true.)

Since a colouring partitions V into cells, it is frequently called a *partition*. However, note that the colours come in a particular order and this matters when defining concepts like "finer".

A pair (G, π), where π is a colouring of G, is called a *coloured graph*.

2.2. Group actions and isomorphisms

Let S_n denote the symmetric group acting on V. We indicate the action of elements of S_n by exponentiation. That is, for $v \in V$ and $g \in S_n$, v^g is the image of v under g. The same notation indicates the induced action on complex structures derived from V; in particular:

(a) If $W \subseteq V$, then $W^g = \{w^g : w \in W\}$, and similarly for sequences.
(b) If $G \in \mathcal{G}$, then $G^g \in \mathcal{G}$ has v^g adjacent to w^g exactly when v and w are adjacent in G. As a special case, a discrete colouring π is a permutation on V so we can write G^π.
(c) If π is a colouring of V, then π^g is the colouring with $\pi^g(v^g) = \pi(v)$ for each $v \in V$.
(d) If (G, π) is a coloured graph, then $(G, \pi)^g = (G^g, \pi^g)$.

Two coloured graphs $(G, \pi), (G', \pi')$ are *isomorphic* if there is $g \in S_n$ such that $(G', \pi') = (G, \pi)^g$, in which case we write $(G, \pi) \cong (G', \pi')$. Such a g is called an *isomorphism*. The *automorphism group* $\mathrm{Aut}(G, \pi)$ is the group of isomorphisms of the coloured graph (G, π) to itself; that is,

$$\mathrm{Aut}(G, \pi) = \left\{ g \in S_n : (G, \pi)^g = (G, \pi) \right\}.$$

A *canonical form* is a function

$$C : \mathcal{G} \times \Pi \to \mathcal{G} \times \Pi$$

such that, for all $G \in \mathcal{G}$, $\pi \in \Pi$ and $g \in S_n$,

(C1) $C(G, \pi) \cong (G, \pi)$,
(C2) $C(G^g, \pi^g) = C(G, \pi)$.

In other words, it assigns to each coloured graph an isomorphic coloured graph that is a unique representative of its isomorphism class. It follows from the definition that $(G, \pi) \cong (G', \pi') \Leftrightarrow C(G, \pi) = C(G', \pi')$.

Property (C2) is an important property that must be satisfied by many functions we define. It says that if the elements of V appearing in the inputs to the function are renamed in some manner, the elements of V appearing in the function value are renamed in the same manner. We call this *label-invariance*.

2.3. Search tree

Now we define a rooted tree whose nodes correspond to sequences of vertices, with the empty sequence at the root of the tree. The sequences become longer as we move down the tree. Each sequence corresponds to a colouring of the graph obtained by giving the vertices in the sequence unique colours then inferring in a controlled fashion a colouring of the other vertices. Leaves of the tree correspond to sequences for which the derived colouring is discrete.

To formally define the tree, we first define a "refinement function" that specifies the colouring that corresponds to a sequence. Let V^* denote the set of finite sequences of vertices. For $v \in V^*$, $|v|$ denotes the number of components of v. If $v = (v_1, \ldots, v_k) \in V^*$ and $w \in V$, then $v \parallel w$ denotes $(v_1, \ldots, v_k, w)$. Furthermore, for $0 \leqslant s \leqslant k$, $[v]_s = (v_1, \ldots, v_s)$. The ordering $\leqslant$ on finite sequences is the lexicographic order: If $v = (v_1, \ldots, v_k)$ and $v' = (v'_1, \ldots, v'_\ell)$, then $v \leqslant v'$ if v is a prefix of v' or there is some $j \leqslant \min\{k, \ell\}$ such that $v_i = v'_i$ for $i < j$ and $v_j < v'_j$.

A *refinement function* is a function

$$R : \mathcal{G} \times \Pi \times V^* \to \Pi$$

such that for any $G \in \mathcal{G}$, $\pi \in \Pi$ and $v \in V^*$,

(R1) $R(G, \pi, v) \preccurlyeq \pi$;
(R2) if $v \in v$, then $\{v\}$ is a cell of $R(G, \pi, v)$;
(R3) for any $g \in S_n$, we have $R(G^g, \pi^g, v^g) = R(G, \pi, v)^g$.

To complete the definition of the tree, we need to specify what are the children of each node. We do this by choosing one non-singleton cell of the colouring, called the *target cell*, and appending an element of it to the sequence.

A *target cell selector* chooses a non-singleton cell of a colouring, if there is one. Formally, it is a function

$$T : \mathcal{G} \times \Pi \times V^* \to 2^V$$

such that for any $\pi_0 \in \Pi$, $G \in \mathcal{G}$ and $v \in V^*$,

(T1) if $R(G, \pi_0, v)$ is discrete, then $T(G, \pi_0, v) = \emptyset$;
(T2) if $R(G, \pi_0, v)$ is not discrete, then $T(G, \pi_0, v)$ is a non-singleton cell of $R(G, \pi_0, v)$;
(T3) for any $g \in S_n$, we have $T(G^g, \pi^g, v^g) = T(G, \pi, v)^g$.

Now we can define the *search tree* $\mathcal{T}(G, \pi_0)$ depending on an initially-specified coloured graph (G, π_0). The nodes of the tree are elements of V^*.

(a) The root of $\mathcal{T}(G, \pi_0)$ is the empty sequence $(\,)$.
(b) If v is a node of $\mathcal{T}(G, \pi_0)$, let $W = T(G, \pi_0, v)$. Then the children of π are

$$\{v \parallel w : w \in W\}.$$

This definition implies by (T2) that a node v of $\mathcal{T}(G, \pi_0)$ is a leaf iff $R(G, \pi_0, v)$ is discrete.

For any node v of $\mathcal{T}(G, \pi_0)$, define $\mathcal{T}(G, \pi_0, v)$ to be the subtree of $\mathcal{T}(G, \pi_0)$ consisting of v and all its descendants. The following lemmas are easily derived using induction from the definition of the search tree and the properties of the functions R, T and I.

Lemma 1. *For any $G \in \mathcal{G}$, $\pi_0 \in \Pi$, $g \in S_n$, we have $\mathcal{T}(G^g, \pi_0^g) = \mathcal{T}(G, \pi_0)^g$.*

Proof. Let $v = (v_1, \ldots, v_k)$ be a node of $\mathcal{T}(G, \pi_0)$. It is easily proved by induction on s that $[v^g]_s$ is a node of $\mathcal{T}(G^g, \pi_0^g)$ for $0 \leqslant s \leqslant k$. Therefore, $\mathcal{T}(G, \pi_0)^g \subseteq \mathcal{T}(G^g, \pi_0^g)$. The reverse inclusion follows on considering g^{-1} instead, so the lemma is proved. $\square$

Corollary 2. *Let v be a node of $\mathcal{T}(G, \pi_0)$ and let $g \in \mathrm{Aut}(G, \pi_0)$. Then v^g is a node of $\mathcal{T}(G, \pi_0)$ and $\mathcal{T}(G, \pi_0, v^g) = \mathcal{T}(G, \pi_0, v)^g$.*

Proof. This follows from Lemma 1 on noticing that $(G, \pi_0)^g = (G, \pi)$ if $g \in \mathrm{Aut}(G, \pi_0)$. □

Lemma 3. *Let v be a node of $\mathcal{T}(G, \pi_0)$ and let $\pi = R(G, \pi_0, v)$. Then $\mathrm{Aut}(G, \pi)$ is the point-wise stabilizer of v in $\mathrm{Aut}(G, \pi_0)$.*

Proof. By condition (R2), every element of $\mathrm{Aut}(G, \pi)$ stabilizes v. Conversely, suppose $g \in \mathrm{Aut}(G, \pi_0)$ stabilizes v. Then by (R3), $\pi^g = R(G, \pi_0, v)^g = R(G, \pi_0, v) = \pi$, so $g \in \mathrm{Aut}(G, \pi)$. □

2.4. Automorphisms and canonical forms

Now we describe how the search tree $\mathcal{T}(G, \pi_0)$, defined as in the previous subsection, can be used to compute $\mathrm{Aut}(G, \pi_0)$ and a canonical form.

Let Ω be some totally ordered set. A *node invariant* is a function

$$\phi : \mathcal{G} \times \Pi \times V^* \to \Omega,$$

such that for any $\pi_0 \in \Pi$, $G \in \mathcal{G}$, and distinct $v, v' \in \mathcal{T}(G, \pi_0)$,

(ϕ1) if $|v| = |v'|$ and $\phi(G, \pi_0, v) < \phi(G, \pi_0, v')$, then for every leaf $v_1 \in \mathcal{T}(G, \pi_0, v)$ and leaf $v_1' \in \mathcal{T}(G, \pi_0, v')$ we have $\phi(G, \pi_0, v_1) < \phi(G, \pi_0, v_1')$;

(ϕ2) if $\pi = R(G, \pi_0, v)$ and $\pi' = R(G, \pi_0, v')$ are discrete, then $\phi(G, \pi_0, v) = \phi(G, \pi_0, v') \Leftrightarrow G^\pi = G^{\pi'}$ (note that the last relation is equality, not isomorphism);

(ϕ3) for any $g \in S_n$, we have $\phi(G^g, \pi_0^g, v^g) = \phi(G, \pi_0, v)$.

Say that leaves v, v' are *equivalent* if $\phi(G, \pi_0, v) = \phi(G, \pi_0, v')$. If this is the case, there is a unique $g \in \mathrm{Aut}(G, \pi_0)$ such that $v^g = v'$, namely $g = R(G, \pi_0, v')R(G, \pi_0, v)^{-1}$. (Recall that $R(G, \pi_0, v)$ is a permutation if v is a leaf.)

According to Corollary 2, if v is a leaf of $\mathcal{T}(G, \pi_0)$, then so is v^g for any $g \in \mathrm{Aut}(G, \pi_0)$. Moreover, by the properties of ϕ, these leaves (over $g \in \mathrm{Aut}(G, \pi_0)$) have the same value of ϕ and no other leaf has that value. Consequently, for any leaf v,

$$\mathrm{Aut}(G, \pi_0) = \big\{ R(G, \pi_0, v')R(G, \pi_0, v)^{-1} \colon v' \text{ is a leaf of } \mathcal{T}(G, \pi_0)$$
$$\text{with } \phi(G, \pi_0, v') = \phi(G, \pi_0, v) \big\}.$$

To define a canonical form, let

$$\phi^*(G, \pi_0) = \max\big\{ \phi(G, \pi_0, v) \colon v \text{ is a leaf of } \mathcal{T}(G, \pi_0) \big\},$$

and let v^* be any leaf of $\mathcal{T}(G, \pi_0)$ that achieves the maximum. Now define $C(G, \pi_0) = (G, \pi_0)^{R(G, \pi_0, v^*)}$. By the properties of ϕ, $C(G, \pi_0)$ thus defined is independent of the choice of v^*. In particular, we have:

Lemma 4. *The function*

$$C : \mathcal{G} \times \Pi \to \mathcal{G} \times \Pi$$

as just defined is a canonical form.

These observations provide an algorithm for computing $\mathrm{Aut}(G, \pi_0)$ and $C(G, \pi_0)$, once we have defined T and ϕ. In practice it is not of much use, since the search tree can be extremely large and the group is found element by element rather than as a set of generators. However, in practice we can dramatically improve the performance by judicious pruning of the tree.

When we refer to a *leaf* of $\mathcal{T}(G, \pi_0)$, we always mean a node ν of $\mathcal{T}(G, \pi_0)$ for which $R(G, \pi_0, \nu)$ is discrete, even if our pruning of the tree results in additional nodes having no children.

We define three types of pruning operation on the search tree.

(A) Suppose ν, ν' are distinct nodes of $\mathcal{T}(G, \pi_0)$ with $|\nu| = |\nu|'$ and $\phi(G, \pi_0, \nu) > \phi(G, \pi_0, \nu')$. Operation $P_A(\nu, \nu')$ is to remove $\mathcal{T}(G, \pi_0, \nu')$.
(B) Suppose ν, ν' are distinct nodes of $\mathcal{T}(G, \pi_0)$ with $|\nu| = |\nu|'$ and $\phi(G, \pi_0, \nu) \neq \phi(G, \pi_0, \nu')$. Operation $P_B(\nu, \nu')$ is to remove $\mathcal{T}(G, \pi_0, \nu')$.
(C) Suppose $g \in \mathrm{Aut}(G, \pi_0)$ and suppose $\nu < \nu'$ are nodes of $\mathcal{T}(G, \pi_0)$ such that $\nu^g = \nu'$. Operation $P_C(\nu, g)$ is to remove $\mathcal{T}(G, \pi_0, \nu')$.

Theorem 5. *Consider any $G \in \mathcal{G}$ and $\pi_0 \in \Pi$.*

(a) *Suppose any sequence of operations of the form $P_A(\nu, \nu')$ or $P_C(\nu, g)$ are performed. Then there remains at least one leaf ν_1 with $\phi(G, \pi_0, \nu_1) = \phi^*(G, \pi_0)$.*
(b) *Let ν_0 be some fixed leaf of $\mathcal{T}(G, \pi_0)$. Suppose any sequence of operations of the form $P_B(\nu, \nu')$ or $P_C(\nu'', g)$ are performed, where $\phi(G, \pi_0, \nu'') \neq \phi(G, \pi_0, [\nu_0]_{|\nu''|})$. Let $g_1, \ldots, g_k$ be the automorphisms used in the operations P_C that were performed, and let*

$$A = \left\{ g \in \mathrm{Aut}(G, \pi_0) \colon \nu_0^g \text{ is a remaining leaf} \right\}.$$

Then $\mathrm{Aut}(G, \pi_0)$ is generated by $\{g_1, \ldots, g_k\} \cup A$.

Proof. To prove claim (a), note that the lexicographically least leaf ν_1 with $\phi(G, \pi_0, \nu_1) = \phi^*(G, \pi_0)$ is never removed.

For claim (b), note that the lexicographically least leaf ν_1 equivalent to ν_0 is not removed by the allowed operations. Choose an arbitrary $g \in \mathrm{Aut}(G, \pi_0)$. By Corollary 2, ν_0^g is a leaf of $\mathcal{T}(G, \pi_0)$. If it has been removed, that must have been by some $P_C(\nu'', g_i)$ with $\nu'' < \nu^g$, since operation $P_B(\nu, \nu')$ only removes leaves inequivalent to ν_0. Note that $\nu_0^{gg_i^{-1}}$ is a leaf descended from ν'' and $\nu_0^{gg_i^{-1}} < \nu_0^g$. If $\nu_0^{gg_i^{-1}}$, has been removed, that must have been due to some $P_C(\nu''', g_j)$ with $\nu''' < \nu_0^{gg_i^{-1}}$, so consider the leaf $\nu_0^{gg_i^{-1}g_j^{-1}} < \nu_0^{gg_i^{-1}}$. Continuing in this way we must eventually find a leaf that has not been removed, since the leaf ν_1 is still present. That is, there is some $h \in \langle g_1, \ldots, g_k \rangle$ such that leaf ν_0^{gh} has not been removed. This shows that g belongs to the group generated by $\{g_1, \ldots, g_k\} \cup A$, as we wished to prove. Part (b) for operation C was first published by McKay (1978a). □

The theorem leaves unspecified where the automorphisms for $P_C(\nu, g)$ operations come from. They might be provided in advance, detected by noticing two leaves are equivalent, or otherwise. This is discussed in the following section.

3. Implementation strategies

In this section, we describe two implementations of the generic algorithm, which are distributed together as `nauty` and `Traces` (McKay and Piperno, 2012a).

3.1. Refinement

Let $G \in \mathcal{G}$. A colouring of G is called *equitable* if any two vertices of the same colour are adjacent to the same number of vertices of each colour.[1] (See Fig. 3 for a simple example.)

---

[1] Unfortunately, "equitable colouring" also has another meaning in graph theory. More commonly, our concept is called an *equitable*, *regular* or *stable* partition.
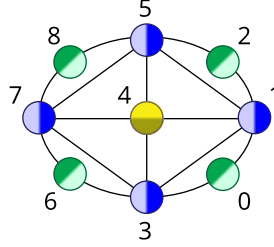
Fig. 1. Example of an equitable colouring.

It is well known that for every colouring π there is a coarsest equitable colouring π' such that $\pi' \preccurlyeq \pi$, and that π' is unique up to the order of its cells. An algorithm for computing π' appears in McKay (1980). We summarize it in Algorithm 1.

Data: π is the input colouring and α is a sequence of some cells of π
Result: the final value of π is the output colouring

while α *is not empty* **and** π *is not discrete* **do**
    Remove some element W from α.
    for *each cell X of π* **do**
        Let $X_1, \ldots, X_k$ be the fragments of X distinguished according
          to the number of edges from each vertex to W.
        Replace X by $X_1, \ldots, X_k$ in π.
        if $X \in \alpha$ **then**
            Replace X by $X_1, \ldots, X_k$ in α.
        else
            Add all but one of the largest of $X_1, \ldots, X_k$ to α.
        end
    end
end

Algorithm 1: Refinement algorithm $F(G, \pi, \alpha)$.

Let $F(G, \pi, \alpha)$ be the function defined by Algorithm 1, which we assume to be implemented in a label-invariant manner. Now define the function

$$I : \Pi \times V \to \Pi,$$

such that, if v is a vertex in a non-singleton cell of π and $\pi' = I(\pi, v)$, then for $w \in V$,

$$\pi'(w) = \begin{cases} \pi(w), & \text{if } \pi(w) < \pi(v) \text{ or } w = v; \\ \pi(w) + 1, & \text{otherwise.} \end{cases}$$

We say that the vertex v is *individualized* in π (see Fig. 2). We see that $I(\pi, v)$ differs from π in that a unique colour has been given to vertex v. Now we can define a refinement function. For a sequence of vertices $v_1, v_2, \ldots$, define

$$R(G, \pi_0, ()) = F(G, \pi_0, \text{ a list of all the cells of } \pi_0),$$
$$R(G, \pi_0, (v_1)) = F(G, I(R(G, \pi_0, ()), v_1), (\{v_1\})),$$
$$R(G, \pi_0, (v_1, v_2)) = F(G, I(R(G, \pi_0, (v_1)), v_2), (\{v_2\})),$$
$$R(G, \pi_0, (v_1, v_2, v_3)) = F(G, I(R(G, \pi_0, (v_1, v_2)), v_3), (\{v_3\})),$$

and so on. According to Theorem 2.7 and Lemma 2.8 of McKay (1980), R satisfies (R1)–(R3) and, moreover, $R(G, \pi_0, v)$ is equitable.

In practice most of the execution time of the whole algorithm is devoted to refining colourings, so the implementation is critical. Since the splitting of X into fragments can be coded more efficiently if
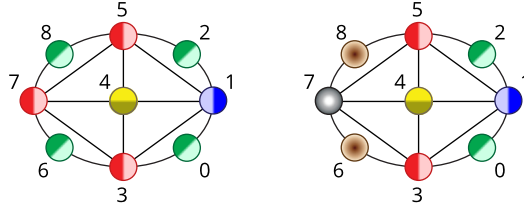
Fig. 2. Individualization of vertex 1 and subsequent refinement.

W is a singleton, we have found it advantageous to choose singletons out of α in preference to larger cells.

While the function R defined above is sufficient for many graphs, there are difficult classes (see Section 4) for which it does not adequately separate inequivalent vertices. Regular graphs are the simplest example, since the colouring with only one colour is equitable. A simple way of doing better is to count the number of triangles incident to each vertex. In choosing such a strategy, there is a trade-off between the partitioning power and the cost. nauty provides a small library of stronger partitioning functions, some of them designed for particular classes of difficult graphs. The improvement in performance can be very dramatic. On the other hand, choice of which partitioning function to employ is left to the user and requires skill, which is not very satisfactory.

Traces has a different approach to this problem, as we will see in Section 3.3.

3.2. Target cell selection

The choice of target cell has a significant effect on the shape of the search tree, and thus on performance. A small target cell may perhaps have a greater chance of being an orbit of the group which fixes the current stabilizer sequence. For this reason, McKay (1980) recommended using the first smallest non-singleton cell. However, Kocay (1996) found (without realizing it) that using the first non-singleton cell regardless of size was better for most test cases, as confirmed by Kirk (1985). The current version of nauty has two strategies. One is to use the first non-singleton cell, and the other is to choose the first cell which is joined in a non-trivial fashion to the largest number of cells, where a non-trivial join between two cells means that there is more than 0 edges and less than the maximum possible.

Traces, on the other hand prefers large target cells, as they tend to make the tree less deep. A strategy developed by experiment is to use the first largest non-singleton cell that is a subset of the target cell in the parent node. If there are no such non-singleton cells, the target cell in the grandparent node is used, and so on, with the first largest cell overall being used if no ancestor has a suitable target cell.

3.3. Node invariants

Information useful for computing node invariants can come from two related sources. At each node ν there is a colouring $R(G, \pi_0, \nu)$ and we can use properties of this colouring such as the number and size of the cells, as well as combinatorial properties of the coloured graph. Another source is the intermediate states of the computation of a colouring from that of the parent node, such as the order, position and size of the cells produced by the refinement procedure and various counts of edges that are determined during the computation.

If $f(\nu)$ is some function of this information, computed during the computation of $R(G, \pi_0, \nu)$ and from the resulting coloured graph, the vector $(f([\nu]_0), f([\nu]_1), \ldots, f(\nu))$, with lexicographic ordering, satisfies conditions ($\phi 1$) and ($\phi 3$) for a node invariant. If ν is a leaf, we can append G^π, where π is the discrete colouring $R(G, \pi_0, \nu)$, to the vector so as to satisfy ($\phi 2$) as well.

In nauty, the value of $f(\nu)$ is an integer, and the pruning rules are applied as each node is computed. Traces introduced a major improvement, defining each $f(\nu)$ as a vector itself. The primary components of $f(\nu)$ are the sizes and positions of the cells in the order that they are created by

the refinement procedure. $\phi(G, \pi_0, \nu)$ thus becomes a vector of vectors, called the *trace* (and hence the name "Traces"). The advantage is that it often enables the comparison of $f(\nu)$ and $f(\nu')$ to be made while the computation of ν' is only partly complete. A limited form of this idea appeared in Bliss (Junttila and Kaski, 2007), and also appears in a recent version of saucy (Darga et al., 2008). For many difficult graph families, only a fraction of all refinement operations need to be completed. A practical consequence is that the stronger refinements used by nauty (see Section 3.1) are rarely needed. This makes good performance in Traces less dependent on user expertise than is the case with nauty.

If π is an equitable colouring of a graph G, we can define the *quotient graph* $Q(G, \pi)$ as follows. The vertices of $Q(G, \pi)$ are the cells of π, labelled with the cell number and size. For any two cells $C_1, C_2 \in \pi$, possibly equal, the corresponding vertices of $Q(G, \pi)$ are joined by an edge labelled with the number of edges of G between C_1 and C_2.

The node invariant $\phi(G, \pi_0, \nu)$ computed by Traces, and also by nauty if the standard refinement process Algorithm 1 is used, is a deterministic function of the sequence of quotient graphs $Q(G, R(G, \pi_0, [\nu]_i))$ for $i = 0, \ldots, |\nu|$. We could in fact use that sequence of quotient graphs, but that would be expensive in both time and space. Our experience is that the information we do use, which is essentially information about the quotient graphs collected during the refinement process, rarely has less pruning power than the quotient graphs themselves would have.

3.4. Strategies for tree generation

Now we have described the search tree $\mathcal{T}(G, \pi_0)$ as defined by nauty and Traces. In general only a fraction of the search tree is actually generated, since the pruning rules of Section 2.4 are applied (see Fig. 3). These pruning rules utilise both node invariants, as described in Section 3.3, and automorphisms, which are mainly discovered by noticing that two discrete colourings give the same coloured graph. Now we will describe the order of generation of the tree, which is fundamentally different for nauty and Traces.

In nauty, the tree is generated in depth-first order. The lexicographically least leaf ν_1 is kept. If the canonical labelling is sought (rather than just the automorphism group), the leaf ν^* with the greatest invariant discovered so far is also kept. A non-leaf node ν is pruned if neither $\phi(G, \pi_0, \nu) = \phi(G, \pi_0, [\nu_1]_{|\nu|})$ or $\phi(G, \pi_0, \nu) \geqslant \phi(G, \pi_0, [\nu^*]_{|\nu|})$. Such operations have both type $P_A(\nu^*, \nu)$ and $P_B(\nu_1, \nu)$, so Theorem 5 applies. Automorphisms are found by discovering leaves equivalent to ν_1 or ν^*, and also to a limited extent from the properties of equitable colourings. Pruning operation P_C is performed wherever possible, as high in the tree as possible (i.e., at the children of the nearest common ancestor of the two leaves found to be equivalent).

Until a recent version of nauty, the only automorphisms used for pruning operation P_C were those directly discovered, without any attempt to compose them. Now we use the random Schreier method (Seress, 2003) to perform more complete pruning. By Lemma 3, nodes $\nu \parallel \nu_1$ and $\nu \parallel \nu_2$ are equivalent if ν_1, ν_2 belong to the same orbit of the point-wise stabilizer of ν in Γ, where Γ is the group generated by the automorphisms found so far. This stabilizer could be computed with a deterministic algorithm as proposed by Kocay (1996) and Butler and Lam (1985), but we have found the random Schreier method (Seress, 2003) to be more efficient and it doesn't matter if occasionally (due to its probabilistic nature) it computes smaller orbits. The usefulness of this for nauty's efficiency with some classes of difficult graph was demonstrated in 1985 by Kirk (1985) but only made it into the distributed edition of nauty in 2011.

Nauty's basic depth-first approach is also followed by Bliss and saucy. However, Traces introduces an entirely different order of generating the tree. Some variations are possible but we will first describe the normative method, which is based on a *breadth*-first search. Define *level k* to be the set of nodes ν with $|\nu| = k$. In the *k*-th phase, Traces computes those nodes ν in level k which have the greatest value of $\phi(G, \pi_0, \nu)$ on that level. By property ($\phi1$), such nodes are the children of the nodes with greatest ϕ on the previous level, so no backtracking is needed. This order of tree generation has the big advantage that pruning operation P_A is used to the maximum possible extent.

As mentioned in Section 3.3, the node invariant $\phi(G, \pi_0, \nu)$ is computed incrementally during the refinement process, so that pruning operation P_A can often be applied when the refinement is only partly complete.
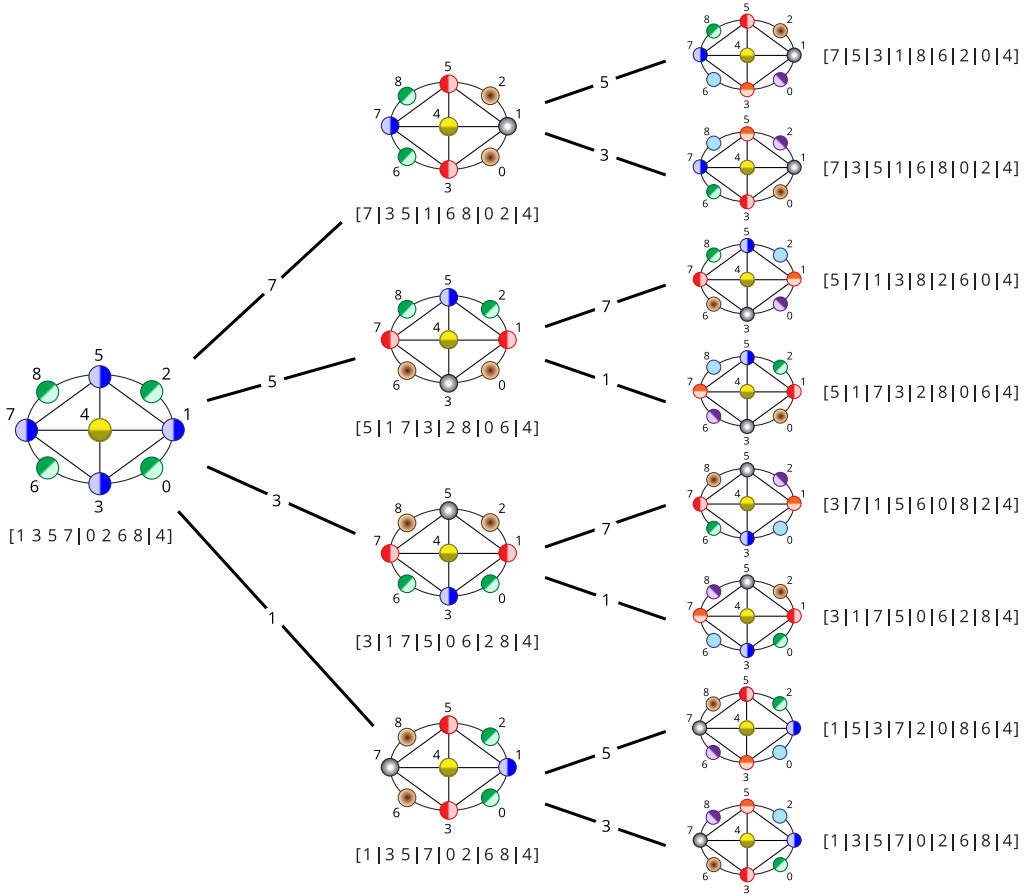
Fig. 3. Example of a search tree for the graph of Fig. 1.

An apparent disadvantage of breadth-first order is that pruning by automorphisms (operation P_C) is only possible when automorphisms are known, which in general requires leaves of the tree. To remedy this problem, for every node a single path, called an "experimental path", is generated from that node down to a leaf of the tree. Automorphisms are found by comparing the labelled graphs that correspond to those leaves, with the value of $\phi(G, \pi_0, \nu)$ at the leaf being used to avoid most unnecessary comparisons. We have found experimentally that generating experimental paths randomly tends to find automorphisms that generate larger subgroups, so that the group requires fewer generators altogether and more of the group is available early for pruning.

The group generated by the automorphisms found so far is maintained using the random Schreier method. Some features of the Schreier method are turned on and off in Traces when it is possible to heuristically infer their computational weight.

Fig. 4 continues the example of Fig. 3, showing the portion of the search tree traversed by nauty (left) and Traces (right). Node labels indicate the order in which nodes are visited, and edge labels indicate which vertex is individualized. During its backtrack search, nauty stores the first leaf (2) for comparison with subsequent leaves. Leaves 2 and 3 provide the generator $g_1^n = (0\,2)(3\,5)(6\,8)$, which for example allows pruning of the greyed subtree formed by individualizing vertex 5 at the root. Traces executes a breadth-first search, storing with each visited node the discrete partition obtained by a randomly chosen experimental path (shown by dashed arrows). After processing node 2 of the
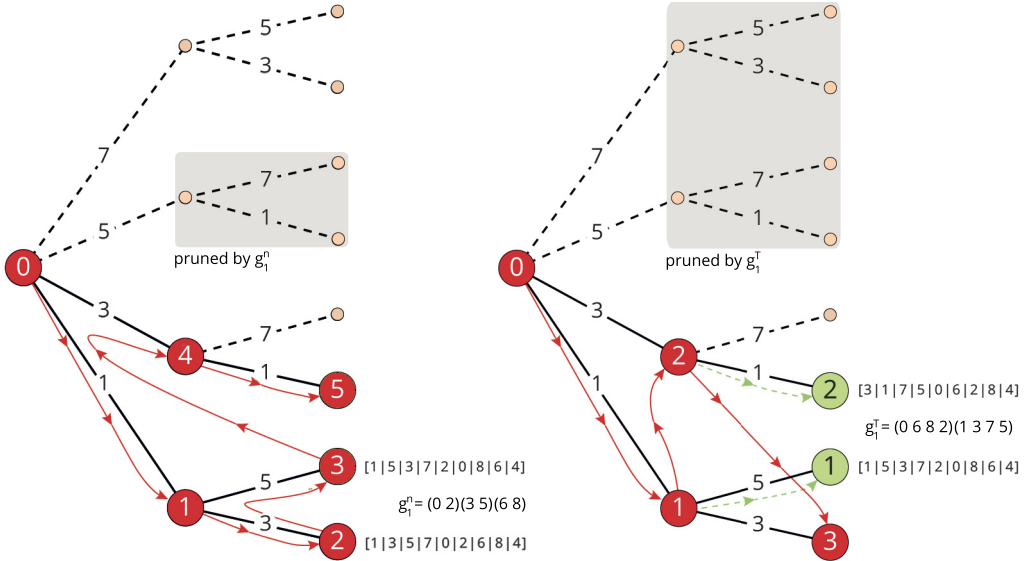
Fig. 4. Search tree order for `nauty` (left) and `Traces` (right).

tree, the experimental leaves 1 and 2 are compared, revealing the generator $g_1^T = (0\,6\,8\,2)(1\,3\,7\,5)$, which allows for pruning the greyed subtrees formed by individualizing vertices 5 and 7 at the root.

3.5. Detection of automorphisms

The primary way that automorphisms are detected, in all the programs under consideration, is to compare the labelled graphs corresponding to leaves of the search tree as described above.

An important innovation of `saucy` (Darga et al., 2008) was to detect some types of automorphism higher in the tree. Suppose that π, π' are equitable colourings with the same number of vertices of each colour. Any automorphism of (G, π_0) that takes π onto π' has known action on the fixed vertices of π: it maps them to the fixed vertices of π' with the same colours. In some cases that `saucy` can detect very quickly, this partial mapping is an automorphism when extended as the identity on the non-fixed vertices. This happens, for example, when a component of G is completely fixed by two different but equivalent stabilization sequences. This is one of the main reasons `saucy` can be very fast on graphs with many automorphisms that move few vertices.

`Traces` extends this idea by finding many automorphisms that do not require the identity mapping on the non-fixed vertices. It does this by a heuristic that extends the mapping from the fixed vertices to the non-fixed vertices, which is applied in certain situations where experience shows it is more likely to succeed.

When `Traces` is only looking for the automorphism group, and not for a canonical labelling, it employs another strategy which is sometimes much faster. Suppose that while generating the nodes on some level L, it notices (during experimental path generation) that one of them, say v, has a child which is discrete. At this point, `Traces` determines and keeps all the discrete children of v (modulo the usual automorphism pruning). Now, for all nodes v' on level L, a single discrete child v'' is found, if any, and an automorphism is discovered if it is equivalent to any child of v. The validity of this approach follows from Theorem 5 with the role of v_0 played by the first discrete child of v.

Fig. 5 (left) shows the whole tree up to level $L + 1$, where a node labelled by X represents a discrete partition corresponding to labelled graph X, while an unlabelled (and smaller) node stands for a non-discrete partition. Fig. 5 (center) shows the part of the tree which is traversed by `Traces` during the search for a canonical labelling. Only the best leaf is kept for comparison with subsequent discrete partitions.
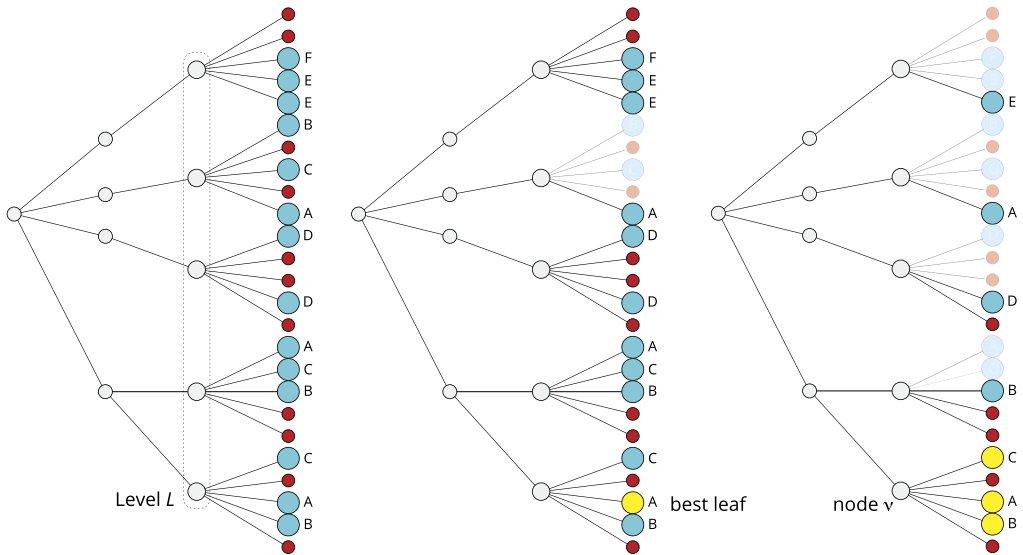
Fig. 5. `Traces` search strategies for canonical labelling or automorphism group.

Fig. 5 (right) shows the part of the tree which is traversed by `Traces` during an automorphism group computation. All the discrete children of ν are kept for comparison with subsequent discrete partitions. When the first discrete partition is found as a child of a node ν' at level L, either it has the same labelled graph as one of those stored, or the whole subtree rooted at ν' has no leaf with one of the stored graphs. In the first case, an automorphism is found. In both cases, the computation is resumed from the next node at level L.

`Traces` also allows the user to provide known automorphisms, which are often available when graphs of algebraic origin are being studied.

3.6. Low degree vertices

Graphs in some applications, such as constraint satisfaction problems described by Darga et al. (2004) have many small components with vertices of low degree, vertices with common neighborhoods, and so on. `Saucy` handles them efficiently by a refinement procedure tuned to this situation plus early detection of sparse automorphisms. `Traces` employs another method. Recall that after the first refinement vertices with equal colours also have equal degrees. The target cell selector never selects cells containing vertices of degree 0, 1, 2 or $n - 1$, and nodes whose non-trivial cells are only of those degrees are not expanded further. Special-purpose code then produces generators for the automorphism group fixed by the node and, if necessary, a unique discrete colouring that refines the node.

This technique is quite successful. However, in our opinion, graphs of this type ought to be handled by preprocessing. For example, sets of vertices with the same neighborhoods ought to be replaced by single vertices with a colour that encodes the multiplicity. All tree-like appendages, long paths of degree 2 vertices, and similar easy subgraphs, could be efficiently factored out in this manner. A future release of `Traces` will have this feature.

4. Performance

In Figs. 6–10, we present some comparisons between programs for a variety of graphs ranging from very easy to very difficult. We made an effort to include graphs that are easy and difficult for each of the programs tested.
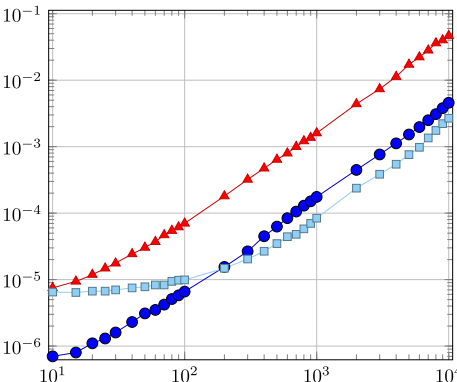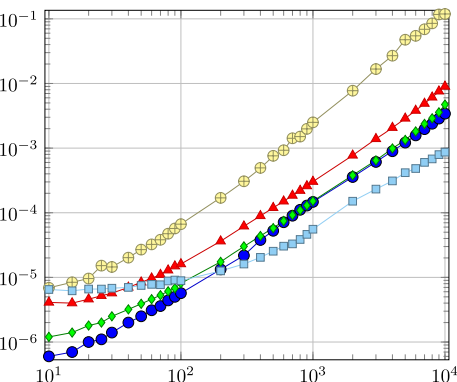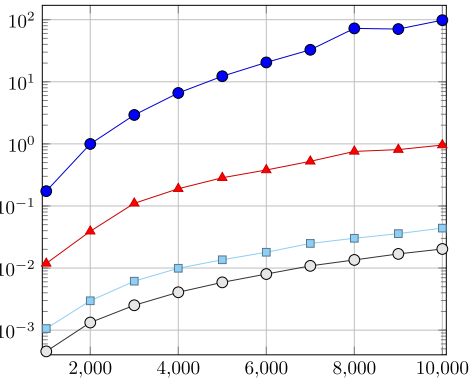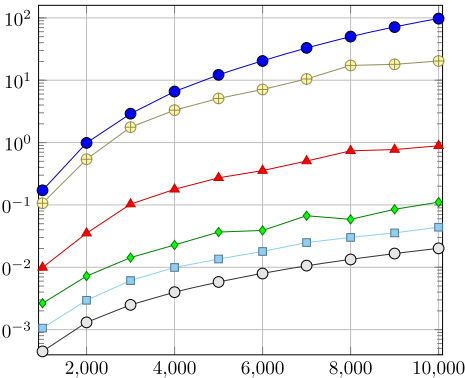
Automorphism group

Canonical label

Random graphs with $p = \frac{1}{2}$
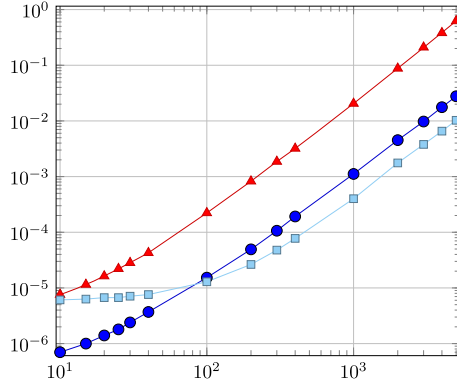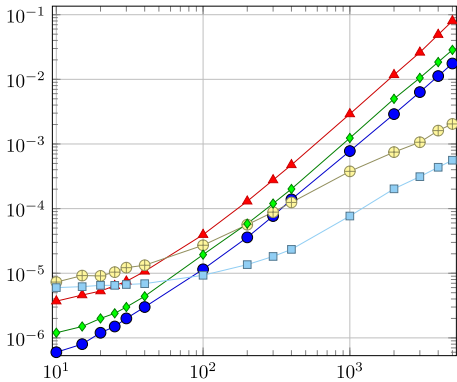
Random graphs with $p = n^{-1/2}$
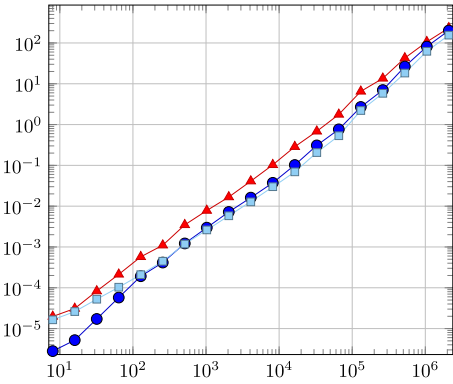
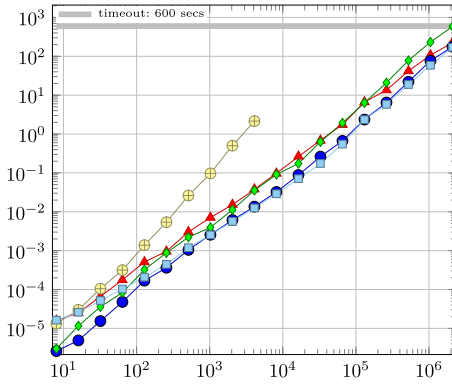Random cubic graphs (`nauty` invariant $distances(2)$)

▲ Bliss   ◆ saucy   ⊕ conauto   ● nauty   ○ nauty with invariant   ■ Traces

Fig. 6. Performance comparison (horizontal: number of vertices; vertical: time in seconds).
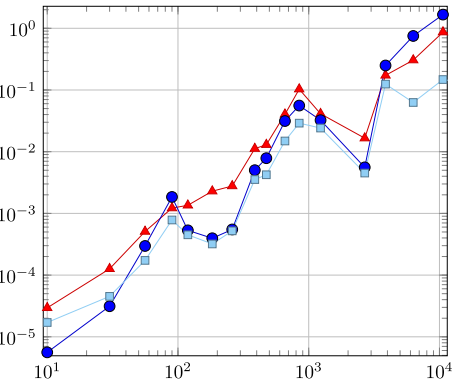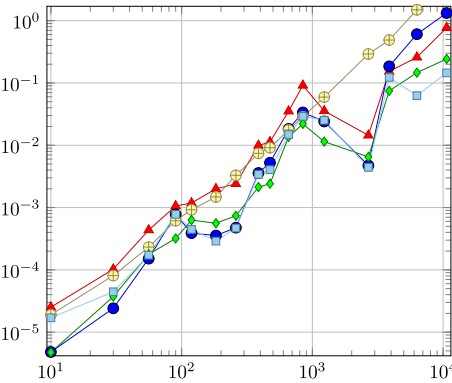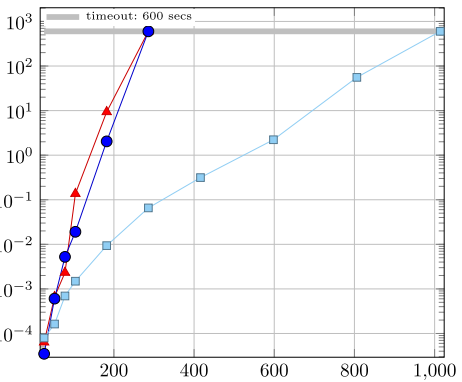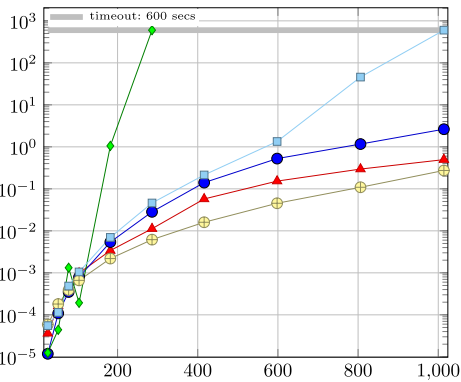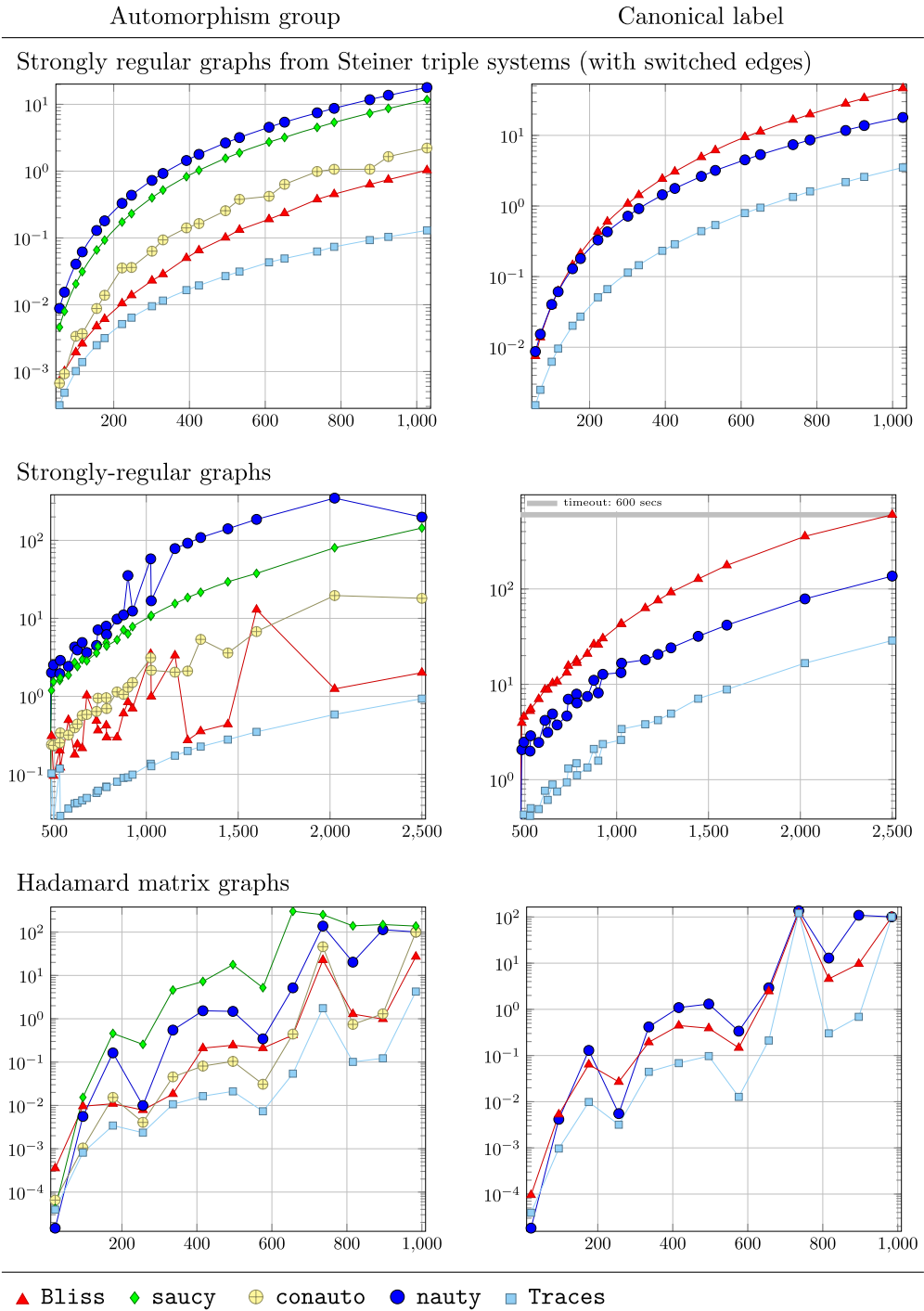
Automorphism group

Canonical label

Hypercubes (vertex-transitive)

Miscellaneous vertex-transitive graphs

(Non-disjoint) union of tripartite graphs
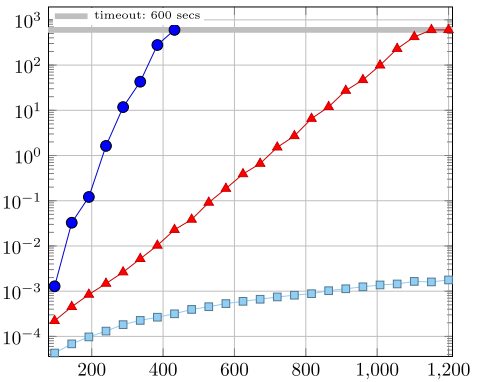
▲ Bliss   ◆ saucy   ⊕ conauto   ● nauty   ■ Traces

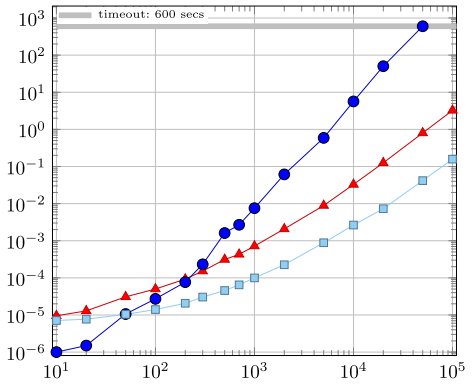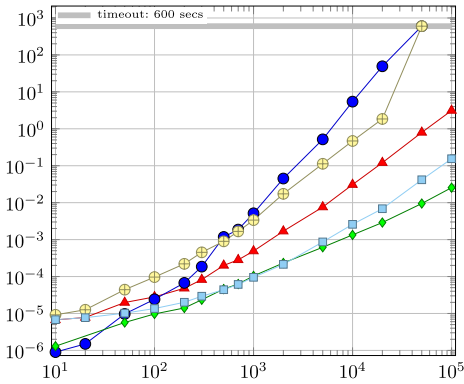**Fig. 7.** Performance comparison (horizontal: number of vertices; vertical: time in seconds).

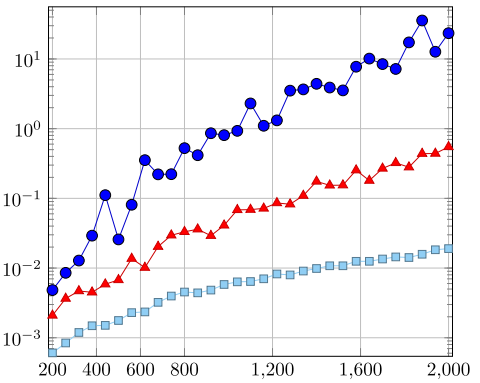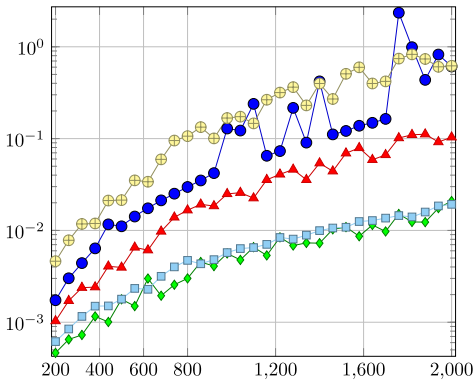**Fig. 8.** Performance comparison (horizontal: number of vertices; vertical: time in seconds).

Fig. 9. Performance comparison (horizontal: number of vertices; vertical: time in seconds).

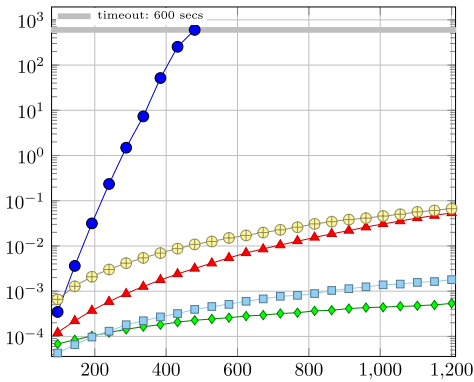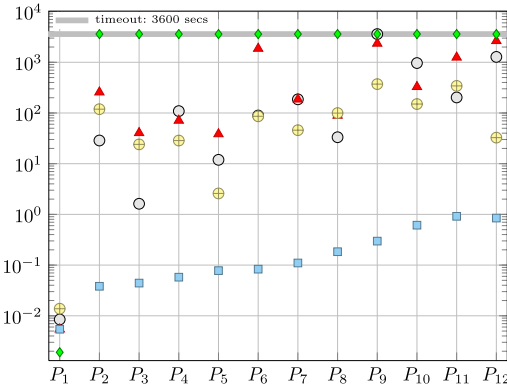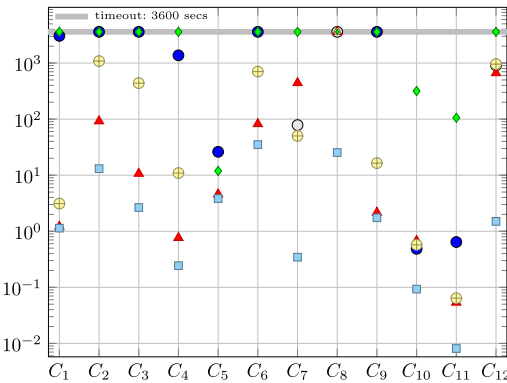## Automorphisms groups of projective planes of order 16 (regular bipartite graphs of order 546 and degree 17)

#	group size	orbits
P_1	$3.42171648e10$	1
P_2	921,600	6
P_3	884,736	3
P_4	258,048	6
P_5	147,456	3
P_6	92,160	8
P_7	55,296	8
P_8	18,432	5
P_9	12,288	6
P_{10}	3,840	10
P_{11}	3,456	12
P_{12}	2,304	14

Automorphisms of some combinatorial graphs

#	(V, E)	group size	orbits
C_1	(3650, 598600)	324	30
C_2	(15984, 10725264)	2,125,873,200	2
C_3	(7300, 2693700)	188,956,800	2
C_4	(2752, 481600)	38,723,328	2
C_5	(900000, 1200000)	600,000	2
C_6	(15984, 10725264)	231,913,440	2
C_7	(1302, 16926)	1,488,000	2
C_8	(8322, 270465)	43,352,064	8
C_9	(3650, 598600)	72	65
C_{10}	(3276, 245700)	9,000,000	3
C_{11}	(756, 49140)	9,000,000	2
C_{12}	(1514, 21196)	122,472	4

Canonical labelling of the above graphs
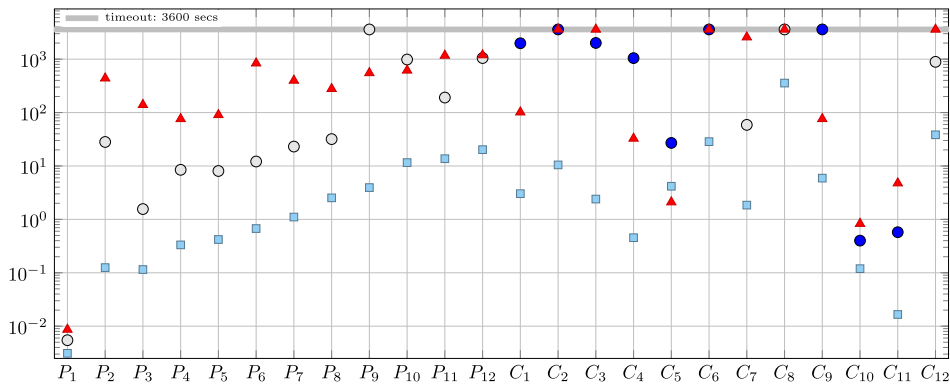
▲ Bliss   ◆ saucy   ⊕ conauto   ● nauty   ○ nauty with invariant *cellfano2*   ■ Traces

Fig. 10. Performance comparison (horizontal: graph number; vertical: time in seconds).

Most of the graphs are taken from the `Bliss` collection, but for the record we provide all of our test graphs at the `nauty` and `Traces` website (McKay and Piperno, 2012a).

The times given are for a Macbook Pro with 2.66 GHz Intel i7 processor, compiled using gcc 4.7 and running in a single thread. Easy graphs were processed multiple times to give more precise times. In order to avoid non-typical behaviour due to the input labelling, all the graphs were randomly labelled before processing. In some classes, such as the "combinatorial graphs", the processing time can depend a lot on the initial labelling; the plots show whatever happened in our tests.

The following programs were included. Programs (c)–(e) reflect their distributed versions at the end of October 2012.

(a) `nauty` version 2.5;
(b) `Traces` version 2.0;
(c) `saucy` version 3.0;
(d) `Bliss` version 7.2;
(e) `conauto` version 2.0.1.

The first column of plots in each figure is for computation of the automorphism group alone. The second column is for computation of a canonical labelling, which for all the programs here includes an automorphism group computation.

For `nauty` we used the dense or sparse version consistently within each class, depending on whether the class is inherently dense or sparse. We did not use an invariant except where indicated, even though it would often help.

`Saucy` does not have a canonical labelling option. Version 3.0, which was released just as this paper neared completion, has an amalgam of `saucy` and `Bliss` that can do canonical labelling, but we have not tested it much.

`Conauto` features automorphism group computation and the ability for testing two graphs for isomorphism. We decided that the latter is outside the scope of this study. For the same reason we did not include the program of Foggia et al. (2001) in our comparisons.

Another excellent program, that we were unfortunately unable to include for technical reasons, is due to Stoichev (2010). Many more experiments and comments can be found at http://pallini. di.uniroma1.it.

5. Conclusions

We have brought the published description of `nauty` up to date and introduced the program `Traces`. In particular, we have shown that the highly innovative tree scanning algorithm introduced by `Traces` can have a remarkable effect on the processing power. Although none of the programs tested have the best performance on all graph classes, it is clear that `Traces` is currently the leader on the majority of difficult graph classes tested, while `nauty` is still preferred for mass testing of small graphs. An exception is provided by some classes of graphs consisting of disjoint or minimally-overlapping components, here represented by non-disjoint unions of tripartite graphs. Conauto and Bliss (Junttila and Kaski, 2011) have special code for such graphs, but as yet `nauty` and `Traces` do not.

References

Aho, A.V., Hopcroft, J.E., Ullman, J.D., 1974. The Design and Analysis of Computer Algorithms. Addison–Wesley, Reading, p. 86.
Arlazarov, V.L., Zuev, I.I., Uskov, A.V., Faradzev, I.A., 1974. An algorithm for the reduction of finite non-oriented graphs to canonical form. Ž. Vyčisl. Mat. Mat. Fiz. 14, 737–743.

Babai, L., Kantor, W.M., Luks, E.M., 1983. Computational complexity and the classification of finite simple groups. In: Proceedings of the 24th Annual Symposium on the Foundations of Computer Science, pp. 162–171.

Bodlaender, H., 1990. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. J. Algorithms 11, 631–643.

Butler, G., Lam, C.W.H., 1985. A general backtrack algorithm for the isomorphism problem of combinatorial objects. J. Symb. Comput. 1, 363–381.

Colbourn, C.S., Booth, K.S., 1981. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. SIAM J. Comput. 10, 203–225.

Corneil, D.G., Gotlieb, C.C., 1970. An efficient algorithm for graph isomorphism. J. ACM 17, 51–64.

Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L., 2004. Exploiting structure in symmetry detection for CNF. In: Proceedings of the 41st Design Automation Conference, pp. 530–534.

Darga, P.T., Sakallah, K.A., Markov, I.L., 2008. Faster symmetry discovery using sparsity of symmetries. In: Proceedings of the 45th Design Automation Conference, pp. 149–154.

Filotti, I.S., Mayer, J.N., 1980. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In: Proceedings of the 12th ACM Symposium on Theory of Computing, pp. 236–243.

Foggia, P., Sansone, C., Vento, M., 2001. A performance comparison of five algorithms for graph isomorphism. In: Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, pp. 188–199.

Goldreich, O., Micali, S., Wigderson, A., 1991. Proofs that yield nothing but their validity, or all languages in NP have zero-knowledge proof systems. J. ACM 38, 690–728.

Grohe, M., 2010. Fixed-point definability and polynomial time on graphs with excluded minors. In: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, pp. 179–188.

Grohe, M., 2012. Structural and logical approaches to the graph isomorphism problem. In: Proceedings of the 23rd Annual ACM–SIAM Symposium on Discrete Algorithms, p. 188.

Junttila, T., Kaski, P., 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments and the 4th Workshop on Analytic Algorithms and Combinatorics, pp. 135–149.

Junttila, T., Kaski, P., 2011. Conflict propagation and component recursion for canonical labeling. In: Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms, pp. 151–162.

Kawarabayashi, K., Mohar, B., 2008. Graph and map isomorphism and all polyhedral embeddings in linear time. In: Proceedings of the 40th Annual ACM symposium on Theory of Computing (STOC'08), pp. 471–480.

Kirk, A., 1985. Efficiency considerations in the canonical labelling of graphs. Technical report TR-CS-85-05. Computer Science Department, Australian National University.

Kocay, W., 1996. On writing isomorphism programs. In: Wallis, W.D. (Ed.), Computational and Constructive Design Theory. Kluwer, pp. 135–175.

Leon, J.S., 1990. Permutation group algorithms based on partitions, I: Theory and algorithms. J. Symb. Comput. 43, 545–581.

López-Presa, J.L., Fernández Anta, A., 2009. Fast algorithm for graph isomorphism testing. In: Proceedings of the 8th International Symposium on Experimental Algorithms, pp. 221–232.

López-Presa, J.L., Fernández Anta, A., Núñez Chiroque, L., 2011. Conauto-2.0: Fast isomorphism testing and automorphism group computation. Preprint. Available at http://arxiv.org/abs/1108.1060.

Luks, E., 1982. Isomorphism of graphs of bounded valence can be tested in polynomial time. J. Comput. Syst. Sci. 25, 42–65.

McKay, B.D., 1978a. Backtrack programming and isomorph rejection on ordered subsets. Ars Comb. 5, 65–99.

McKay, B.D., 1978b. Computing automorphisms and canonical labellings of graphs. In: Combinatorial Mathematics. In: Lect. Notes Math., vol. 686. Springer-Verlag, Berlin, pp. 223–232.

McKay, B.D., 1980. Practical graph isomorphism. Congr. Numer. 30, 45–87.

McKay, B.D., Piperno, A., 2012a. `nautyTraces`, Software distribution web page. http://cs.anu.edu.au/~bdm/nauty/ and http://pallini.di.uniroma1.it/.

McKay, B.D., Piperno, A., 2012b. `nauty` and `Traces` User's Guide (Version 2.5). Available at McKay and Piperno (2012a).

Miller, G.L., 1980. Isomorphism testing for graphs of bounded genus. In: Proceedings of the 12th ACM Symposium on Theory of Computing, pp. 225–235.

Myrvold, W., Kocay, W., 2011. Errors in graph embedding algorithms. J. Comput. Syst. Sci. 77, 430–438.

Parris, R., Read, R.C., 1969. A coding procedure for graphs. Scientific Report. UWI/CC 10. Univ. of West Indies Computer Centre.

Piperno, A., 2008. Search space contraction in canonical labeling of graphs. Preprint 2008–2011. Available at http://arxiv.org/abs/0804.4881.

Ponomarenko, I.N., 1988. The isomorphism problem for classes of graphs that are invariant with respec to contraction. Zap. Nauchn. Sem. Leningrad. Otdel. Mat. Inst. Steklov. (LOMI) 174 (3), 147–177 (in Russian).

Read, R.C., Corneil, D.G., 1977. The graph isomorphism disease. J. Graph Theory 1, 339–363.

Seress, Á., 2003. Permutation Group Algorithms. Cambridge University Press, Cambridge. x+264 pp.

Stoichev, S.D., 2010. Polynomial time and space exact and heuristic algorithms for determining the generators, orbits and order of the graph automorphism group. Preprint. Available at http://arxiv.org/abs/1007.1726.