

Search Space Contraction in Canonical Labeling of Graphs*

Adolfo Piperno

Dipartimento di Informatica, Università di Roma La Sapienza

Via Salaria 113, I-00198 Roma

`piperno@di.uniroma1.it`

October 1, 2008

Abstract

The individualization-refinement paradigm for computing a canonical labeling and/or the automorphism group of a graph is investigated. New techniques are presented with the aim of reducing the size of the associated search space. To substantiate this motivation, a tool named *Traces*⁽¹⁾ is introduced. Experimental results and comparisons with existing tools, like McKay's "nauty", reveal that the presented approach produces a huge contraction of the search space. Such reduction will be shown to be exponential for special classes of graphs which are intractable by "nauty".

1 Introduction

When we consider the literature about methods for approaching the graph isomorphism problem (GI) or the related problem of canonical labeling of graphs, a peculiar panorama appears.

From the theoretical side, besides papers substantiating the thesis that GI is not NP-complete [21, 28] (a survey is in [1]), there is a huge amount of notable pieces of mathematics showing the existence of polynomial (time) solutions of GI, for significant classes of graphs. While moderately exponential solutions have been provided for the general problem of graph isomorphism [3, 6], polynomial algorithms exist for planar graphs [13, 14], graph of bounded genus [11], graphs with colored vertices and bounded color-classes [2], graphs with bounded multiplicity of eigenvalues [5], graphs of bounded valence [19], and more (see [4]).

From the practical side, there are some notable pieces of software, which originate from the outstanding tool *nauty* [22]. *nauty* was introduced in the 80's by McKay [23]; it has become a standard in the area of canonical labeling and detection of the automorphism group of a graph; moreover, it has been incorporated into more general mathematical software tools such as GAP [12] and MAGMA [20].

It is remarkable to observe that none of the polynomial algorithms, mentioned above, has been implemented in software, as noted by Junttila and Kaski in [16]. A reasonable justification for this absence seems to be that, usually, given a class \mathbb{C} of graphs for which an efficient algorithm for isomorphism testing exists, *nauty* is able to treat almost all the graphs of \mathbb{C} , still remaining (considerably, indeed) below the theoretically established bound. However, there might exist subsets of \mathbb{C} for which *nauty* exhibits an exponential behavior, as it results from Miyazaki's series of graphs [24].

In recent years, the tools *saucy* [10, 9] and *bliss* [16, 15] have been introduced, with the aim of handling large and sparse graphs coming either from the satisfiability problem (SAT) or from industrial applications. Like *nauty*, these are general purpose devices implementing backtrack algorithms based on the so called individualization-refinement (IR) technique and, though similar in nature, they differ from each other in the used data structures and heuristics⁽²⁾.

For any tool based on the IR technique, the running time of canonical labeling of a graph essentially comes out from the size of the associated search space, usually implemented as a tree, and from the complexity of the refinement function (which is invoked once for every node of the search tree). In addition, the

*A preliminary version of the paper is available at <http://arxiv.org/abs/0804.4881>.

¹*Traces* is available at <http://www.dsi.uniroma1.it/~piperno/pers/Traces.html>.

²We briefly recall here that a *refinement* is a classification of vertices of a graph, usually represented by a (stable) coloring; an *individualization* is the operation of imposing a singleton class to a vertex.

behavior of the refinement procedure (namely, the granularity of the classification of vertices) affects the size of the search tree.

In this paper we propose a new algorithm design, to address four primary issues of the IR method: (i) the visiting strategy of the search space, (ii) the refinement procedure, (iii) the selector of the next individualization step, (iv) the manipulation of information about the automorphism group of the graph. The theoretical bases of the new approach originate from investigating the positive aspects of some negative results in the area of canonical labeling. From a practical standpoint, our aim is to introduce a *general purpose* algorithm making the computation feasible for all the graphs which are hard for *nauty* and other similar tools. In such sense, our reference graphs are those built from Hadamard matrices, graphs coming from projective planes, other graphs of combinatorial origin and, obviously, Miyazaki's graphs, but the new algorithm is not tailored on any particular class of graphs. We will show that the presented algorithm causes a considerable reduction of the size of the associated search space: in particular, such decrease will be shown to be dramatic when very large search trees are needed by *nauty* and other tools; as an example, an exponential reduction will come out for Miyazaki's sequence of graphs.

With the mentioned motivations, we will introduce a tool named *Traces*, whose main contributions can be summarized as follows: *Traces* computes a canonical form for a colored graph and/or a set of generators for its automorphism group; in the latter case, a method is described and implemented, which can be applied to check isomorphism between two graphs G_1 and G_2 without considering $G_1 \cup G_2$ ⁽³⁾. *Traces* does not use backtrack to traverse the search space; at the implementation level, the latter is not even treated as a tree, though, for expository reasons, we will still refer to it as the *search tree*. In *Traces*, the next individualization step is chosen during the refinement procedure as the one which maximizes the size of the partition which will come *after* it, in order to define an individualization selector which does not leave vertices of the graph unconsidered as individualized ones. The refinement procedure of *Traces* produces a partition such that, for any cell W , the usual individualization and refinement of all elements of W induce the same quotient graph; moreover, partitions are compared (and possibly discarded) without computing them completely, using a linear representation which we will call *trace*. Detected automorphisms are manipulated in *Traces* by means of the Schreier-Sims algorithm [30] (see also [29]); information on the group structure is also used by the refinement procedure to eliminate redundant computations. The choice of Schreier-Sims algorithm is due to the fact that it allows, by automorphism detection, the highest range of pruning of the search tree (see [17]), which is the main concern of this paper. Leon's implementation [18] of the Schreier-Sims algorithm is integrated in *Traces*: such code is extremely efficient in the case of graphs which do not exhibit a very large automorphism group, those we are mainly interested in. We will point out the cases where the use of the Schreier-Sims algorithm heavily affects the whole computation.

We will produce results and performance tables using fragments of the huge catalogue of benchmark graphs for canonical labeling and automorphism group computation compiled by Junttila and Kaski [16]. In order to give an anticipation of our results and a final account on our motivations, we report some experiments from Tables 1 and 2, executed on an Apple Mac Pro with 2 Quad Core processors at 2.8 GHz:

pp-16-9 is a (546 vertices, 4641 edges) graph associated to one of the known projective planes of order 16 (see [27]): *bliss*, which is considerably faster than *nauty* on this graph, takes approximately one hour to canonize it, traversing more than 300 million nodes of the search tree, which has depth 5; *Traces* reduces the depth of the search tree down to 3 and traverses 187 nodes, only, in 0.39 seconds; the time spent in group computation is negligible; the size of the automorphism group of the graph is 921,600;

had-100 is a (400, 20200) graph associated to a 100×100 Hadamard matrix (see [31]): *bliss* takes 4.23 seconds to canonize had-100, traversing more than 50,000 nodes of the search tree, which has depth 5; *Traces* reduces the depth of the search tree down to 2 and traverses 725 nodes in 3,20 seconds; the time spent in group computation is negligible, too; the size of the automorphism group of the graph is 400;

the sequence of graphs mz-aug2-2 \times derives from Miyazaki's construction; *bliss* traverses more than 1 million nodes to canonize mz-aug2-18 (a (432, 684) graph), more than 4 million nodes to canonize mz-aug2-20 (a (480, 760) graph), more than 16 million nodes to canonize mz-aug2-22 (a (528, 836) graph), and so on. *Traces* traverses 145 nodes to canonize mz-aug2-18, 161 nodes to canonize mz-aug2-20, 177 nodes to canonize mz-aug2-22, a linear sequence.

These are prototypical cases. They show that the difference in performance between *Traces* and other tools for canonical labeling of graphs is located in the ability to prune the search space. When, as in the first case, the ratio between sizes of the search spaces is substantial, *Traces* runs much faster than other tools; otherwise, their computation times are comparable, modulo the time spent in the group-related computations. *Traces*

³Recall that Mathon's proof [21] of the equivalence between the graph isomorphism problem for G_1 and G_2 and the problem of computing generators of the automorphism group of a graph requires the computation of the automorphism group of $G_1 \cup G_2$.

is slower than other tools when the associated search spaces are very small, usually when the input graph has a low degree of regularity or a high degree of symmetry. Even better results will be shown in the case of automorphism group computation instead of canonical labeling.

Experiments suggest that *Traces* has a polynomial behavior on all the benchmarks families of graphs collected in [16], which include all the known hard graphs for tools based on the IR technique. The ability of *Traces* in capturing the structure of graphs will be briefly discussed in the paper, together with theoretical issues arising from the analysis of the behavior of the presented algorithm.

Traces is available at <http://www.dsi.uniroma1.it/~piperno/pers/Traces.html>.

2 Graphs and partitions

A (labeled simple) *graph* is a pair (V, E) , where V is a finite set of *vertices* and E is a set of unordered pairs of vertices called *edges*. If $(u, v) \in E$, we say that v and w are *adjacent* or *neighbors*. A (vertex) *colored graph* is a pair $G = (H, \chi)$, where H is a graph and χ is a function assigning colors to vertices of H . A more precise definition of the coloring function will be given in (1).

In this paper, $[n]$ will denote the set $\{1, \dots, n\}$, while $\mathbb{G}_{[n]}$ will denote the set of colored graphs with vertex set $[n]$. The whole set of colored graphs will be denoted by \mathbb{G} . Note that any graph can be interpreted as a colored graph in which all vertices have the same color (see (1)).

Isomorphisms of graphs in $\mathbb{G}_{[n]}$ are permutations of $[n]$ preserving adjacency as well as non adjacency. When considering graphs with colored vertices, isomorphisms must preserve colors, too. We will write $G_1 \simeq G_2$ when G_1 and G_2 are isomorphic. An *automorphism* is an isomorphism between a graph and itself. The *automorphism group* $\text{Aut}(G)$ of a graph G is the set of all automorphisms of G with composition as group operation. An (*isomorphism*) *invariant* on \mathbb{G} is a function f from \mathbb{G} to a set \mathbb{D} such that $G_1 \simeq G_2 \Rightarrow f(G_1) = f(G_2)$. A function $\mathcal{C} : \mathbb{G} \rightarrow \mathbb{G}$ is a *canonical form* iff (i) $\forall G \in \mathbb{G} : \mathcal{C}(G) \simeq G$; (ii) $\forall G_1, G_2 \in \mathbb{G} : G_1 \simeq G_2 \Leftrightarrow \mathcal{C}(G_1) = \mathcal{C}(G_2)$.

Definition 2.1 (Ordered partition). An *ordered partition* of $[n]$ is a sequence $\pi = (W_1, \dots, W_r)$ of disjoint non-empty subsets of $[n]$, called *cells*, whose union is $[n]$. The set of ordered partitions of $[n]$ will be denoted by $\Pi_{[n]}$, while Π will denote the set of all ordered partitions.

A cell of a partition $\pi \in \Pi_{[n]}$ is *trivial* when it contains only one element. The partition π is *discrete* if all its cells are trivial; π is the *unit partition* when it is constituted of only one cell, i.e. $\pi = ([n])$. For any $\pi \in \Pi_{[n]}$ and $v, w \in [n]$, we will write $v \sim_\pi w$ when v and w belong to the same cell of π .

Definition 2.2 (Index, position).

1. The *index* $\text{ind}(v, \pi)$ of a vertex $v \in [n]$ in an ordered partition $\pi \in \Pi_{[n]}$ is the index of the cell of π in which v appears, namely $\text{ind}(v, (W_1, \dots, W_r)) = k$ when $v \in W_k$.
2. The *position* of a node $v \in [n]$ in an ordered partition $\pi \in \Pi_{[n]}$ is defined by means of the function $\text{pos} : [n] \times \Pi_{[n]} \rightarrow [n]$ such that: $\text{ind}(v, (W_1, \dots, W_r)) = k \Rightarrow \text{pos}(v, (W_1, \dots, W_r)) = 1 + \sum_{i=1}^{k-1} |W_i|$.
3. The position of a cell W in an ordered partition π is defined as the position of an element of W in π (indeed, all the elements of W share the same position in π); with some overloading: $\text{pos}(W, \pi) = \text{pos}(v, \pi)$, for any $v \in W$.

Definition 2.3. Let $\preceq = \bigcup_n \preceq_n$, where the relation $\preceq_n (\subseteq \Pi_{[n]}^2)$ is defined as follows: $\pi_1 \preceq_n \pi_2 \Leftrightarrow \forall v \in [n] : \text{pos}(v, \pi_1) \geq \text{pos}(v, \pi_2)$. If $\pi_1 \preceq \pi_2$, we say that π_1 is *finer* than π_2 (and that π_2 is *coarser* than π_1).

Let $\pi \in \Pi_{[n]}$ and let (χ_1, \dots, χ_n) be a sequence of colors. The graph $G = ([n], E, \pi)$ is a colored graph if we interpret the partition π as a function assigning the $\text{pos}(v, \pi)$ -th color $\chi_{\text{pos}(v, \pi)}$ to any vertex v of G . Up to renaming of colors, the converse is also true, i.e. a coloring identifies an ordered partition of vertices. Therefore in the rest of the paper we will assume a colored graph to be a pair

$$G = ([n], E, \pi), \tag{1}$$

where π is an ordered partition of $[n]$. Note that, with the present assumption, the ordered partition π induces an ordering over colors of the graph G .

2.1 The individualization-refinement technique for canonical labeling

In this section, the behavior of the individualization-refinement technique for canonical labeling is revisited.

Definition 2.4 (Individualization). Let $\pi = (W_1, \dots, W_r) \in \Pi_{[n]}$ be a partition of $[n]$. For any $v \in [n]$, if v belongs to a non-trivial cell W_i , then we denote by $\pi \downarrow v$ the partition obtained from π by splitting the cell W_i into the cells $\{v\}$ and $W_i - \{v\}$, namely: $\pi \downarrow v = (W_1, \dots, W_{i-1}, \{v\}, W_i - \{v\}, W_{i+1}, \dots, W_r)$. If $G = ([n], E, \pi) \in \mathbb{G}_{[n]}$, then we say that $G' = ([n], E, \pi \downarrow v)$ is obtained from G by *individualizing* the vertex v .

$$\text{Note that } \text{pos}(w, \pi \downarrow v) = \begin{cases} \text{pos}(w, \pi) + 1 & \text{if } w \in W_i - \{v\}, \\ \text{pos}(w, \pi) & \text{otherwise.} \end{cases}$$

Definition 2.5 (Equitable graph). Let $G = ((V, E), \pi) \in \mathbb{G}$ be a colored graph. We say that G is *equitable* when for any two vertices $v, w \in V$ and for every cell W of π , if $v \sim_\pi w$ then v and w have the same number of neighbors in W .

Definition 2.6 (Quotient graph). Let $G = ((V, E), \pi) \in \mathbb{G}$ be an equitable graph. The *quotient graph* of G , notation $Q(G) = (V', E')$, is a graph, with possible multiple edges and loops, having vertex set $V' = \{\text{pos}(v, \pi) \mid v \in V\}$ and edge multiset $E' = \{\{\text{pos}(v, \pi), \text{pos}(w, \pi)\} \mid (v, w) \in E\}$.

A Figure showing an equitable graph and its quotient graph can be found in Appendix.1.

Definition 2.7 (Refinement). Let $G = (H, \pi) \in \mathbb{G}$. A *refinement* of G is a function $R : \mathbb{G} \rightarrow \mathbb{G}$ such that: (i) $R(H, \pi) = (H, \pi')$ is an equitable graph with $\pi' \preceq \pi$; (ii) R preserves isomorphisms, i.e. $(H_1, \pi_1) \simeq (H_2, \pi_2) \Rightarrow R(H_1, \pi_1) \simeq R(H_2, \pi_2)$.

The refinement function implemented in *nauty*, *saucy* and *bliss* is the *1-dimensional Weisfeiler-Lehman refinement* (1-dim WL, or *vertex classification*) [4]: using a specific scheduling, which is isomorphism invariant, cells of π are visited. For each visited cell W , the multiplicities of adjacencies of elements of W are counted. For any cell Z , the elements of Z are rearranged according to these values, and Z is split into the cells Z_1, \dots, Z_k such that all the elements of Z_i have the same amount of neighbors in W . This process is repeated until the graph is equitable.

Definition 2.8 (Target cell). Let $G = (H, \pi) \in \mathbb{G}$. A *target cell (selector)* of G is a function $T : \mathbb{G} \rightarrow [n]$ such that: (i) $T(H, \pi) = k$ is the position of a non-trivial cell of π ; (ii) T is an isomorphism invariant, i.e. $(H_1, \pi_1) \simeq (H_2, \pi_2) \Rightarrow T(H_1, \pi_1) = T(H_2, \pi_2)$.

The target cell selector used in *nauty*, *saucy* and *bliss* has a static nature: cells are classified according to the presence of edges and non-edges wrt other cells. The leftmost cell having the highest value in such classification is chosen as the target cell.

Notation 2.9. Given an equitable graph $G(H, \pi)$ and a vertex v of H , we will write $(H, \pi^{(v)})$ as a shorthand for the refinement of the graph obtained by individualizing v : $(H, \pi^{(v)}) = R(H, \pi \downarrow v)$.

The *search tree* of an equitable graph $G = (H, \pi)$ is a labeled tree $\mathcal{T}(H, \pi)$ such that: (i) the root of $\mathcal{T}(H, \pi)$ has label π ; (ii) if π is discrete, then the tree rooted at it consists of that single node; otherwise, let $\pi = (W_1, \dots, W_i, \dots, W_k)$ and let $W_i = \{v_1, \dots, v_h\}$ be the target cell of (H, π) . Then the tree rooted at π has as children the trees $\mathcal{T}(H, \pi^{(v_1)}), \dots, \mathcal{T}(H, \pi^{(v_h)})$ and, for $i = 1, \dots, h$, the edge joining π with $\mathcal{T}(H, \pi^{(v_i)})$ has label v_i .

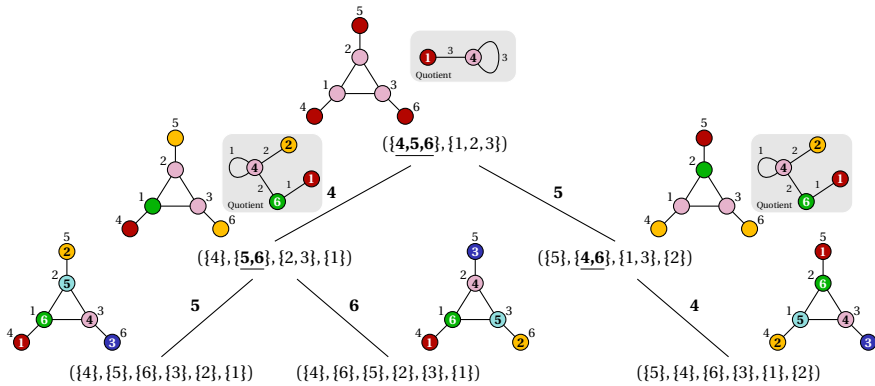


Figure 1: The pruned search tree; at the leaves, graphs and their quotients coincide

The typical behavior of algorithms based on the individualization-refinement mechanism is exemplified by the backtrack search in Figure 1, which is borrowed from *nauty*'s user guide [22]: given an equitable colored graph $G = (H, \pi)$, the root of the tree is labeled by the partition π . The target cell is here the underlined one,

⁴By abuse of notation, while considering, for some \mathbb{W} , a function $F : \mathbb{G} \rightarrow \mathbb{W}$, we will write $F(H, \pi)$ instead of $F((H, \pi))$ in order to avoid duplicated parentheses.

namely $\{4, 5, 6\}$. Any node of the tree is labeled by an equitable partition which is obtained by its father by individualizing each vertex in the target cell and then refining the obtained partition. The individualized vertices appear as edge labels in the tree. When the leftmost leaf is reached, the corresponding quotient graph is stored as a potential canonical form C_G for G . When the next leaf is reached, its corresponding quotient graph C'_G is compared with C_G . If they are equal, an automorphism of G has been found. Otherwise, C'_G is either discarded or it substitutes C_G if it is “better” according to some predefined ordering. Only part of the whole backtrack tree is actually generated. The other parts of the tree are either shown to be equivalent to parts already generated, or are pruned by means of invariant information discovered while traversing the tree itself. As an example, in Figure 1, the vertex 6, which belongs to the target cell of the root, is not individualized, since the first two visited leaves identify the permutation $(2, 3)(5, 6)$ as an automorphism of G . This implies that the tree obtained by individualizing the vertex 6 is isomorphic to the one individualizing the vertex 5: therefore it does not carry any additional information.

3 Introducing *Traces*

In this Section, we will introduce an IR tool named *Traces*, in which the refinement procedure, the target cell selector and the visiting strategy of the search space have a completely different structure from that of its predecessors. *Traces* is introduced with the aim of substantiating, from an experimental standpoint, an investigation on the IR technique, which has been inspired by two negative results: (i) Miyazaki’s sequence of graphs ([24]) showing the exponential behavior of *nauty*; (ii) Cai, Fürer and Immerman’s construction ([7]) about (non) identification of graphs via the k -dim WL refinement (vertex classification by means of k -tuples of vertices). Miyazaki proved that a wrong choice of the target cell is responsible for the existence of intractable graphs for *nauty*. We will therefore define and experiment some new criteria for selecting the target cell. On the other hand, in [7] the authors show that there does not exist a refinement procedure (in the generalized Weisfeiler-Lehman style) which is powerful enough to capture the orbit partition of any graph.

3.1 Traversing the search tree

As a matter of fact, while Miyazaki’s result has a negative impact on *nauty* and other practical isomorphism tools, Cai, Fürer and Immerman’s construction provides a theoretical justification for the IR technique, since it proves that it is impossible to solve the isomorphism problem by means of Weisfeiler-Lehman refinement, without having to associate to it a (backtrack) search space.

With such assumption, we first observe that a backtrack search (i.e. a depth-first visit of the search space) may cause inefficiencies when, at some level, it is possible to prune the tree by a node invariant. This happens when two nodes appearing at the same level in $\mathcal{T}(H, \pi)$ are labeled by partitions which induce different quotient graphs. In this case, a depth-first strategy might force visiting a whole subtree which will be later discarded. A breadth-first strategy, instead, allows for pruning the mentioned subtree before traversing it.

On the other hand, we must observe that a simple breadth-first strategy does not allow for an early pruning of the search tree by means of automorphism detection: in effect, automorphisms are discovered comparing the leaves of the tree itself. Therefore, *Traces* will experiment the use of a (variant of a) breadth-first strategy for traversing the search space: for any level ℓ of $\mathcal{T}(H, \pi)$ and for any node v appearing at ℓ , either v is discarded or one and only one path toward a leaf of $\mathcal{T}(H, \pi)$ is computed (see Figure 2). In particular: (i) nodes at level ℓ will share the same quotient graph; (ii) the computation of the path toward a leaf will be started only if v is not equivalent (by automorphism) to some previously computed node at level ℓ .

This strategy makes *Traces* conceptually different from its predecessors. In addition, it has to be noted that, while *nauty* allows the user for defining his own refinement procedure and target cell selector, it includes them into a classical backtrack algorithm. Therefore, *Traces* cannot be directly implemented in *nauty*.

Traces’ strategy for traversing the search space can be viewed as an iterated visit of a forest of AND/OR trees. In such trees, nodes are discriminated between AND nodes and OR nodes (the black nodes and the white ones in Figure 2, respectively). AND nodes are those which require all their children to be visited, while only one child is visited when starting from an OR node. The tree is visited by level iterations; at the i -th iteration, the forest of trees rooted at level i is considered, together with the assumption that their roots are AND nodes, while all other nodes are OR nodes.

The adopted strategy has the following property with respect to automorphism detection.

Definition 3.1. Let us call *congruous* any node μ_0 of the search tree of a colored graph $(H, \bar{\pi})$ such that any descendant μ of μ_0 (including μ_0 itself) has a label π which represents the orbit partition of (H, π) .

Property 3.2. Let μ be a congruous node, labeled by π , of the search tree of a colored graph. Every leaf of the tree rooted at μ identifies the same quotient graph. Therefore, any pair of paths from μ to a leaf determines one

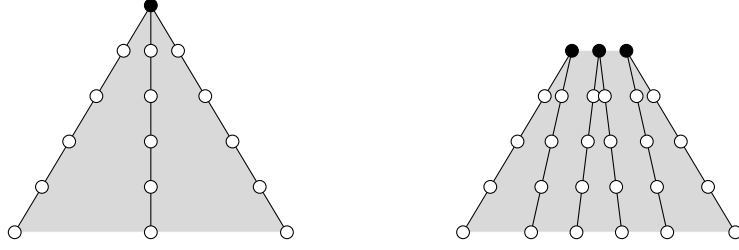


Figure 2: Visiting the search space (first and second iteration)

generator of the automorphism group of the colored graph (H, π) .

Proof. By induction on the depth of the tree rooted at μ . □

It follows that *Traces'* strategy allows for the maximal pruning of a subtree rooted at a congruous node, since it leaves only one descendant to be considered at the next level iteration.

3.2 The refinement procedure and the target cell selector

The refinement procedure of *Traces* is based on the following notion, which is a necessary condition to produce congruous nodes of the search tree:

Definition 3.3. Given $G = (H, \pi) \in \mathbb{G}$, let \bar{R} be a refinement function. The partition π respects individualization wrt \bar{R} when, for any pair of vertices v and w of G , $v \sim_{\pi} w \Rightarrow Q(\bar{R}(H, \pi \downarrow v)) = Q(\bar{R}(H, \pi \downarrow w))$, i.e., if v and w belong to the same cell of π , then the \bar{R} -refinements of their individualizations induce the same quotient graph.

Definition 3.4 (Multi-refinement). Let \bar{R} be a refinement function. A *multi-refinement* of \bar{R} is any refinement $R : \mathbb{G} \rightarrow \mathbb{G}$ such that $R(H, \pi) = (H, \pi')$, where π' respects individualization wrt \bar{R} .

Given a refinement function \bar{R} , a multi-refinement of \bar{R} can be easily defined as follows:

Algorithm 3.5. Uses the refinement function \bar{R} ; input: $G = (H, \pi) \in \mathbb{G}$;

1. let $(H, \bar{\pi}) = \bar{R}(H, \pi)$;
2. let W be the leftmost non-trivial cell of $\bar{\pi}$;
3. for any $v \in W$, compute $Q(\bar{R}(H, \bar{\pi} \downarrow v))$, the quotient graph of the \bar{R} -refinement of the individualization of v in $\bar{\pi}$;
4. order the so obtained quotient graphs (e.g., lexicographically) and split the cell W according to such ordering, thus obtaining a partition $\bar{\pi}'$;
5. if W has been split, then repeat steps 1-5 with $\bar{\pi}'$ in place of $\bar{\pi}$, otherwise consider the next non-trivial cell (if any) and restart from step 3.

Assuming \bar{R} to be the 1-dim WL refinement, *Traces* uses a modification of the multi-refinement of \bar{R} defined by Algorithm 3.5: in particular, if the individualizations-refinements of the elements of a subset W' of the cell W (step 3 of Algorithm 3.5) give discrete partitions, then the value of the multi-refinement procedure is directly produced by taking the discrete partition which gives the lexicographically smallest quotient graph. Note that any discrete partition vacuously respects individualizations and that such choice is isomorphism invariant. In addition, when two discrete partitions (coming from W') induce the same quotient graph, then an automorphism of the input graph G has been found; therefore, such an automorphism is added (during the execution of the refinement function) to the set of generators of the automorphism group of G .

Multi-refinement is computationally much heavier than usual refinement. Algorithm 3.5 can be viewed, when \bar{R} is the 1-dim WL refinement, as representing the 2-dim WL procedure (or *edge classification* [4]). Therefore, in the worst case we could be losing a factor of n (the number of vertices of the considered graph) in time, when the refinements of all vertex individualizations are needed to produce the multi-refinement. However, from one hand some techniques will be introduced (in Section 3.3) in order to manage the required refinements without computing them completely, including a new representation of the whole refinement process. From the other hand, multi-refinement naturally suggests its parallel execution, though this is outside the scope of the present paper.

In addition, information coming from the automorphism group of the graph is used to avoid the individualization of vertices which are known to belong to the same orbit of an already individualized one.

The behavior of *Traces'* refinement over Miyazaki's graphs is described in Appendix.2.

The choice of the target cell as a further motivation for multi-refinements. A serious motivation for introducing multi-refinements is that they define a family of target cell selectors, each of which is based on information coming from *the next* individualization step.

Let (H, π) be a colored graph. During the construction of its multi-refinement (H, π') , we keep information about the sizes of partitions deriving from refinements of vertex individualizations, and then we choose as target cell the leftmost one in π' which will subsequently produce the partition with the maximal size. This choice is isomorphism invariant. It comes out that multi-refinement allows the cell selector to behave in a *fair* way: a cell is never left unconsidered for being the one from which the next individualizations will come out. This appears as a major property if we want to avoid inefficiencies coming from the wrong choice of the target cell.

3.3 Comparing refinements and multi-refinements

In *Traces*, refinements are compared without computing them completely. This possibility has been already observed by Junttila and Kaski in [16] in the case of singleton cells emerging during the refinement process (the consequent invariant is called *partial leaf certificate*, and is adopted in *Traces*, too). In addition to this, we introduce and implement a different invariant which we will name *refinement trace*. Assume that the cell W of the partition π is split during the refinement process into W' and W'' . This splitting gives the partition π' such that: $pos(W', \pi') = pos(W, \pi)$ and $k = pos(W'', \pi') = pos(W, \pi) + |W'|$. The new position k created by splitting the cell W is a *trace element* of the refinement process. The refinement trace is the sequence of trace elements successively introduced during refinement; it is isomorphism invariant and can be stored into an array. Moreover, let us consider the alternation of individualization and refinement steps which is needed to compute a discrete partition; the whole process has its own trace, since the individualization operation consists of a cell splitting, too. Such trace has length at most n , and each of its elements appears exactly once in it. Note that, being sequences of integers, traces can be ordered, e.g. component-wise.

Assume now that the refinement of $G_1 = (H_1, \pi_1)$ has been computed and $\tau = (\tau_1, \dots, \tau_m)$ is its trace. While computing the refinement of $G_2 = (H_2, \pi_2)$ we stop as soon as we find a trace element τ'_i different from the corresponding τ_i . This is sufficient to realize that the refinements of G_1 and G_2 are different. Obviously, if we choose an ordering on traces and τ'_i is "better" than τ_i , than the refinement of G_2 will be completed and its trace will be stored for later comparisons.

Multi-refinements are classified by means of their traces, too. In this case, the sequence which contains the new positions created by the splitting cell is initially recorded, and then expanded with the sequence originating from calling the refinement procedure (first statement of Algorithm 3.5) on the updated partition.

3.4 Automorphisms

As previously mentioned, automorphisms are manipulated in *Traces* by means of Leon's implementation [18] of the Schreier-Sims algorithm [30, 29]. More precisely, when a new automorphism is detected, it is added as a new generator to the group computed so far. In addition, information coming from the automorphism group of the graph is used to avoid the individualization of vertices which are known to belong to the same orbit of an already individualized one; this occurs during the multirefinement process and during the individualization of elements of the target cell. In both cases, the orbits of the stabilizer of already individualized vertices are computed and the computation proceeds considering only one representant per orbit.

4 Experimental results

In the present section, the current implementation of *Traces* is compared with *nauty* and *bliss*. A comparison with *saucy* can be obtained from our performance tables and those presented in [16].

Graphs are selected from the library of benchmarks which is attached to the *bliss* distribution ([15]). In order to obtain information on sizes of search trees, it has been decided not to time out, whenever reasonably possible, any experiment. As a consequence, only a few graphs have been considered, and, for each of them, only one instance. However, this seems to be sufficient to obtain a clear comparison between the different approaches. Experiments have been carried out on a Apple Mac Pro with 2 Quad Core processors at 2.8 GHz, under gcc 4.0. For each experiment, the following information is reported in Tables 1 and 2: (1) the name of the graph, (2) the number of its vertices and edges, (3) the size of the automorphism group of the graph, (4) the number of its orbits; for *nauty* and *bliss*, the execution time (in seconds) and the size and maximal level of

the associated search tree (column Refine (ℓ), considered as the amount of calls of the refinement function). For *Traces*, the following information is reported: (i) the execution time (in seconds), (ii) the amount of calls to the multi-refinement function (which gives the measure of the search space) together with the amount of individualizations needed to compute a discrete partition (which indicates the depth of the search space), (iii) the number of computed generators of the automorphism group, (iv) the time spent in managing the group. The algorithm used by *Traces* in order to compute the automorphism group (without searching for the canonical form) of the input graph is a slight variant of the one presented above: such an algorithm returns a sequence of permutations sufficient for isomorphism testing of two graphs without considering their union; the final column of Table 1 indicates the amount of such permutations, whereas the full description of the algorithm can be found in [26].

With reference to the classification in the mentioned graph library, we have selected graphs from the following families:

- affine and projective geometries: graphs ag2-x, pg2-32;
- Cai-Fürer-Immerman construction: graphs cfi-x;
- constraint satisfaction problems: graphs difp-20-0, fpga-x-y, s3-3-3-3, urq8-5;
- Hadamard matrices: graphs had-x, had-sw-x (with some switching operations);
- Miyazaki constructions: graphs mz-x, mz-aug-x, mz-aug2-x;
- projective planes: graphs pp16-x; taken from [25]: flag-6;
- random regular graphs: graphs rnd-3-reg-x-y;
- strongly regular graphs: graphs latin-x, latin-sw-x-y, lattice-30, sts-x, sts-sw-x-y;
- complete graphs: graphs k-x;
- grid graphs: graphs grid-x-y, grid-w-x-y.

Remark 4.1. A relevant feature of *nauty* is that it gives the user the opportunity of using some sort of invariant to assist the built-in refinement procedure. Some of these invariants are very useful for some families of difficult graphs. However, their selection is left by *nauty* to the user, as motivated in the following paragraph, quoted from *nauty 2.4* user manual [22]: *Even amongst the vertex invariants which are known to be useful, their usefulness varies so much with the type of graph they are applied to, or the levels of the search tree at which they are applied, that intelligent automatic selection of a vertex-invariant by nauty would seem to be a task beyond our current capabilities.* Following this advice, in our main comparison tables we will only consider the default refinement for *nauty*, since the use of a vertex invariant would require the identification of the input graph, thus contradicting the assumption of a general purpose algorithm. From a methodological standpoint, this allows an homogeneous comparison of the considered tools. Of course, it would be possible to implement *nauty's* invariants in *Traces*, too, but this is beyond the scope of the paper. We conclude this remark observing that, in the case of very hard graphs, *Traces* outperforms *nauty* even when the latter is run with the appropriate invariant, as shown by the next table:

	<i>nauty 2.2 - aut</i>			<i>Traces - aut</i>
	Time (secs)	With invariant	Time (no invariant)	Time (no invariant)
had-232	142.76	cellquads @ level 2	29352.07	14.99
pp16-2	130.50	cellfano2 @ levels 1,2	42288.08	2.51
pp16-8	251.09	cellfano2 @ levels 1,2	683.93	0.22
flag-6	163028.21	cellfano2 @ levels 1,2	Timed out	40.45

Remark 4.2. At present, *Traces* is implemented as an additional command of *nauty*; in such sense, it uses the data structures of *nauty 2.2*, but it contains new procedures for the refinement function and the target cell selector, together with a new representation of the search space and new procedures for visiting it. A new implementation, totally independent, is under development.

We provide two tables of results: in Table 1, graphs having large search trees in *nauty* and *bliss* are considered, while in Table 2 (which, due to the lack of space, is placed in Appendix) experiments with graphs having small search trees are reported.

Table 1: graphs with large search space. For all the considered graphs, *Traces* exhibits a drastic decrease for the size of their search space, with clear consequences over computation time. It comes out that the hardest instances are graphs with small automorphism group, such as had-236. The gain in performance of *Traces* wrt *nauty* and *bliss* is considerable for all graphs in the pp-16 family, with the addition of flag-6 (taken from Moorhouse's library [25]), which comes from a projective plane of order 27.

The reader can verify the exponential contraction of the search space of *Traces* wrt the ones of *nauty* and *bliss* in the case of Miyazaki's sequence mz-aug2, also observing that timings include group computation.

Concerning the depth of the search space, its contraction is evident in all cases.

Graph	(V,E)	Aut	Orb	<i>naivy (2.2) - aut</i>		<i>bliss (0.35) - can</i>		<i>bliss (0.35) - aut</i>		<i>Traces - can</i>			<i>Traces - aut</i>				
				Time	Refine (L)	Time	Refine (L)	Time	Refine	Time	MRef (ind)	Gns	Grp	Time	MRef	Grp	Rsd
had-52	(208,5512)	2 ⁴ ·13	2	10.73	190034 (5)	0.50	14536 (4)	0.50	15533	0.26	287 (2)	5	0.02	0.15	128	0.01	55
had-100	(400,20200)	2 ⁴ ·5 ²	2	185.00	1348634 (5)	4.23	52819 (4)	4.43	58613	3.20	725 (2)	5	0.04	1.83	321	0.03	103
had-184	(736,68080)	2 ⁶ ·23	2	8544.87	4287786 (6)	33.34	111862 (5)	34.96	125285	10.13	778 (2)	7	0.12	6.11	356	0.08	143
had-232	(928,108112)	2 ⁶ ·29	2	29352.07	8592888 (6)	80.48	183720 (5)	84.51	199679	26.41	968 (2)	7	0.21	14.99	438	0.04	179
had-236	(944,111864)	2	472	Timed out	--	23322.51	104043906 (3)	23150.37	104043906	7599.86	222787 (2)	1	16.99	103.36	1888	0.16	941
had-sw-32-1	(128,2112)	2 ²	42	36.53	1149410 (5)	6.06	192369 (4)	4.64	141832	0.59	2251 (2)	2	0.05	0.08	168	0.02	81
had-sw-88	(352,15664)	2 ²	132	17633.55	80502284 (6)	309.90	3173883 (3)	322.61	3645512	40.90	19452 (2)	2	0.71	2.53	663	0.01	305
had-sw-112	(448,25312)	2	132	Timed out	--	939.28	10990338 (3)	925.85	10990338	240.79	50179 (2)	1	1.42	7.12	896	0.01	445
mz-aug2-18	(432,684)	2 ⁸	252	106.08	5374331 (38)	26.30	1048954 (37)	22.64	1048954	1.21	145 (19)	38	0.41	1.22	145	0.50	2
mz-aug2-20	(480,760)	2 ⁴ ·2	280	525.01	23593421 (42)	116.30	4194764 (41)	110.14	4194764	1.51	161 (21)	42	0.62	1.50	161	0.61	2
mz-aug2-22	(528,836)	2 ⁶	308	2500.61	102760999 (46)	515.66	16777766 (45)	439.06	16777766	2.10	177 (23)	46	0.84	2.06	177	0.80	2
mz-aug2-30	(720,1140)	2 ⁶ ·2	420	--	--	--	--	--	--	5.48	241 (31)	62	2.24	5.47	241	2.25	2
mz-aug2-50	(1200,1900)	2 ¹⁰ ·2	420	--	--	--	--	--	--	25.53	401 (51)	102	9.39	25.44	401	9.32	2
pp-16-1	(546,4641)	> 3.42e10	1	0.06	144 (6)	0.02	133 (5)	0.02	144	0.09	16 (4)	6	0.02	0.09	16	0.03	1
pp-16-2	(546,4641)	2 ⁸ ·3·5	10	42288.08	80597650 (5)	23886.33	1994354150 (5)	1868.28	80597650	50.74	11907 (3)	5	1.54	2.51	646	0.09	223
pp-16-4	(546,4641)	2 ¹² ·3	6	5267.15	10311534 (5)	3624.68	301374561 (5)	282.26	10311534	15.78	3877 (3)	3	0.55	2.77	622	0.07	311
pp-16-6	(546,4641)	2 ¹¹ ·3 ²	5	Timed out	--	17769.12	901487681 (5)	18438.75	968486421	19.94	4864 (3)	5	0.97	0.96	304	0.06	84
pp-16-7	(546,4641)	2 ¹⁴ ·3 ²	3	Timed out	--	370.87	16177794 (5)	4662.65	210159039	1.63	402 (3)	8	0.19	0.38	106	0.04	24
pp-16-8	(546,4641)	2 ¹⁵ ·3 ³	3	683.93	1706149 (5)	197.71	16589102 (5)	23.58	1170686	0.42	139 (3)	8	0.10	0.22	70	0.05	14
pp-16-9	(546,4641)	2 ¹² ·3 ² ·5 ²	6	15598.57	35144658 (5)	3575.12	303816170 (5)	825.15	35107649	0.39	187 (3)	4	0.07	0.24	118	0.04	32
pp-16-11	(546,4641)	2 ¹² ·3 ² ·7	6	Timed out	--	916.16	45141041 (5)	4276.61	219395299	1.07	379 (3)	6	0.15	0.35	132	0.02	32
pp-16-15	(546,4641)	2 ¹¹ ·3 ³	8	Timed out	--	1929.39	145623212 (5)	3515.83	278419116	3.88	1092 (3)	6	0.18	0.46	193	0.05	70
pp-16-17	(546,4641)	2 ¹¹ ·3 ² ·5	8	7032.42	14074851 (5)	2576.56	216934489 (5)	343.41	14074851	2.37	691 (3)	7	0.18	0.46	193	0.05	55
pp-16-19	(546,4641)	2 ⁸ ·3 ²	14	Timed out	--	8288.57	666219267 (5)	3792.12	130609514	78.80	18877 (3)	5	2.93	3.50	858	0.15	355
pp-16-21	(546,4641)	2 ⁷ ·3 ³	12	Timed out	--	10249.02	617020245 (5)	25871.22	1622520018	53.41	12895 (3)	4	1.71	2.79	681	0.14	300
flag-6	(1514,21196)	2 ³ ·3 ⁷ ·7	4	Timed out	--	16396.88	749371122 (5)	26665.97	706556032	584.45	12304 (3)	7	5.85	40.45	1385	0.68	67
sts-67	(737,35376)	3	253	8.91	23841 (3)	8.04	157566 (2)	7.77	157566	1.67	255 (1)	1	0.02	2.12	255	0.01	253
sts-sw-21-10	(70,945)	1	70	0.03	1961 (3)	0.03	3011 (2)	0.03	3011	0.00	71 (0)	0	-	0.00	71	-	70
sts-sw-55-1	(495,19305)	1	495	7.76	39106 (3)	6.72	206416 (2)	6.57	206416	1.75	496 (1)	0	-	2.12	496	-	495
sts-sw-79-11	(1027,58539)	1	1027	73.09	118106 (3)	50.74	937652 (2)	49.80	937652	17.01	1028 (1)	0	-	19.03	1028	-	1027

Table 1: Graphs with large search tree

In addition, *Traces* seems to have a more stable behavior on different instances of graphs in the same family (the required multi-refinements and computation time being related to the sizes of the input graph and of its automorphism group), and also on different representations of the same graph, as shown by the following Table, where the canonical forms of graphs coming from random permutations of vertices of pp-16-8, a projective plane of order 16, are computed.

	<i>bliss (0.35) - can</i>		<i>Traces - can</i>	
	Canon. time	Refinements (ℓ)	Canon. time	Multi-refs (ind)
pp-16-8	197.71	16589102 (5)	0.42	139 (3)
perm ₁	3663.95	311919402 (5)	0.44	162 (3)
perm ₂	1765.20	150066473 (5)	0.40	137 (3)
perm ₃	8183.51	704831435 (5)	0.50	167 (3)

It comes out that *Traces* has a stronger ability of capturing the structure of the graph, thus abstracting from its representation. A further evidence of such claim can be also deduced from the series cfi- x , where the depth of *Traces*' search space is equal to the main parameter introduced by Cai, Fürer and Immerman in their construction [7]. Finally, all experiments (those reported in Table 2, too) exhibit congruous nodes (recall Def. 3.1) starting from at most level 3 in the associated search tree.

Table 2: graphs with small search space. Graphs exhibiting a small search space (wrt the size of their vertex set) come out to have either a large automorphism group or a trivial one. In the first case, the search space is massively pruned by automorphisms, in the second case just a few individualization steps are needed to obtain discrete partitions. These are the most favorable situations for the individualization-refinement technique. Still *Traces* is able to reduce both the depth and size of search space.

However, it is clear that the overhead determined by multiple refinements causes a (sometimes considerable) loss of performance. The most critical experiments for *Traces* are very sparse graphs (s3-3-3-3 and grid graphs): we observe that, in such graphs, partitions induced by simple refinement are equal to those deriving from multi-refinement, and this happens for every individualization step. Therefore multi-refinements are not able to get any additional information. As expected, in these cases, the ratio between the execution time of *Traces* and *nauty* (or *bliss*) can reach the number of vertices of the input graph. Comparing these results, however, we must notice that *Traces*, beyond representing graphs by means of adjacency lists, does not implement any other technique for handling large and sparse graphs, yet.

Table 2 shows that the time spent in group computation becomes significant when dealing with very large automorphism groups: this is evident in the case of complete graphs.

5 Concluding remarks and further work

Starting from an investigation of the individualization-refinement technique for canonical labeling of graphs, a new algorithm has been presented with the aim of reducing the associated search space. The main novelties of the proposed method can be found (i) in the strategy for visiting the search space, (ii) in the choice of the refinement procedure and the selector of the next individualization steps, (iii) in the possibility of comparing refinements without computing them completely, (iv) in the use of Schreier-Sims algorithm for handling information from the automorphism group of the input graph.

Experimental results have been provided showing a significant improvement of performances in the case of graphs of combinatorial origin which are considered as critical for any algorithm adopting the IR method. The comparison between tables of results presented in the paper leaves an interesting scenario and several directions for further work. In effect, the new approach and the state of the art ones are in some sense complementary, which hints to their possible integration. As an example, *Traces* and *nauty* (or *bliss*) could exchange information about detected automorphism of the input graph in order to obtain the best performances from one another. At the experimental level, the main achievement of this paper is that *Traces* does not have an exponential behavior on any of the benchmark families of graphs considered in the literature for the IR method. At the implementation level, *Traces* is still at an initial stage; a new implementation is currently under development, with a special attention to the multi-refinement procedure and to its possible parallel implementation. From a theoretical standpoint, the positive aspects of some negative results in the area of canonical labeling have been enlightened. A further investigation on the theoretical aspects of the IR technique seems to be in order. A better understanding of the consequences of Cai, Fürer and Immerman's construction has been achieved: experiments suggest that, for the search space of such graphs, *Traces* immediately produces congruous nodes. Among other directions for further work, we ask whether this might put Cameron-Gel'fand theorem ([8]), which roughly proves that an excess of regularity implies symmetry, back into the game of canonical labeling and graph isomorphism.

Acknowledgments

I would like to thank my friends and colleagues Maurizio Bonuccelli, Angelo Monti and Riccardo Silvestri for their suggestions and discussions about the topics of this paper.

References

- [1] Vikraman Arvind and Jacobo Torán. Isomorphism testing: Perspective and open problems. *Bulletin of the EATCS*, 86:66–84, 2005.
- [2] László Babai. Monte Carlo algorithms in graph isomorphism testing. Technical Report 79-10, Dép. Math. et Stat., Univ. de Montréal, 1979.
- [3] László Babai. Moderately exponential bound for graph isomorphism. In *FCT*, pages 34–50, 1981.
- [4] László Babai. Automorphism groups, isomorphism, reconstruction. In *Handbook of combinatorics (vol. 2)*, pages 1447–1540. MIT Press, Cambridge, MA, USA, 1995.
- [5] László Babai, D. Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *STOC*, pages 310–324, 1982.
- [6] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC*, pages 171–183, 1983.
- [7] Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992.
- [8] Peter J. Cameron. 6-transitive graphs. *J. Comb. Theory, Ser. B*, 28(2):168–179, 1980.
- [9] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. *saucy 1.1*, available at <http://vlsicad.eecs.umich.edu/BK/SAUCY/>.
- [10] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *DAC*, pages 530–534, 2004.
- [11] I. S. Filotti and Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus (working paper). In *STOC*, pages 236–243, 1980.
- [12] GAP. System for computational discrete algebra: <http://www-gap.dcs.st-and.ac.uk>.
- [13] John E. Hopcroft and Robert Endre Tarjan. A $v \log v$ algorithm for isomorphism of triconnected planar graphs. *J. Comput. Syst. Sci.*, 7(3):323–331, 1973.
- [14] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC*, pages 172–184, 1974.
- [15] Tommi A. Junttila and Petteri Kaski. *bliss 0.35*, available at <http://www.tcs.hut.fi/Software/benchmarks/ALENEX-2007/>.
- [16] Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX*, 2007.
- [17] William Kocay. On writing isomorphism programs. In *Computational and Constructive Design Theory*, pages 135–175, 1996.
- [18] Jeffrey S. Leon. *Partition backtrack programs*, available at <ftp.math.uic.edu/pub/leon/partn>.
- [19] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.
- [20] MAGMA. Computational algebra system: <http://magma.maths.usyd.edu.au/magma>.
- [21] Rudolf Mathon. A note on the graph isomorphism counting problem. *Inf. Process. Lett.*, 8(3):131–132, 1979.
- [22] Brendan D. McKay. *nauty 2.2/2.4*, available at <http://cs.anu.edu.au/~bdm/nauty>.

- [23] Brendan D. McKay. Practical graph isomorphism. *Congr. Numer*, 30:45–87, 1981.
- [24] Takunari Miyazaki. The complexity of mckay's canonical labeling algorithm. In *Groups and Computation II, DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 239–256, 1997.
- [25] G. Eric Moorhouse. *Projective Planes*, available at <http://www.uwo.edu/moorhouse/pub/>.
- [26] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, <http://arxiv.org/abs/0804.4881>, 2008.
- [27] Gordon Royle. *Projective Planes Of Order 16*, available at <http://people.csse.uwa.edu.au/gordon/data.html>.
- [28] Uwe Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.
- [29] Ákos Seress. *Permutation group algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003.
- [30] Charles C. Sims. Computation with permutation groups. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 23–28. ACM, 1971.
- [31] Neil J.A. Sloane. *A Library of Hadamard Matrices*, available at <http://www.research.att.com/~njas/>.

Appendix

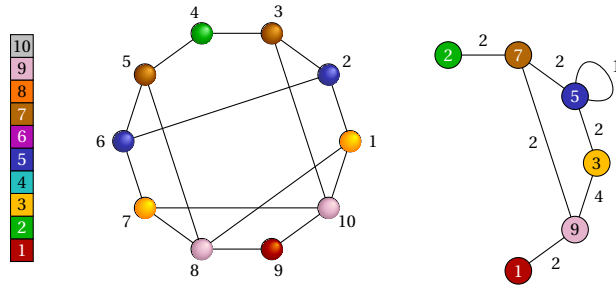


Figure 3: An equitable graph and its quotient. Here $\pi = (\{9\}, \{4\}, \{1, 7\}, \{2, 6\}, \{3, 5\}, \{8, 10\})$.

1. Figure 3 shows an equitable graph G and its quotient graph. For every color appearing in G , there exists a corresponding vertex in $Q(G)$; an edge in $Q(G)$ has multiplicity k if G has k edges joining vertices with the corresponding colors.

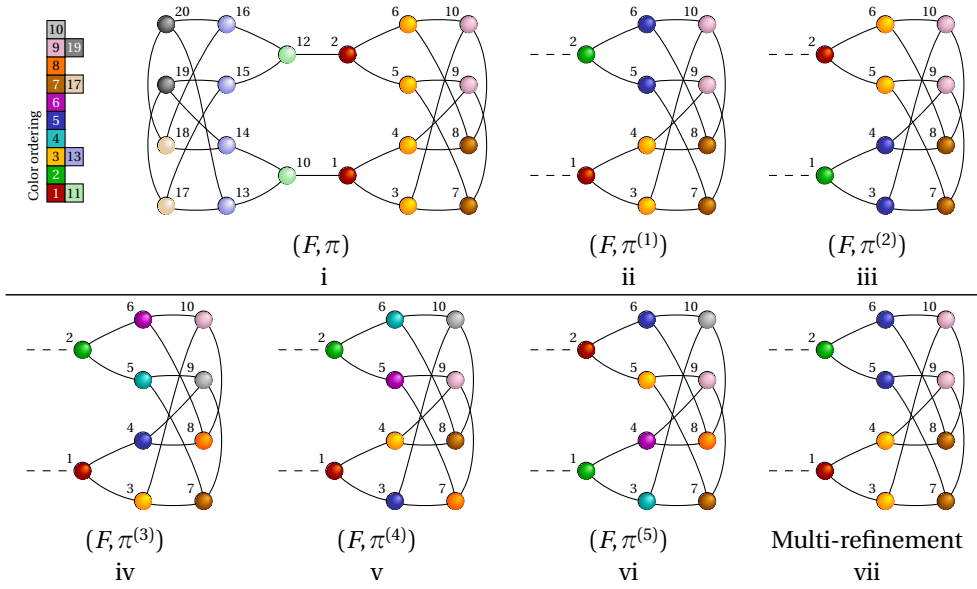


Figure 4: Multi-refining the Furer gadget in the first Miyazaki's graph

2. In order to give a flavor of the behavior of the multi-refinement procedure, and to realize that Miyazaki's construction is not critical for *Traces*, we show in Figure 4 the simplest graph in Miyazaki's series. The graph and its coloring appear in Figure 4.i. We focus on the right part of the graph, only. For this graph F , first introduced by Furer (see [7]), we show the refinements obtained after individualization of the vertices 1 to 5, in Figure 4.ii to vi, respectively. It comes out that $(F, \pi^{(1)})$ and $(F, \pi^{(2)})$ have the same quotient graph, as well as $(F, \pi^{(3)})$ and $(F, \pi^{(4)})$. $Q(F, \pi^{(4)})$, the quotient graph of $(F, \pi^{(4)})$, however, is different from $Q(F, \pi^{(5)})$, since the vertex 9 has the same color in both graphs, but different colors for neighbors. Therefore, the cell $\{3, 4, 5, 6\}$ is split, and this also causes the splitting of the cell $\{1, 2\}$. The multi-refined graph is reported in Figure 4.vii. Any cell will produce at the next step a discrete partition after the individualization of its vertices, for this fragment of the whole graph. We conclude this section observing that for any colored graph in Miyazaki's sequence [24], the initial multi-refinement is able to produce the orbit partition.

Graph	(VE)	Aut	Orb	nauty (2.2) - can		bliss (0.35) - can		Traces - can			
				Time	Refine (ℓ)	Time	Refine (ℓ)	Time	MRef (ind)	Gns	Grp
ag2-16	(528,4352)	$2^{14} \cdot 3^2 \cdot 5^2 \cdot 17$	2	0.03	82 (5)	0.02	100 (4)	0.08	12 (3)	6	0.02
ag2-49	(4851,120050)	$2^{10} \cdot 3^2 \cdot 5^2 \cdot 7^6$	2	1.58	52 (5)	0.48	51 (4)	44.63	12 (3)	5	0.14
cf-20	(200,300)	2^{11}	80	0.03	146 (10)	0.01	262 (9)	0.03	28 (5)	11	0.01
cf-80	(800,1200)	2^{41}	320	3.81	1125 (36)	0.26	2802 (36)	3.05	118 (20)	41	1.19
cf-200	(2000,3000)	2^{101}	800	117.07	5597 (90)	5.30	33893 (92)	67.65	298 (50)	101	37.13
difp-20-0	(8965,23082)	1	8965	0.02	1 (1)	0.01	1 (0)	0.61	1 (0)	0	-
fpga-10-8	(688,1320)	$2^{23} \cdot 3$	519	0.05	300 (24)	0.01	300 (23)	0.52	71 (16)	21	0.16
fpga-13-11	(1500,3125)	$2^{32} \cdot 3^7$	1116	0.39	780 (39)	0.02	39 (38)	4.09	112 (26)	31	1.27
grid-3-20	(8000,22800)	$2^4 \cdot 3$	220	2.27	5 (2)	0.05	5 (1)	49.49	1 (0)	3	0.08
grid-w-2-100	(10000,20000)	$2^7 \cdot 5^4$	1	18.33	7 (3)	0.09	8 (2)	850.35	3 (1)	5	0.23
grid-w-3-20	(8000,24000)	$2^{10} \cdot 3 \cdot 5^3$	1	4.22	8 (3)	0.08	8 (2)	144.29	3 (1)	6	0.25
k-70	(70,2415)	$> 1.97e100$	1	0.01	2845 (70)	0.02	2485 (69)	3.08	148 (68)	8	3.04
k-100	(100,4950)	$> 9.33e157$	1	0.01	5050 (100)	0.05	5050 (99)	26.33	208 (98)	8	26.25
latin-30	(900,39150)	$2^6 \cdot 3^3 \cdot 5^2$	1	0.17	60 (5)	0.18	99 (3)	0.27	3 (1)	5	0.02
latin-sw-21-8	(441,13230)	1	441	5.44	26902 (3)	6.43	168022 (2)	1.23	442 (1)	0	-
latin-sw-30-11	(900,39150)	1	900	49.79	79255 (4)	46.32	731701 (2)	12.43	901 (1)	0	-
lattice-30	(900,26100)	$> 1.40e65$	1	0.30	930 (31)	0.40	901 (30)	9.90	122 (28)	33	8.18
mz-18	(360,540)	2^{39}	90	0.20	722 (21)	0.01	631 (20)	0.64	76 (18)	24	0.18
mz-50	(1000,1500)	2^{103}	250	> 24000	Timed out(*)	0.30	8653 (52)	24.76	204 (50)	56	8.09
mz-aug-22	(440,1012)	2^{47}	110	0.24	845 (25)	0.03	778 (24)	1.46	139 (23)	28	0.60
mz-aug-50	(1000,2300)	2^{103}	250	> 24000	Timed out(**)	> 24000	Timed out(**)	24.51	307 (51)	56	14.32
paley-461	(461,53015)	$2 \cdot 5 \cdot 23 \cdot 461$	1	0.01	6 (3)	0.07	6 (2)	0.52	3 (1)	2	0.00
pg2-32	(2114,34881)	$> 1.09e13$	1	4.70	912 (6)	0.24	858 (5)	1.10	14 (6)	6	0.13
rnd-3-reg-3000-1	(3000,4500)	1	3000	328.57	3001 (2)	0.39	3001 (1)	0.31	1 (0)	0	-
rnd-3-reg-10000-1	(10000,15000)	1	10000	42821.01	10001 (2)	5.60	10001 (1)	4.24	1 (0)	0	-
s3-3-3-3	(11076,20218)	2^{12}	7836	130.06	91 (13)	0.10	91 (12)	2053.71	43 (11)	12	1.28
urq8-5	(3906,20331)	1	3906	0.00	1 (1)	0.01	1 (0)	0.18	1 (0)	0	-

(*)9.68 secs, 3249 (52) nodes in *nauty-aut*; (**)3.34 secs, 3004 (53) nodes in *nauty-aut*; (***)0.14 secs, 3005 (53) nodes in *bliss-aut*;

Table 2: Graphs with small search tree