Hindawi Publishing Corporation
Journal of Applied Mathematics
Volume 2014, Article ID 934637, 15 pages
http://dx.doi.org/10.1155/2014/934637

Research Article

Novel Techniques to Speed Up the Computation of the Automorphism Group of a Graph

José Luis López-Presa,[1] Luis F. Chiroque,[2] and Antonio Fernández Anta[2]

[1] *DIATEL-UPM, 28031 Madrid, Spain*
[2] *IMDEA Networks Institute, 28918 Madrid, Spain*

Correspondence should be addressed to José Luis López-Presa; jllopez@diatel.upm.es

Received 3 April 2014; Accepted 24 June 2014; Published 24 July 2014

Academic Editor: Renat Zhdanov

Graph automorphism (GA) is a classical problem, in which the objective is to compute the automorphism group of an input graph. Most GA algorithms explore a search tree using the individualization-refinement procedure. Four novel techniques are proposed which increase the performance of any algorithm of this type by reducing the depth of the search tree and by effectively pruning it. We formally prove that a GA algorithm that uses these techniques correctly computes the automorphism group of an input graph. Then, we describe how these techniques have been incorporated into the GA algorithm conauto, as *conauto-2.03*, with at most an additive polynomial increase in its asymptotic time complexity. Using a benchmark of different graph families, we have evaluated the impact of these techniques on the size of the search tree, observing a significant reduction both when they are applied individually and when all of them are applied together. This is also reflected in a reduction of the running time, which is substantial for some graph families. Finally, we have compared the search tree size of conauto-2.03 against those of other popular GA algorithms, observing that, in most cases, conauto explores less nodes than these algorithms.

1. Introduction

Graph automorphism (GA), graph isomorphism (GI), and finding of a canonical labeling (CL) are closely related classical graph problems that have applications in many fields, ranging from mathematical chemistry [1, 2] to computer vision [3]. Their general time-complexity is still an open problem, although there are several cases for which they are known to be solvable in polynomial time. Hence, the construction of tools that are able to solve these problems efficiently for a large variety of problem instances has significant interest. This work focuses on the GA problem, whose objective is to compute the automorphism group of an input graph (e.g., by obtaining a set of generators, the orbits, and the size of this group). In this paper, novel techniques to speed up algorithms that solve the GA problem are proposed. Additionally, most of these techniques can be applied to increase the performance of algorithms for solving the other two problems as well.

1.1. Related Work. There are several practical algorithms that solve the GA problem. Most of them can also be used for CL (and consequently, for GI testing). For the last three decades, *nauty* [4, 5] has been the most widely used tool to tackle all these problems. Other interesting algorithms that solve GA and CL are *bliss* [6, 7], *Traces* [8], and *nishe* [9, 10]. Recently, McKay and Piperno have jointly released a new version of both nauty and Traces [11] with significant improvements over their previous versions. Another tool, named *saucy* [12–15], which solves GA (but not CL), has the advantage of being the most scalable for many graph families, since it is specially designed for efficiently processing big and sparse graphs. Recently, it was shown that the combined use of saucy and bliss improves the running times of bliss for the canonical labeling of graphs for a variety of graph families [16].

All these tools are based on the same principles, using variants of the Weisfeiler-Lehman individualization-refinement procedure [17]. They explore a search tree, whose nodes are equitable vertex partitions, using a backtracking algorithm to compute the automorphism group of the graph and, optionally, a canonical labeling. In more detail, using the Weisfeiler-Lehman individualization-refinement procedure, they generate a *first-path* from the root of this tree (which corresponds to the trivial partition) to a leaf (*first-leaf*, which is a partition where all cells are singleton). Then, using the same procedure, alternative branches of the tree are explored, backtracking, when a leaf is reached or a conflict is found. (A *conflict* is a partition that is not compatible with the partition at the same level in the first-path.) If no conflict is found, a leaf is reached that is compatible with that of the first-path, and an automorphism has been found. The efficiency of an algorithm depends on the speed at which it performs basic operations, like refinement, and, mainly, on the size of the search tree generated (the number of nodes of the search tree which are explored). There are two main ways to reduce the search space: pruning and choosing a good target cell (and vertex) for individualization.

Miyazaki showed in [18] that it is possible to make nauty choose bad target cells for individualization, so its search space becomes exponential in size when computing the automorphism group for a family of colored graphs. This suggests that a rigid criterion cell selector may be easily misled so that many nodes are explored, while choosing the right cells could dramatically reduce the search space. Thus, different colorings of a graph, or just differently labeled instances, may generate radically different search trees. Algorithms for CL use different criteria to choose the target cell for individualization. These criteria must be isomorphism invariant to ensure that the search trees for isomorphic graphs are isomorphic. However, this is not necessary for GA. Examples of cell selectors are the first cell, the maximum nonuniformly joined cell, the cell with more adjacencies to nonsingleton cells, and so forth. A cell selector immune to this dependency on the coloring or the labeling would be desirable.

Pruning the search tree may be accomplished using several techniques. Orbit pruning and coset pruning are extensively used by GA and CL algorithms. Perhaps, the most sophisticated pruning based on orbit stabilizer algorithms is that of the latest versions of nauty and Traces [11], which use the random Schreier method. However, when the number of generators grow, the overhead imposed might not be negligible in general. *Conflict propagation* is used by bliss [7] to prune sibling nodes when one of them generates a conflict which was not found in the corresponding node of the first-path. Conflicts may be detected at the nodes of the search tree, or during the refinement process as done by conauto [19] and saucy [14]. Conflicts can also be used to *backjump* several nodes in the search tree as done in [15]. In this case, it is necessary to update the backjump level of a node every time a conflict is found at that node.

Limited early automorphism detection, when a node has exactly the same nonsingleton cells (in the same position) as the corresponding (and compatible) node in the first-path, is present in all versions of conauto [20]. Recently, this feature has been added to saucy [14] under the name of *matching OPP pruning*. A more ambitious *component detection* was added to bliss [7] for early automorphism detection. However, components are not always easy to discover and keep track of.

1.2. Contributions. In this paper we propose a novel combination of four techniques to speed up GA algorithms. Most of these techniques can be applied to GI and CL algorithms as well. (Such extensions are out of the scope of this work.) These techniques can be used in GA algorithms that follow the individualization-refinement approach. One key concept that we define, which is used by some of the proposed techniques, is the property of a partition being a *subpartition* of another partition (see the definition in Section 3).

We propose a novel approach to *early automorphism detection* (EAD) which allows infering an automorphism without the need to reach a leaf-node of the search tree. The early automorphism detection in bliss [7] relies on component recursion which needs to identify components. However, component identification is not easy and the cell selectors must be aware of the structure to the graphs in order to select the cells that belong to the component currently being explored. This is specially difficult when components are structured in a multilevel fashion. Yet, our approach relies only on the structure of the partitions. Specifically, EAD is based on the concept of subpartition and its correctness is proved by Theorem 10. This technique is useful, for example, when the graph is built from regularly connected sets of isomorphic components or from components which have automorphisms themselves.

A second technique which, to our knowledge, has never been used in any other GA algorithm is *subpartition backjumping*, or *backjumping* (BJ) for short, in the search tree. BJ is done under the condition that the partition of the current node is a subpartition of its parent node. In this case, if the current node has been fully explored and no automorphism has been found, instead of backtracking to its parent node, it is possible to backtrack directly to another ancestor. Specifically, to the nearest ancestor of which the current node is not a subpartition. The correctness of BJ is proved by Theorem 11. This technique helps, for example, when not all the components in a component-based graph are isomorphic. Note that this backjumping only relies on the structure of the partitions at the nodes of the search tree, while the backjumping proposed in [15] relies on the conflicts found during the search for automorphisms. In fact, we compute the backjump points just after the generation of the first-path.

As previously stated, the target cell selector for individualization is key to yield a good search tree. We propose a *dynamic cell selector* (DCS) that tries to generate a tree in which nodes are subpartitions of their parent nodes, so that the previous techniques can be applied. If that is not possible, it chooses the vertex to individualize to be the one, among a nonisomorphism invariant subset of all the possible candidates that generates the partition with the largest number of cells. DCS adapts to a large variety of graph families. Since it is not isomorphism invariant, it cannot be applied for CL. However, it can be used for GA and, once the automorphism group has been computed, use it for CL.

This can be done in a way similar to the combined use of saucy and bliss proposed in [16].

The last technique proposed is *conflict detection and recording* (CDR). With this technique, in addition to recording a hash for each different conflict found exploring branches of the nodes of the first-path, the number of times each conflict appeared is counted. If the number of times a certain conflict has been found at a node (not in the first-path) exceeds the number of times it was found in the node at the same level of the first-path, then no more branches need to be explored in this node. This technique helps pruning the search tree in a large variety of graph families, and it is an improvement over the conflict propagation described in [7].

The original algorithm conauto [19] solves the GI problem but not the GA problem; conauto-2.0 is a modified version that computes automorphism groups and uses limited, though quite effective, coset and orbit pruning. We have implemented the four techniques described, and integrated them into our program conauto-2.0, resulting in the new version conauto-2.03. It is worth to mention that all versions of conauto process both directed and undirected graphs (in fact they consider all graphs as directed).

We have performed an analysis of the time complexity of conauto-2.03. It is easy to adapt prior analyses [19] to show that conauto-2.0 has asymptotic time complexity $O(n^3)$ with high probability when processing a random graph $G(n, p)$, for $p \in [\omega(\ln^4 n/n \ln\ln n), 1 - \omega(\ln^4 n/n \ln\ln n)]$ [21]. We then show that, in the worst case, the techniques proposed here increase the asymptotic time complexity of conauto-2.03 by an additive polynomial term with respect to that of conauto-2.0. In particular, DCS can increase the asymptotic time complexity in up to $O(n^5)$, while EAD and BJ in up to $O(n^3)$. Finally, CDR does not increase the asymptotic time complexity. Hence, if conauto-2.0 had polynomial time complexity for a graph family, the time complexity of conauto-2.03 would stay polynomial. Furthermore, as will be observed experimentally, the techniques added drastically reduce the search tree size (and the running time) in many cases.

We have experimentally evaluated the impact of each of the above techniques for the processing of several graph families and different graph sizes for each family. To do so, we have compared the number of nodes traversed by conauto-2.0 and the number of nodes traversed when each of the above techniques is applied. Then we have compared the number of nodes traversed and the running times of conauto-2.0 and conauto-2.03. The improvements are significant as the size of the search tree increases, and the overhead introduced is only noticeable for very small search trees. Finally, we have compared the search tree size of conauto-2.03 against those of nauty-2.5, Traces-2.5, saucy-3.0, and bliss-0.72, showing that in most cases conauto explores less nodes than these algorithms. In fact, there is only one family of graphs in the benchmark for which the search tree size of conauto-2.03 goes over the limit of 10^8 explored nodes imposed in the experiments.

1.3. Structure. The next section defines the basic concepts and notation used in the analytical part of the paper. In Section 3 we define the concept of subpartition and state the main theoretical properties, which imply the correctness of EAD and BJ. Then, in Section 4 we describe how these results have been implemented in conauto-2.03, and in Section 5 we evaluate the time complexity of conauto-2.03. In Section 6 we give an example of how these techniques can drastically reduce the size of the search tree. Finally, in Section 7 we present the experimental evaluation of conauto-2.03 (which implements the proposed techniques), concluding the paper with Section 8.

2. Basic Definitions and Notations

Most of the concepts and notations introduced in this section are of common use. For simplicity of presentation, graphs are considered undirected. However, all the results obtained can be almost directly extended to directed graphs.

2.1. Basic Definitions. A *graph* G is a pair (V, E), where V is a finite set and E is a binary relation over V. The elements of V are the *vertices* of the graph, and the elements of E are its *edges*. The set of graphs with vertex set V is denoted by $\mathcal{G}(V)$. Let $W \subseteq V$; the subgraph induced by W in G is denoted by G_W. Let $W \subseteq V$ and $v \in V$; we denote by $\delta(G, W, v)$ the number of neighbors of vertex v which belong to W. More formally, $\delta(G, W, v) = |\{(v, w) \in E : w \in W\}|$. If $W = V$, then it denotes the *degree* of the vertex. Let $W' \subseteq V$; if for all $v, w \in W'$, $\delta(G, W, v) = \delta(G, W, w)$, then this notation can be extended to denote the number of neighbors of W' which belong to W as $\delta(G, W, W')$.

Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are *isomorphic* if and only if there is a bijection $\gamma : V_G \to V_H$, such that $(v, w) \in E_G \Leftrightarrow (\gamma(v), \gamma(w)) \in E_H$. This bijection γ is an isomorphism of G onto H. An *automorphism* of a graph G is an isomorphism of G onto itself. The *automorphism group* $\mathrm{Aut}(G)$ is the set of all automorphisms of G with respect to the composition operation.

An *ordered partition* (or *partition* for short) of V is a list $\pi = (W_1, \ldots, W_m)$ of nonempty pairwise disjoint subsets of V whose union is V. The sets W_i are the *cells* of the ordered partition. For each vertex $v \in V$, $\pi(v)$ denotes the index of the cell of π that contains v (i.e., if $v \in W_i$, then $\pi(v) = i$). The number of cells of π is denoted by $|\pi|$. Let $A \subseteq V$, π^A denotes the partition of A obtained by restricting π to A. The set of all partitions of V is denoted by $\Pi(V)$. A partition is *discrete* if all its cells are singletons, and *unit* if it has only one cell. Let $\pi, \rho \in \Pi(V)$, then ρ is *finer* than π, if π can be obtained from ρ by replacing, one or more times, two or more consecutive cells by their union. Let $\pi = (W_1, \ldots, W_m)$ and $v \in W_i$; the partition obtained by *individualizing* vertex v is $\pi \downarrow v = (W_1, \ldots, W_{i-1}, \{v\}, W_i \setminus \{v\}, W_{i+1}, \ldots, W_m)$.

A *colored graph* is a pair $(G, \pi) \in \mathcal{G}(V) \times \Pi(V)$. Partition π assigns color $\pi(v)$ to each vertex $v \in V$. Let $\pi = (W_1, \ldots, W_m)$; for each vertex $v \in V$, its *color-degree vector* is defined as $d(G, \pi, v) = (\delta(G, W_i, v) : i = 1, \ldots, m)$. A colored graph (G, π) is *equitable* if for all $v, w \in V$, $\pi(v) = \pi(w)$ implies $d(G, \pi, v) = d(G, \pi, w)$. (i.e., if all vertices of the same color have the same number of adjacent vertices of each color.) The notion of isomorphism and automorphism can be extended to colored graphs as follows. Two colored graphs (G, π) and

(H,ρ) are isomorphic if there is an isomorphism γ of G onto H, such that $\gamma(v) = w$ implies $\pi(v) = \rho(w)$.

Two equitable colored graphs $(G,\pi) \in \mathcal{G}(V_G) \times \Pi(V_G)$ and $(H,\rho) \in \mathcal{G}(V_H) \times \Pi(V_H)$ are *compatible* if and only if (1) $|\pi| = |\rho| = m$; (2) let $\pi = (W_1,\dots,W_m)$ and $\rho = (W'_1,\dots,W'_m)$, then for all $i \in [1,m]$, $|W_i| = |W'_i|$; and (3) for all $v \in V_G$, $w \in V_H$, $\pi(v) = \rho(w)$ implies $d(G,\pi,v) = d(H,\rho,w)$. Note that if two colored graphs are not compatible, then they can not be isomorphic. Besides, two compatible colored graphs (G,π) and (H,ρ), such that π and ρ are discrete, are isomorphic.

2.2. Individualization-Refinement and Search Trees. Most algorithms for computing GA or CL use variants of the Weisfeiler-Lehman individualization-refinement procedure [17]. This procedure requires two functions: a *cell selector* and a *partition refiner*. A *cell selector* is a function S that, given a colored graph (G,π), returns the index i of a cell $W_i \in \pi$ such that $|W_i| > 1$. In the case of CL, S must be isomorphism invariant, that is, if (G,π) is isomorphic to (H,ρ), then $S(G,\pi) = S(H,\rho)$. Although this restriction is not necessary for automorphism group computation, provided that the selections made are stored for future use, most algorithms use isomorphism invariant cell selectors for automorphism group computation. A *partition refiner* is an isomorphism-invariant function R that, given a colored graph (G,π), returns either (G,π) if it is already equitable, or an equitable colored graph (G,ρ) such that ρ is finer than π. The partition refiners usually used are optimized versions of the 1-dim Weisfeiler-Lehman stabilization procedure.

The automorphism group of a graph is usually computed by traversing a search tree in a *depth-first* manner. A *search tree* of a graph $G \in \mathcal{G}(V)$ is a rooted tree $\mathcal{T}(G)$ of colored graphs defined as follows.

(1) The root of $\mathcal{T}(G)$ is the colored graph $R(G,(V))$. (We write $R(G,(V))$ and $S(G,\pi)$ instead of $R((G,(V)))$ and $S((G,\pi))$ to avoid duplicated parentheses).

(2) Let (G,π) be a node of $\mathcal{T}(G)$. If π is discrete, it is a leaf node.

(3) Otherwise, let $\pi = \{W_1,\dots,W_m\}$. Assume that $S(G,\pi) = j$ and $W_j = \{v_1,\dots,v_k\}$ (recall that $|W_j| > 1$ from the definition of a cell selector). Then, (G,π) has exactly k children, where the ith child is $(G,\pi_i) = R(G,\pi \downarrow v_i)$.

A *path* in $\mathcal{T}(G)$ starts at some internal (non-leaf) node and moves toward a leaf. A path can be denoted as $\pi_0[v_1\rangle\pi_1\cdots[v_k\rangle\pi_k$, indicating that, starting at node (G,π_0) and individualizing vertices $v_1,\dots,v_k$, node (G,π_k) is reached. The *depth* (or *level*) of a node in $\mathcal{T}(G)$ is determined by the number of vertices which have been individualized in its path from the root. Thus, if (G,π_0) is the root node, then π_0 is the partition at level 0, and π_k is the partition at level k. The first-path traversed in $\mathcal{T}(G)$ is called the *first-path*, and the leaf node of the first-path is called the *first-leaf*.

Theorem 1. *Let $G = (V,E)$ be a graph. Let (G,π) and (G,ρ) be two compatible leaf-nodes in $\mathcal{T}(G)$. Then, mapping $\gamma : V \to V$ such that, for all $v \in V$, $\pi(v) = \rho(\gamma(v))$ is an automorphism of G.*

Proof. Direct from the definition of compatibility among colored graphs, and the fact that, since (G,π) and (G,ρ) are leaf-nodes, all their cells are singleton. □

3. Correctness of EAD and BJ

In this section we define specific concepts needed to develop our main results, like the concept of *kernel* of a partition, and that of a partition being a *subpartition* of another partition. Then, using these concepts, we prove the correctness of the EAD and BJ techniques.

3.1. Definitions. We start by defining the *kernel* of a partition, which intuitively is the subset of vertices in non-singleton cells with edges to other vertices in non-singleton cells, but not to all of them. More formally, we can define the kernel as follows.

Definition 2. Let $(G,\pi) \in \mathcal{G}(V) \times \Pi(V)$ be an equitable colored graph, $\pi = (W_1,\dots,W_m)$ and $W = \bigcup_{i:|W_i|>1} W_i$. Then, the *kernel* of partition π is defined as $\kappa(\pi) = \{v \in W : \delta(G, W \setminus \{v\}, v) \in [1, |W|-1]\}$. The *kernel complement* of π is defined as $\overline{\kappa}(\pi) = (V \setminus \kappa(\pi))$.

Note that the kernel complement may contain non-singleton cells: those non-singleton cells whose vertices have no adjacencies with the vertices of the kernel. If such cells exist, then a simple EAD technique can be used to derive generators for a subgroup of the automorphism group of the graph in the following way.

Observation 1. Let (G,π) be an equitable colored graph. Let $\pi = (W_1,\dots,W_m)$. For each $i \in [1,m]$ such that $|W_i| > 1$ and $W_i \subseteq \overline{\kappa}(\pi)$, it holds that for each $u,v \in W_i$, $d(G,\pi^{\overline{\kappa}(\pi)},u) = d(G,\pi^{\overline{\kappa}(\pi)},v)$, and for all $v \in W_i$, $\delta(G,W_j,v) = 0$ for all $j \in [1,m]$ such that $|W_j| > 1$. Hence, u and v are inditinguishable from each other. Thus, any permutation of the vertices of W_i (fixing the remaining vertices of G) is an automorphism of G. In fact, they form a subgroup of the automorphism group of graph G.

A set of generators for this subgroup may be built in the following way: let $|W_i| = k$ and $W_i = \{v_1,\dots,v_k\}$. Then, $k-1$ generators, each of them defined by permuting vertex v_i with vertex v_{i+1} for all $i \in [1,k-1]$ generate this subgroup of size $k!$

Now we can define the concept of a partition being a subpartition of another partition, which is based on the kernel.

Definition 3. Let (G,π) and (G,ρ) be two equitable colored graphs such that ρ is finer than π. Then, ρ is a *subpartition* of π if and only if each cell in the kernel of ρ is contained in a different cell of π. (I.e., $\rho^{\kappa(\rho)} = \pi^{\kappa(\rho)}$).

3.2. Early Automorphism Detection. The next results allow for *early automorphism detection* (EAD) when, at some node in the search tree, the node's partition is a subpartition of an ancestor's partition. In practice, it limits the maximum depth in the search tree necessary to determine if a path is automorphic to a previously explored one.

Lemma 4. *Let $(G, \pi) \in \mathcal{G}(V) \times \Pi(V)$ be an equitable colored graph. Let $\pi = (W_1, \ldots, W_m)$. Then, for each vertex $v \in \overline{\kappa}(\pi)$, for all $i \in [1, m]$, $\delta(G, W_i, v) \in \{|W_i|, 0\}$.*

Proof. Let $A = \{v \in \overline{\kappa}(\pi) : |W_{\pi(v)}| = 1\}$. Since π is equitable and $|W_{\pi(v)}| = 1$ for all $v \in A$, the claim holds for all $v \in A$. Since the vertices in $(\overline{\kappa}(\pi) \setminus A)$ have no adjacencies with non-singleton cells, the claim also holds for these vertices. □

Lemma 5. *Let $(G, \pi), (H, \rho) \in \mathcal{G}(V) \times \Pi(V)$ be two equitable and compatible colored graphs. Let $\pi = (W_1, \ldots, W_m)$ and $\rho = (W_1', \ldots, W_m')$. For all $i \in [1, m]$, let β_i be any bijection from W_i to W_i'. For all $v \in V$, let $\gamma(v) = \beta_{\pi(v)}(v)$. Then, for all $v \in \overline{\kappa}(\pi)$, for all $i \in [1, m]$, $\delta(G, W_i, v) = \delta(G, W_i', \gamma(v)) \in \{|W_i|, 0\}$.*

Proof. Since (G, π) and (H, ρ) are equitable, from Lemma 4, for all $v \in \overline{\kappa}(\pi)$, for all $i \in [1, m]$, $\delta(G, W_i, v) \in \{|W_i|, 0\}$, and for all $v \in \overline{\kappa}(\rho)$, for all $i \in [1, m]$, $\delta(G, W_i', v) \in \{|W_i'|, 0\}$. For all $i \in [1, m]$, since (G, π) and (H, ρ) are compatible, $|W_i| = |W_i'|$, and for all $v \in V$, $\delta(G, W_i, v) = \delta(G, W_i', \gamma(v))$. Hence, for all $v \in \overline{\kappa}(\pi)$, for all $i \in [1, m]$, $\delta(G, W_i, v) = \delta(G, W_i', \gamma(v)) \in \{|W_i|, 0\}$. □

Corollary 6. *For all $u \in \overline{\kappa}(\pi), v \in V$, u is adjacent to v if and only if $\gamma(u)$ is adjacent to $\gamma(v)$.*

Definition 7. Let $G \in \mathcal{G}(V)$ and $\mathcal{T}(G)$ its search tree. Let (G, π_k) be a node of $\mathcal{T}(G)$. Let (G, π_l) and (G, ρ_l) be two descendants of (G, π_k) such that (1) they are compatible, and (2) π_l and ρ_l are subpartitions of π_k. Let $\pi_l = (W_1, \ldots, W_m)$ and $\rho_l = (W_1', \ldots, W_m')$. For all $i \in [1, m]$, let β_i be any bijection from W_i to W_i'. Let us define the function $\alpha : V \to V$ as follows

(i) For all $v \in \overline{\kappa}(\pi_l)$, $\alpha(v) = \beta_{\pi_l(v)}(v)$.

(ii) For all $v \in \kappa(\pi_l)$, $\alpha(v) = f(v)$, where $f(v) = v$ if $v \in \kappa(\rho_l)$, and $f(v) = f(\beta^{-1}(v))$ if $v \in \overline{\kappa}(\rho_l)$.

Observation 2. For all $i \in [1, m]$, α is a bijection from W_i to W_i'. Hence, α is a bijection (and a permutation of V).

Proof. Recall that β_i is a bijection from W_i to W_i' for all $i \in [1, m]$. Additionally, since π_l and ρ_l are subpartitions of π_k, for all $v \in \kappa(\pi_l)$, $v \in W_i$ implies $v \in W_i'$. □

Let us define the following subsets

$$\begin{aligned} A &= \overline{\kappa}(\pi_l) \cap \overline{\kappa}(\rho_l), \\ D &= \kappa(\pi_l) \cap \kappa(\rho_l), \\ B &= \kappa(\pi_l) \setminus D = \overline{\kappa}(\rho_l) \setminus A, \\ C &= \overline{\kappa}(\pi_l) \setminus A = \kappa(\rho_l) \setminus D. \end{aligned} \tag{1}$$

Similarly, for each $i \in [1, m]$, we define the following subsets

$$\begin{aligned} A_i &= A \cap W_i = A \cap W_i', \\ D_i &= D \cap W_i = D \cap W_i', \\ B_i &= B \cap (W_i \cup W_i'), \\ C_i &= C \cap (W_i \cup W_i'). \end{aligned} \tag{2}$$

Note that for all $i \in [1, m]$, $W_i \subseteq \overline{\kappa}(\pi_l)$ implies $W_i = A_i \cup C_i$, $W_i' \subseteq \overline{\kappa}(\rho_l)$ implies $W_i' = A_i \cup B_i$, $W_i \subseteq \kappa(\pi_l)$ implies $W_i = B_i \cup D_i$, and $W_i' \subseteq \kappa(\rho_l)$ implies $W_i' = C_i \cup D_i$.

Observation 3. α maps the vertices in $\overline{\kappa}(\pi_l)$ to the vertices in $\overline{\kappa}(\rho_l)$, the vertices in D to themselves, and hence, the vertices in B to the vertices in C.

Lemma 8. *For all $u \in B$, $v \in D$, u and v are adjacent if and only if $\alpha(u)$ and $\alpha(v)$ are adjacent.*

Proof. Take any vertex $x \in C$. Since $x \in \overline{\kappa}(\pi_l)$, then from Corollary 6, for all $v \in D$, x and v are adjacent if and only if $\alpha(x)$ and $\alpha(v)$ are adjacent. Note that, from the construction of α, $\alpha(v) = v$ and, either $\alpha(x) \in B$ or $\alpha(x) \in A$. In case $\alpha(x) = u \in B$, then from the construction of α, $\alpha(u) = \alpha(\alpha(x)) = f(\alpha^{-1}(\alpha(x))) = f(x) = x$. Hence, since α is a bijection of W_i onto W_i' for all $i \in [1, m]$, then for all $u \in B$ such that $\alpha^{-1}(u) \in C$, u and v are adjacent if and only if $\alpha(u)$ and $\alpha(v)$ are adjacent. Consider now the case in which $\alpha(x) \in A$. Note that, for all $a \in A$, $\alpha(a) \in (A \cup B)$. Then, from the construction of α, there is a sequence $(a_1, \ldots, a_n)$, such that $\alpha(x) = a_1$, for all $j \in [1, n]$, $a_j \in A$, for all $j \in [1, n-1]$, $\alpha(a_j) = a_{j+1}$, and $\alpha(a_n) = u \in B$. Hence, $\alpha^{n+1}(x) = u$, and $\alpha(u) = x$ from the construction of α. Note that, since α is a bijection, there are no two such sequences which share a vertex. Applying Corollary 6 to the powers of α, we get that $x = \alpha(u)$ and $v = \alpha(v)$ are adjacent if and only if $\alpha^z(x)$ and $\alpha^z(v)$ are adjacent. For the case $z = n+1$, we get that $\alpha(u)$ and $\alpha(v)$ are adjacent if and only if $\alpha^{n+1}(x) = u$ and $\alpha^{n+1}(v) = v$ are adjacent.

This applies to all $x \in C$ and $v \in D$. Since α is a bijection that maps B to C, the proof applies to all $u = \alpha^{-1}(x) \in B$. □

Lemma 9. *For all $u, v \in B$, u and v are adjacent if and only if $\alpha(u)$ is adjacent to $\alpha(v)$.*

Proof. Let $x, y \in C$. Like in the proof of Lemma 8, from the construction of α, there is a (smallest) z such that $\alpha^z(x) = u \in B$ and $\alpha(u) = x$ (from the construction of α), and a (possibly different smallest) z', such that $\alpha^{z'}(y) = v \in B$ and $\alpha(v) = y$ (also from the construction of α). Note that for all $h \in [1, z \cdot z']$, either $\alpha^h(x), \alpha^h(y) \in B$, or $\alpha^h(x) \in A$ or $\alpha^h(y) \in A$ (or both). Let $c = \mathrm{LCM}(z, z')$ be the *least common multiple* of z and z' (or a multiple of it). Then, $\alpha^c(x), \alpha^c(y) \in B$ and, for all $h \in [1, c-1]$, $\alpha^h(x) \in A$ or $\alpha^h(y) \in A$. Hence, Corollary 6 may be applied to conclude that $x = \alpha(u)$ and $y = \alpha(v)$ are adjacent if and only if $\alpha^c(x) = u$ and $\alpha^c(y) = v$ are adjacent.

This applies to all $x, y \in C$. Since α is a bijection that maps B to C, the proof applies to all $u = \alpha^{-1}(x) \in B$ and $v = \alpha^{-1}(y) \in B$. □

Theorem 10. *Let $G \in \mathcal{G}(V)$ and $\mathcal{T}(G)$ its search tree. Let (G, π_k) be a node of $\mathcal{T}(G)$. Let (G, π_l) and (G, ρ_l) be two descendants of (G, π_k) such that (1) they are compatible, and (2) π_l and ρ_l are subpartitions of π_k. Then, (G, π_l) and (G, ρ_l) are isomorphic, and α (as defined in Definition 7) is an automorphism of G.*

Proof. To prove that α is an isomorphism of (G, π_l) onto (G, ρ_l), we will prove that for all $u, v \in V$, u and v are adjacent if and only if $\alpha(u)$ and $\alpha(v)$ are adjacent. From Observation 3, there are four cases to consider:

(1) $u \in \overline{\kappa}(\pi_l)$, $v \in V$: Direct from Corollary 6;

(2) $u, v \in D$: since $D = \kappa(\pi_l) \cap \kappa(\rho_l)$, from the definition of α, $\alpha(u) = u$ and $\alpha(v) = v$. Hence, α applies the trivial automorphism of G_D;

(3) $u \in B$, $v \in D$: Direct from Lemma 8;

(4) $u, v \in B$: Direct from Lemma 9.

□

Interestingly, some of the properties used for early automorphism detection in other graph automorphism algorithms are special cases of the above theorem. For instance, the early automorphism detection used in saucy-3.0 is limited to the case in which all the nonsingleton cells are the same in both partitions. This corresponds to the particular case of Theorem 10 in which $\overline{\kappa}(\pi_l) \cap \kappa(\rho_l) = \emptyset$, and all the cells in $\overline{\kappa}(\pi_l)$ are singleton.

3.3. Backjumping. The following theorem shows the correctness of *backjumping* (BJ) when searching for automorphisms. This allows to backtrack various levels in the search tree at once.

Theorem 11. *Let (G, π_k) be a node of $\mathcal{T}(G)$. Let (G, π_l) and (G, ρ_l) be two compatible descendants of (G, π_k). Let (G, π_m) and (G, ρ_m) be two descendants of (G, π_l) and (G, ρ_l), respectively, such that π_m is a subpartition of π_l and ρ_m is a subpartition of ρ_l. If (G, π_m) and (G, ρ_m) are compatible but not isomorphic, then (G, π_l) and (G, ρ_l) are not isomorphic either.*

Proof. Assume otherwise. If (G, π_m) and (G, ρ_m) are isomorphic, (G, π_l) and (G, ρ_l) would be isomorphic too. From Theorem 10, all descendant nodes of (G, π_l) compatible with (G, π_m) (which is a subpartition of (G, π_l)) will be isomorphic to it. Then, since (G, π_m) and (G, ρ_m) are not isomorphic, (G, π_l) and (G, ρ_l) can not be isomorphic either. □

A direct practical consequence of Theorem 11 is that, when exploring alternative paths at level k, if a level m is reached that satisfies the conditions of the theorem, it is not necessary to explore alternative paths at level l. Instead, it is possible to backjump directly to the closest level $j \in [k, l)$ such that ρ_m is not a subpartition of ρ_j.

4. Implementation of the Techniques in *Conauto-2.03*

The starting point of algorithm conauto-2.03 is algorithm conauto-2.0, which is the first version of conauto that solves GA. It obtains a set of generators, and computes the orbits and the size of the automorphism group using the individualization-refinement approach. The cell selector of conauto-2.0 uses the following criteria: (a) a cell which has adjacencies with non-singleton cells is better than those which only have adjacencies to singleton cells, (b) among the cells that satisfy the previous criterion, the smallest cells are preferred, (c) among the cells that satisfy the previous criteria, the cells whose vertices have the highest number of adjacencies to vertices of nonsingleton cells are preferred, and (d) among those which satisfy the previous criteria, the cell with the smallest index is chosen.

In all the algorithms based on individualization-refinement, it has been experimentally observed that the cell selector determines, in most cases, the depth of the search tree, and the number of bad paths (those that do not yield an automorphism of the graph) in the search tree. However, no known cell selector yields an optimal search tree for all graph families. Besides, automorphisms discovered are used to prune the search tree. Yet, there are times when it is easy to know in advance that a path will be successful (will yield an automorphism of the graph). EAD can be used to generate such automorphism without the need to reach a leaf node, hence pruning the search tree quite effectively.

In conauto-2.03, the proposed techniques have been implemented with very good results. First, it must be noted that, in conauto-2.03, the leaf-nodes of the search tree are those which have a partition with an empty kernel, not necessarily those with discrete partitions. This already prunes the search tree, since vertices which are adjacent to the same vertices may remain in the same cell in the leaf nodes.

Additionally, EAD is implemented as follows. Once the first-path has been generated, the obtained partition is tested to see if Observation 1 can be applied to it. After that, the first-path is explored to find, for each non-leaf node (G, π), its nearest successor (G, ρ) which is a subpartition of (G, π). (Note that a leaf node is a subpartition of all its ancestors). (G, ρ) is recorded as the search limit for (G, π). Then, when searching for automorphisms from node (G, π), if a new node compatible with (G, ρ) is found, an automorphism is inferred applying Theorem 10, and a generator α is obtained applying Definition 7. This requires a subpartition test which is linear in the number of cells. The test will be executed, for each nonleaf node in the first-path, at most as many times as the length of the path from that node to the first-leaf. Every time the search limit is not a leaf, a subtree is pruned.

On its hand, BJ requires the execution of the subpartition test for the ancestors of each node (G, π) of the first-path, until a node (G, ρ) is found such that π is not a subpartition of ρ. That will be the backjump point for node (G, π). The point is recorded, and BJ can be subsequently applied with zero overhead. If there is no such ancestor, then that fact is also recorded. Thus, if a node compatible with (G, π) is reached during the search for automorphisms from an ancestor node and that path is unsuccessful, no more paths in the search tree will be tested at that ancestor (since no one could yield an automorphism, according to Theorem 11). Note that although it is out of the scope of this paper, this technique is used in the isomorphism testing algorithm of conauto-2.03, with good results.

EAD and BJ are only applied if there are nodes in the first-path that satisfy the subpartition condition. Without a cell selector that favours subpartitions, they cannot be expected to be useful in general. Hence, a cell selector like DCS is needed to choose a good cell for individualization. In conauto-2.03, DCS is implemented in the following way. At node (G, π), for each cell $c \in \kappa(\pi)$, it computes its size $s = |c|$ and degree $d = \delta(G, \kappa(\pi), c)$. For each pair of values (s, d), one cell is selected as a candidate for individualization. From each such cell, it takes the first vertex v, and computes the corresponding refinement $R(G, \pi \downarrow v)$. If it gets a partition which is a subpartition of π, it selects that cell (and vertex) for individualization. If no such cell is found, it selects the cell (and vertex) which produces the partition with the largest number of cells. Observe that this function is not isomorphism-invariant (not all the vertices of a cell will always produce compatible colored graphs), and it has a non-negligible cost in both time and number of additional nodes explored. However, it pays off because the final search tree is drastically reduced for a great variety of graphs, and other techniques compensate for the overhead introduced.

The implementation of conflict detection and recording (CDR) in conauto-2.03 requires the computation of a hash value for each conflict found. This way, conflicts may be identified by an integer value, what simplifies both recording and comparing conflicts. Additionally, an integer is associated to each conflict, indicating the number of vertices that generated that conflict. The cost in time and memory incurred by this computation is very limited and there is a large variety of graphs that benefit from this technique. Conflicts are recorded during the search for automorphisms at each node (G, π) of the first-path. Then, when searching for automorphisms at some ancestor node of (G, π), if a node compatible with (G, π) is reached, then several paths would need to be explored at this level. If a path finds an automorphism, no more paths need to be explored. If a path finds a bad-node (a node which is incompatible with the corresponding node of the first-path), then its hash value is computed. If it was not recorded as a valid conflict, no more paths are tested and this node is considered a bad-node. If this is a valid conflict, then the number of times this conflict has been found is incremented. If the number of times this conflict has been found is greater than the number of times it was originally found, this node is considered a bad-node. This way, bad-nodes are detected much faster than without CDR.

5. Complexity Analysis

It was shown in [19] that conauto-1.0 is able to solve the GI problem in polynomial time with high probability if at least one of the two input graphs is a random graph $G(n, p)$ for $p \in [\omega(\ln^4 n / n \ln \ln n), 1 - \omega(\ln^4 n / n \ln \ln n)]$. Using a similar analysis, it is not hard to show a similar result for the complexity of conauto-2.0 solving the GA problem. That is, conauto-2.0 solves the GA problem in polynomial time with high probability if the input graph is a random graph $G(n, p)$ for $p \in [\omega(\ln^4 n / n \ln \ln n), 1 - \omega(\ln^4 n / n \ln \ln n)]$.

We argue now that the techniques proposed in this work only increase the asymptotic time complexity of conauto-2.0 by a polynomial additive term. This implies that there is no risk that if a graph is processed in polynomial time by conauto-2.0, by using these techniques, it will require superpolynomial time with conauto-2.03. Let us consider each of the techniques proposed independently.

DCS only increases the execution time during the computation of the first-path. This follows since it is only used by the cell selector to choose a cell, and the cell selector is only used to choose the first-path. (Every time the cell selector returns a cell index, this index is recorded to be used in the rest of the search tree exploration.) The cell selector is called at most a linear number of times in n, where n is the number of vertices of the graph. Then, DCS is applied a linear number of times. Each time it is applied it may require to explore a linear number of branches. Each branch is explored with a call to the partition refiner function, whose time complexity if $O(n^3)$. Therefore, DCS increases the asymptotic time complexity of the execution by an additive term of $O(n^5)$. However, in our experiments, the increase in the number of nodes traversed is always far below this asymptotic bound.

Regarding EAD, like DCS, it requires additional processing while the first-path is created. In particular, for each partition π in the first-path, the closest partition down the path which is a subpartition of π is determined. This process always finishes, since the leaf of the first-path is a trivial subpartition of all the other partitions in the first-path. There is at most a linear number of partitions π and, hence, at most a linear number of candidate subpartitions. Moreover, checking if a partition is a subpartition of another takes at most linear time. Hence, EAD adds a term $O(n^3)$ to the time complexity of processing the first-path. On the other hand, when the rest of the search tree is explored, checking the condition to apply EAD has constant time complexity. If EAD can be applied, an automorphism is generated in linear time. Observe that if EAD were not used, then an equivalent automorphism would have been found, but at the cost of exploring a larger portion of the search tree (which takes at least linear time and may have up to exponential time complexity). Hence the application of EAD does not increase the asymptotic time complexity of exploring the rest of the search tree, and may in fact significantly reduce it.

The time complexity added by BJ to the processing of the first-path is similar to that of EAD, that is, $O(n^3)$, since for each partition in π the task is finding the closest partition up the first-path which is not a subpartition of π (if such a partition exists). The application of BJ in the exploration of the rest of the search tree takes constant time to check and to apply, while the time complexity reduction can be exponential.

CDR on its hand involves no processing during the generation of the first-path. Then, during the exploration of the rest of the search tree, every time a conflict is detected, the hash of that conflict is computed and the corresponding counter has to be updated (see Section 4). This takes in total at most linear time. Observe that conflict detection, which takes at least linear time, has to be done in any case. Hence, CDR does not increase the asymptotic time complexity of the algorithm.

6. Example of the Effectiveness of the New Techniques

Most algorithms that follow the individualization-refinement scheme work in the following way. They start by generating the first-path, recording which cells are used for individualization at each node of the first-path for future use. Then, starting from the first-leaf and moving towards the root, they explore each alternative branch in the search tree. When a leaf node compatible with the first-leaf is reached, an automorphism is found and a generator is stored. After all the branches of some node are either explored or discarded by automorphism pruning, the algorithm moves to the parent node to explore new branches of the search tree. This process continues until the root node of the search tree has been explored.

The sample graph shown in Figure 1 is used to illustrate the reduction, in the search tree size, attained with the combined use of DCS and EAD. This graph is a relabeling of the smallest graph of the TNN family described in the Appendix. The search tree obtained for this graph when using the cell selector of conauto-2.00 (and automorphism pruning) but no EAD is shown in Figure 2. (Note that the EAD based on Observation 1 is already used in conauto-2.00). This search tree has 75 nodes (not all of them have been numbered). Each branch is labeled with the vertex that is individualized. The partitions corresponding to the most relevant nodes of the search tree are shown in Table 1.

The root node (node 0) corresponds to the degree partition, which is already equitable. The cell selector used chooses the smallest cell, which is the leftmost one (see Table 1). Among the vertices of this cell, the first is chosen, namely a. After individualizing vertex a and subsequent refinement, node 1 is obtained. The next nodes of the first-path (denoted by solid lines) are generated in the same way. The first-leaf is node 11, which defines the base used for the automorphism group computation. Since, at node 10, vertex d was individualized to generate the first-leaf, vertex j is subsequently individualized to generate node 12, which is compatible with the first-leaf. Hence, an automorphism has been found and a generator of the automorphism group is stored. When node 39 is reached, an automorphism is dicovered which puts vertices m and u in the same orbit. At node 3, vertex m is individualized but node 40 is a bad-node (denoted by a striped pattern) since, as it can be easily seen in Table 1, the partition of node 40 is not compatible with that of node 4. Then, since vertices m and u are in the same orbit, it is not necessary to try vertex u. This is an example of orbit pruning. There are other examples of orbit pruning. For example, at the root node, vertices r, u and y are not considered because vertex r is in the orbit of b, and vertices u and y are in the same orbit as a and m. The total number of bad-nodes found directly determines the effectiveness of an algorithm. A search tree with no bad-nodes has a number of nodes which is polynomial in the number of nodes of the graph, since the number of leaves (base + generators of the automorphism group) is bounded by the number of nodes of the graph. In this case, the number of generators found is 10. This search tree is similar to those generated by nauty-2.5 and bliss-0.72 (without EAD).

Figure 3 shows the tree traversed when generating the first-path using DCS. Note that, in this case, at each level, several children nodes are explored before one is chosen (and revisited). At each node, the kernel of the corresponding colored graph is shown. At each branch, the individualized vertex is shown. The first benefit from using DCS is that the length of the first-path is shortened and, thus, the search tree is less deep than that of Figure 2. The nodes of the first-path are revisited to avoid extra storing. However, it pays off as it will be seen. DCS finds subpartitions twice, what can subsequently be used by EAD, applying Theorem 10. Remember that the leaf nodes are those with an empty kernel, not necessarily those with a discrete partition. Thus, Observation 1 is applied only at the first-leaf.

Figure 4 shows the search tree generated after DCS has been used in the generation of the first-path. In this case, EAD is extensively used to prune the search tree. The first-path is denoted by solid lines. The other paths explored are denoted with dashed lines. The first-leaf yields 4 generators (applying Observation 1), one for each nonsingleton cell in the kernel complement. Each other path explored yields a new generator (applying Theorem 10), what yields 4 more generators. Note that only one leaf node (apart from the first-leaf) is reached, since EAD makes it unnecessary in every other case. Thus, the total number of generators found is 8. In this example, the combined use of DCS and EAD allows for a reduction of the number of nodes of the search tree, from 75 to 17. Note that some of them are counted twice because they are revisited during the generation of the first-path. Besides, the number of bad-nodes is 0. That is, all the work done is useful.

7. Evaluation of the Techniques in *Conauto-2.03*

In this section we start showing how adding the proposed techniques to conauto-2.0 affects the size of the search tree. Then, we compare the size of the search tree of conauto-2.03 (which includes all the proposed techniques) against those of nauty, Traces, saucy, and bliss. This comparison shows how the application of these techniques to other algorithms could drastically improve their performance by reducing the size of their search trees.

The experiments have been carried out in an Intel(R) Core(TM) i5 750 @2.67 GHz, with 16 GiB of RAM under Ubuntu Server 9.10. All the programs have been compiled with gcc 4.4.1 and optimization flag "-O2," and all the results have been verified to be correct. For the experiments, we have used all the undirected graphs described in the Appendix, which includes a variety of graph families with different characteristics.

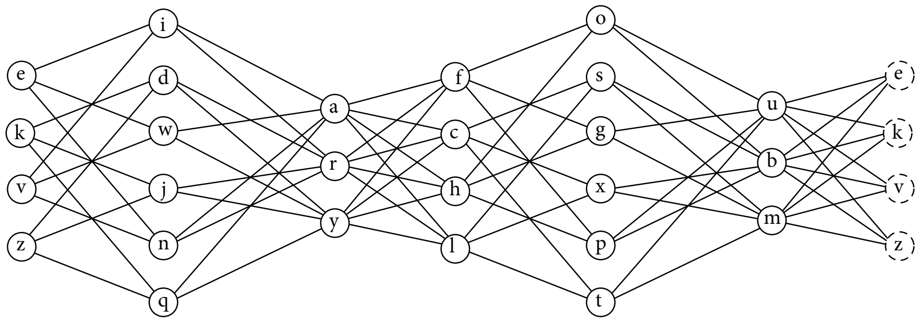

Figure 1: Sample graph used for automorphism group computation.

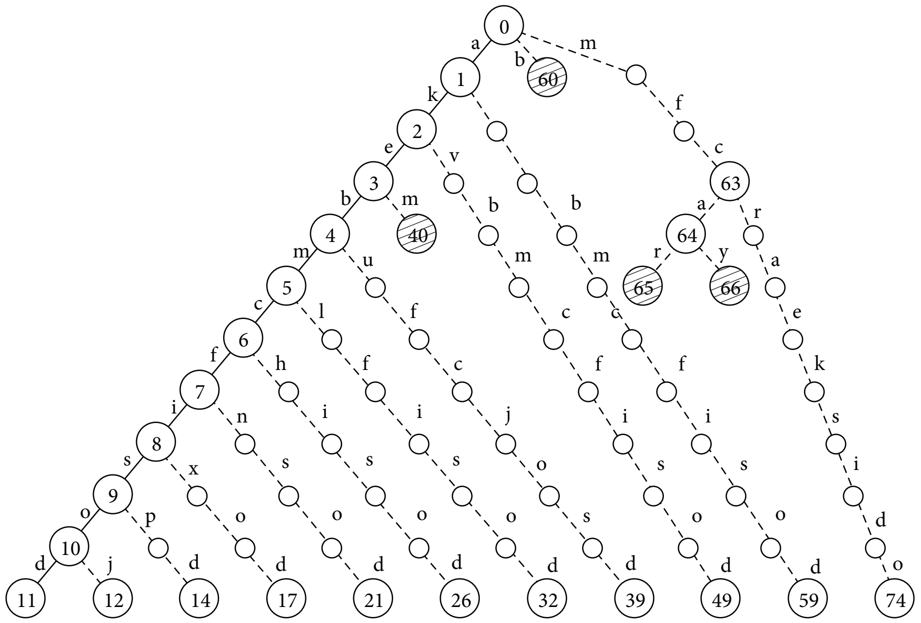

Figure 2: Search tree when the basic cell selector and no EAD are used.

7.1. First Experiment: Conauto-2.03 versus Conauto-2.0. First, we evaluate the proposed techniques separately. To do so, we consider the number of nodes that are explored during the search, since we consider this to be the key parameter that reflects the performance of an algorithm. The execution times measured present a similar behavior as the one shown by the number of nodes explored. The corresponding plots are shown in Figure 5. Then, we evaluate the impact of their joint use in conauto-2.03 with respect to conauto-2.0, on the size of the search tree and the running time. These plots are shown in Figure 6.

When counting the number of nodes of the search tree, each execution was terminated when the node count reached 10^8. For the time comparison, a timeout of 5,000 seconds was established. When an execution reached the limit, its corresponding point is placed on the boundary of the plotting area.

As can be observed in the plots, EAD, BJ, and CDR never increase the number of nodes explored. This number slightly increases with DCS in some graphs but only in a few executions with small search trees, and the benefit attained for most graphs is very noticeable. In fact, many executions that reached the count limit without DCS, lay within the limit when DCS is used (see the rightmost boundary of the plot).

In the case of component-based graphs with subsets of isomorphic components, EAD is able to prune many branches, but with other graph families it has no visible effect. That is why the diagonal of the plot is crowded. BJ has a similar effect but for different classes of graphs. It is mostly useful for component-based graphs which have few automorphisms, so they are complementary. EAD exploits the existence of automorphisms, and BJ exploits the absence of automorphisms.

CDR is useful with a variety of graphs. It is mostly useful when the target cells used for individualization are big and there are few automorphisms. It has been observed experimentally that when EAD or BJ are combined with DCS, their effect increases, since DCS favours the subpartition condition, generating more nodes in the search tree at which EAD and BJ are applicable. Hence, when all the techniques proposed are used together (in conauto-2.03), the gain is general (big search trees have disappeared from the diagonal),

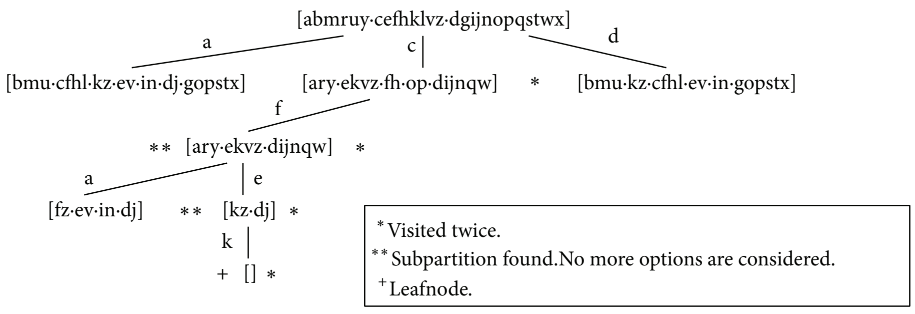
[abmruy·cefhklvz·dgijnopqstwx]
a
c
d
[bmu·cfhl·kz·ev·in·dj·gopstx]
[ary·ekvz·fh·op·dijnqw]
*
[bmu·kz·cfhl·ev·in·gopstx]
f
** [ary·ekvz·dijnqw] *
a
e
[fz·ev·in·dj]
** [kz·dj] *
k
+ [] *
* Visited twice.
** Subpartition found.No more options are considered.
+ Leafnode.

Figure 3: Generation of the first-path using DCS.

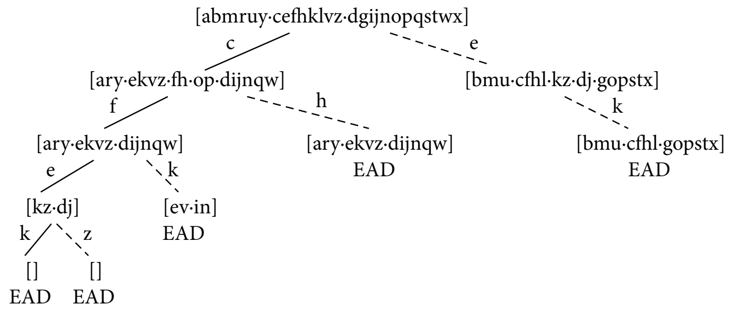

Figure 4: Search tree using DCS and EAD (other techniques do not apply here).

and the overhead generated by DCS is compensated by the other techniques in almost all cases.

The techniques presented help pruning the search tree, but they have a computational cost. Hence, we have compared the time required by conauto-2.0 and conauto-2.03, to evaluate the computation time paid for the pruning attained. The results obtained show that the improvement in processing time is general and only a few runs are slower (with running time below one second). Additionally, many executions that timed out in conauto-2.0 are able to complete in conauto-2.03 (see the rightmost boundary of the *time* plot in Figure 6).

7.2. Second Experiment: Conauto-2.03 versus Nauty-2.5, Traces-2.5, Saucy-3.0 and Bliss-0.72. In the second experiment, we compare the search-space of conauto-2.03 against those of nauty-2.5, Traces-2.5, saucy-3.0, and bliss-0.72. When counting the number of nodes of the search tree explored, each execution was terminated when the count reached $1.5 \cdot 10^8$ (observe that we are more permissive in this experiment). Again, when an execution reached the limit, its corresponding point is placed on the boundary of the plotting area. The plots are shown in Figure 7.

Comparing conauto-2.03 against nauty-2.5, we observe that in most cases conauto-2.03 explores many fewer nodes than nauty-2.5 and there are many cases in which nauty-2.5 exceeds the limit, while conauto-2.03 remains in values below 10^4 nodes. However, there are also some cases in which conauto-2.03 exceeds the limit, while nauty-2.5 explores around 10^4 nodes. All these cases correspond to graphs of the same family, the F-LEX-srg, which is the only one for which conauto-2.03 exceeds the limit (see Table 2). A very remarkable fact is that there are no cases in which both conauto-2.03 and nauty-2.5 exceed the limit.

Traces-2.5 generates search trees of one node for complete graphs, what explains the dots in the left boundary. Then, in the cases where Traces-2.5 remains below 10^3 nodes there is no clear winner between conauto-2.03 and Traces-2.5. Above that point, there are several cases in which Traces-2.5 exceeds the limit, and there are some cases in which conauto-2.03 is clearly worse that Traces-2.5. These are the case of the non-desarguesian projective planes of order 16 (PP16 family). In every case that conauto-2.03 exceeds the limit, Traces-2.5 does too.

In the case of saucy-3.0, conauto-2.03 is better in every case except the only family that makes conauto-2.03 exceed the limit, namely F-LEX-srg. However, saucy-3.0 is not as good as nauty-2.5 for this family. While saucy is especially suited for large sparse graphs, we have not compared the respective performance of saucy-3.0 and conauto-2.03 with these graphs because the latter currently has a limit on the size of the graphs it can process.

As it can be seen, bliss-0.72 is almost always worse than conauto-2.03. In fact, there are many cases in which bliss-0.72 exceeds the limit but conauto-2.03 does not, while there is no case in which conauto-2.03 exceeds the limit and bliss-0.73 does not.

In order to obtain a further per-graph-family performance information, we have compared the maximum search tree sizes for each algorithm and graph family in the

Table 1: Partitions at significant nodes of the search tree of Figure 2.

0	[abmruy·cefhklvz·dgijnopqstwx]	1	[a·y·r·bmu·cfhl·kz·ev·q·in·w·gopstx·dj]
2	[a·y·r·bmu·cfhl·k·z·ev·q·in·w·gopstx·dj]	3	[a·y·r·bmu·cfhl·k·z·e·v·q·in·w·gopstx·dj]
4	[a·y·r·b·mu·cfhl·k·z·e·v·q·in·w·opsx·gt·dj]	5	[a·y·r·b·m·u·cl·fh·k·z·e·v·q·in·w·sx·op·t·g·dj]
6	[a·y·r·b·m·u·c·l·fh·k·z·e·v·q·in·w·sx·op·t·g·dj]	7	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·in·w·sx·op·t·g·dj]
8	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·sx·op·t·g·dj]	9	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·s·x·op·t·g·dj]
10	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·s·x·o·p·t·g·dj]	11	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·s·x·o·p·t·g·d·j]
12	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·s·x·o·p·t·g·j·d]	14	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·s·x·p·o·t·g·d·j]
17	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·i·n·w·x·s·o·p·t·g·d·j]	21	[a·y·r·b·m·u·c·l·f·h·k·z·e·v·q·n·i·w·s·x·o·p·t·g·d·j]
26	[a·y·r·b·m·u·c·l·h·f·k·z·e·v·q·i·n·w·s·x·o·p·t·g·d·j]	32	[a·y·r·b·m·u·l·c·f·h·k·z·e·v·q·i·n·w·s·x·o·p·t·g·d·j]
39	[a·y·r·b·u·m·f·h·c·l·k·z·e·v·q·i·n·w·o·p·s·x·g·t·d·j]	40	[a·y·r·m·bu·cl·fh·k·z·e·v·q·in·w·gstx·op·dj]
49	[a·y·r·b·m·u·c·l·f·h·k·z·v·e·q·i·n·w·s·x·o·p·t·g·d·j]	59	[a·y·r·b·m·u·c·l·f·h·z·k·e·v·q·i·n·w·s·x·o·p·t·g·d·j]
60	[b·mu·ary·ekvz·cfhl·opsx·dijnqw·gt]	63	[m·u·b·ary·ekvz·f·h·c·l·g·sx·t·dijnqw·op]
64	[m·u·b·a·ry·ekvz·f·h·c·l·g·sx·t·inqw·dj·op]	65	[m·u·b·a·r·y·ev·kz·f·h·c·l·g·sx·t·in·w·q·dj·op]
66	[m·u·b·a·y·r·ev·kz·f·h·c·l·g·sx·t·w·q·in·dj·op]	74	[m·u·b·r·a·y·e·v·k·z·f·h·c·l·g·s·x·t·i·n·d·j·w·q·o·p]

Table 2: Maximum search tree size for each family in the benchmark.

Family	conauto-2.03	conauto-2.00	nauty-2.5	Traces-2.5	saucy-3.0	bliss-0.72
STH	15871	57401	57529	1825	365833	17903
LSN	107	59	63	50	557	59
LSP	24	15	28	25	38	22
PAN	19	21	18	17	19	14
PAP	9	8	8	9	9	8
LAT	719	1953	1953	1124	1925	629
TRI	678	990	990	1747	1937	990
USR	23847	158422	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	28544
CHH	545	18867	$>1.5 \cdot 10^8$	26421020	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$
TNN	1135	25680	$>1.5 \cdot 10^8$	61454925	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$
MZN	801	5747	5700	1466	$>1.5 \cdot 10^8$	2227
CMZ	1392	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	500	2348	$>1.5 \cdot 10^8$
MSN	743	5710	5678	1613	$>1.5 \cdot 10^8$	2495
MZA	567	5717	$>1.5 \cdot 10^8$	919	$>1.5 \cdot 10^8$	2580
MA2	705	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	204	641	11956
AG2	47	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	215	$>1.5 \cdot 10^8$	165
COM	1999	500500	500500	1	3995	500500
PG2	52	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	222	215	247
F-LEX-reg	6544	3035831	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	39054	12355
F-LEX-srg	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	295751	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$
HAD	461668	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	447452	$>1.5 \cdot 10^8$	458466
KEF	747	1865	2994	26558	900	1971
LKG	706	987	987	1760	1846	987
PTO	13	10	10	12	13	10
PP16	35211586	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$	63012	$>1.5 \cdot 10^8$	$>1.5 \cdot 10^8$
R1N	1	1	1	1	1	1
G2N	8	8	8	5	7	5

benchmark. The results are shown in Table 2. Thus, we can have an idea of the worst per-family behaviour of each algorithm. First of all, it is remarkable how much conauto-2.03 outperforms conauto-2.00 in some particular cases in which conauto-2.00 reached the limit, whilst conauto-2.03 keeps the search tree in manageable sizes (see the CMZ, MA2, AG2, PG2 and HAD families). The overload imposed by DCS in small search trees is noticeable, like in the LSN, LSP, PAP, and PTO families, but, as mentioned before, it tends to be compensated by the other techniques for big search trees.

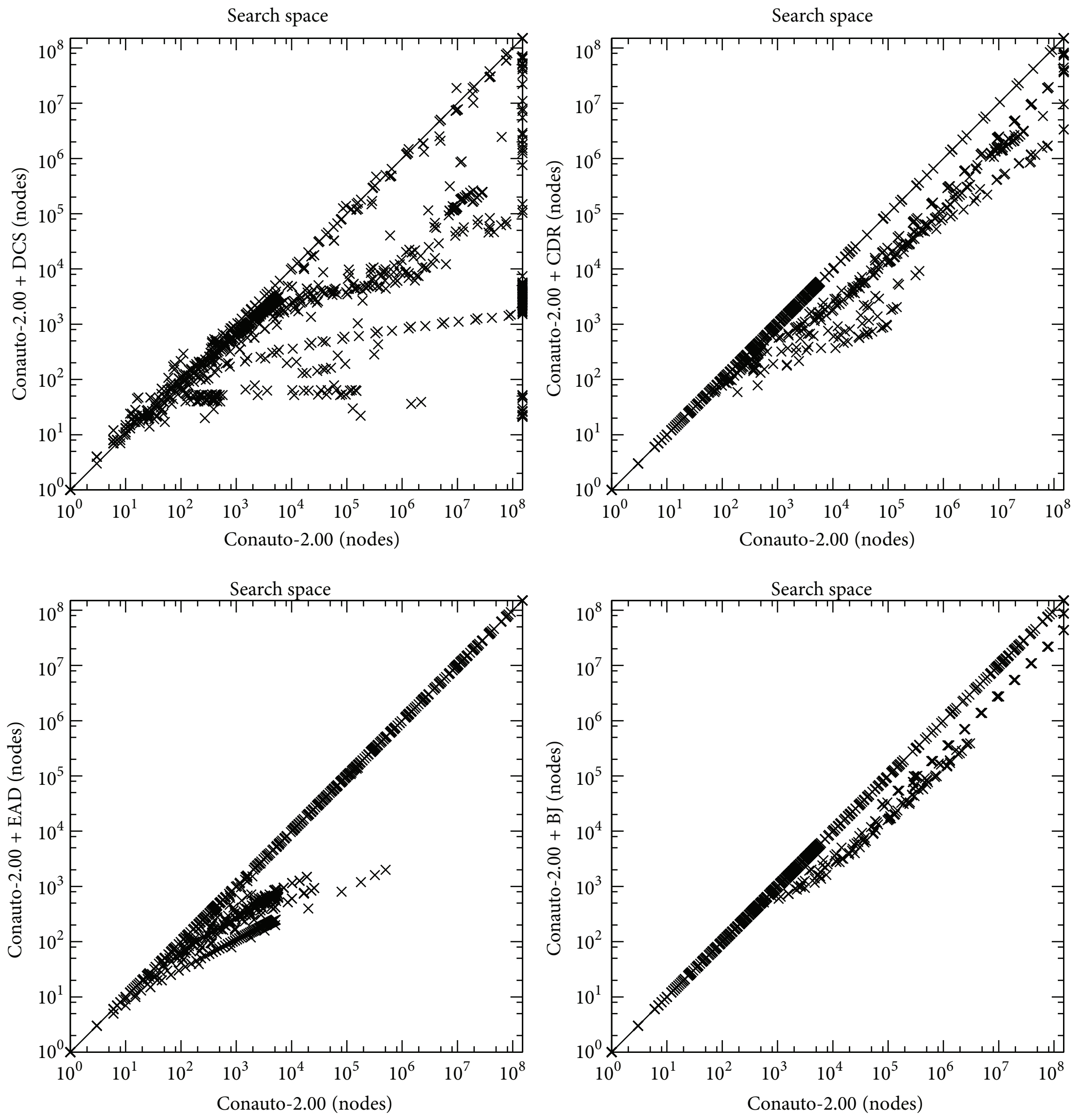

Figure 5: Search-Space comparison for the different techniques in conauto-2.0.

In a general comparison of conauto-2.03 with the other algorithms, we can say that the worst cases for conauto-2.03 are F-LEX-srg (where nauty-2.5 is the only one that did not reach the search tree limit) and PP16 (where Traces-2.5 and conauto-2.03 did not reach the limit, but Traces-2.5 outperforms conauto-2.03 by two orders of magnitude). In all the other cases, conauto-2.03 is very competitive with at least one algorithm.

8. Conclusions

We have presented four techniques than can be used to improve the performance of any GA algorithm that follows the individualization-refinement approach. In particular, a new way to achieve *early automorphism detection* has been proposed which is simpler and more general than previous approaches, and its correction has been proved. These techniques have been integrated in the algorithm conauto with only a polynomial additive increase in asymptotic time complexity. We have experimentally shown that, both isolated and combined, the proposed techniques drastically prune the search tree for a large collection of graph instances.

Appendix

A. Graph Benchmark

In this section we describe a wide range of graph families, which are used in our performance evaluation (Section 7). This benchmark can be found in [22]. Only undirected graphs have been considered, although directed versions of some graph families are also available in [22].

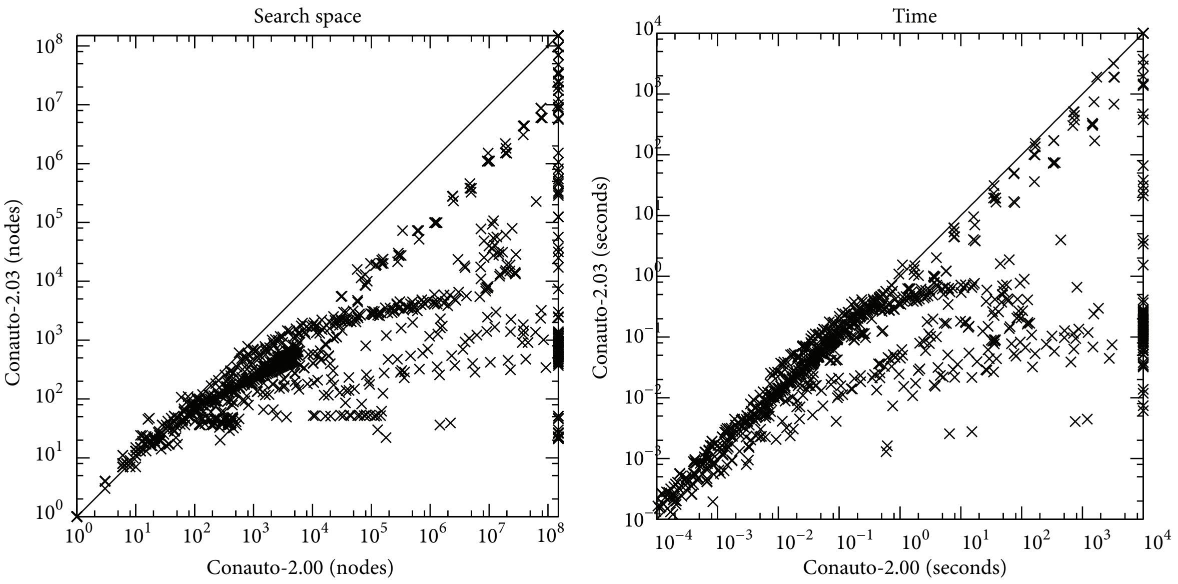

FIGURE 6: Conauto-2.00 versus conauto-2.03 search-space and running time comparison.

A.1. Strongly Regular Graphs

Steinter Triple Systems [STH]. These are the line graphs $\mathrm{srg}(v(v-1)/6, 3(v-3)/2, (v+3)/2, 9)$ of Steiner triple systems which have $(v(v+1)-2)/18$ orbits.

Latin Square [LSP, LSN]. This family consists of Latin square graphs $\mathrm{srg}(n^2, 3(n-1), n, 6)$, where n is the order of the Latin squares. They are split into two subfamilies: those of *prime order* (LSP) and those of *prime power order* (LSN). Prime power order Latin square graphs are usually harder than those of prime order.

Paley [PAP, PAN]. These are strongly regular graphs $\mathrm{srg}(q, (q-1)/2, (q-5)/4, (q-1)/4)$, where q is the order of the graphs. We have classified them in two subfamilies: those of prime order (PAP) and those of prime power order (PAN). Prime power order graphs are usually harder than prime order ones.

Lattice [LAT]. These are strongly regular graphs $\mathrm{srg}(n^2, 2(n-1), n-2, 2)$.

Triangular [TRI]. These are strongly regular graphs $\mathrm{srg}(q(q-1)/2, 2(q-2), q-2, 4)$.

A.2. Component Based Graphs

Unions of Strongly Regular Graphs [USR]. The graphs of this family are built using some Strongly Regular Graphs $\mathrm{srg}(29, 14, 6, 7)$ as basic components. Each vertex of each component is connected to all the vertices of all the other components. These graphs are extremely dense.

Cubic Hypo-Hamiltonian Clique-Connected [CHH]. The graphs of this family are built using two nonisomorphic cubic Hypo-Hamiltonian graphs with 22 vertices as basic components. Both graphs have four orbits of sizes: one, three, six, and twelve. A graph $CHH\_cc(m, n)$ has n complex components built from m basic components. The components of a complex component are connected through a complete m-partite graph using the vertices that belong to the orbits of size three of each basic component. The n complex components are interconnected with a complete n-partite graph using the vertices of each complex component that belong to the orbits of size one in the basic components.

Nondisjoint Unions of Undirected Tripartite Graphs [TNN]. We take two nonisomorphic digraphs with 13 vertices as basic components. Each of these components has 4 vertices with out-degree 3, 6 vertices with in-degree 4, and 3 vertices with out-degree 4. Then, each graph in the TNN family $\mathrm{tnn}(n)$ is generated taking n pairs of components and joining them by adding, for each vertex with out-degree 4, out-arcs connecting it to all the vertices with out-degree 3 of the other components of the graph and, finally transforming this digraphs into undirected graphs.

A.3. Miyazaki's Based Graphs

Base Construction [MZN]. This family contains the original construction of Miyazaki, not considering colours.

[CMZ]. This family is the "cmz" series of the bliss benchmark [23]. It is a variant of the original Miyazaki's construction.

Switched [MSN]. The family is obtained from the original construction of Miyazaki, changing one bridge for a switch.

Augmented [MZA, MA2]. These are the "mz-aug" series (MZA) of the bliss benchmark [23] and "mz-aug2" series (MA2) of the bliss benchmark [23].

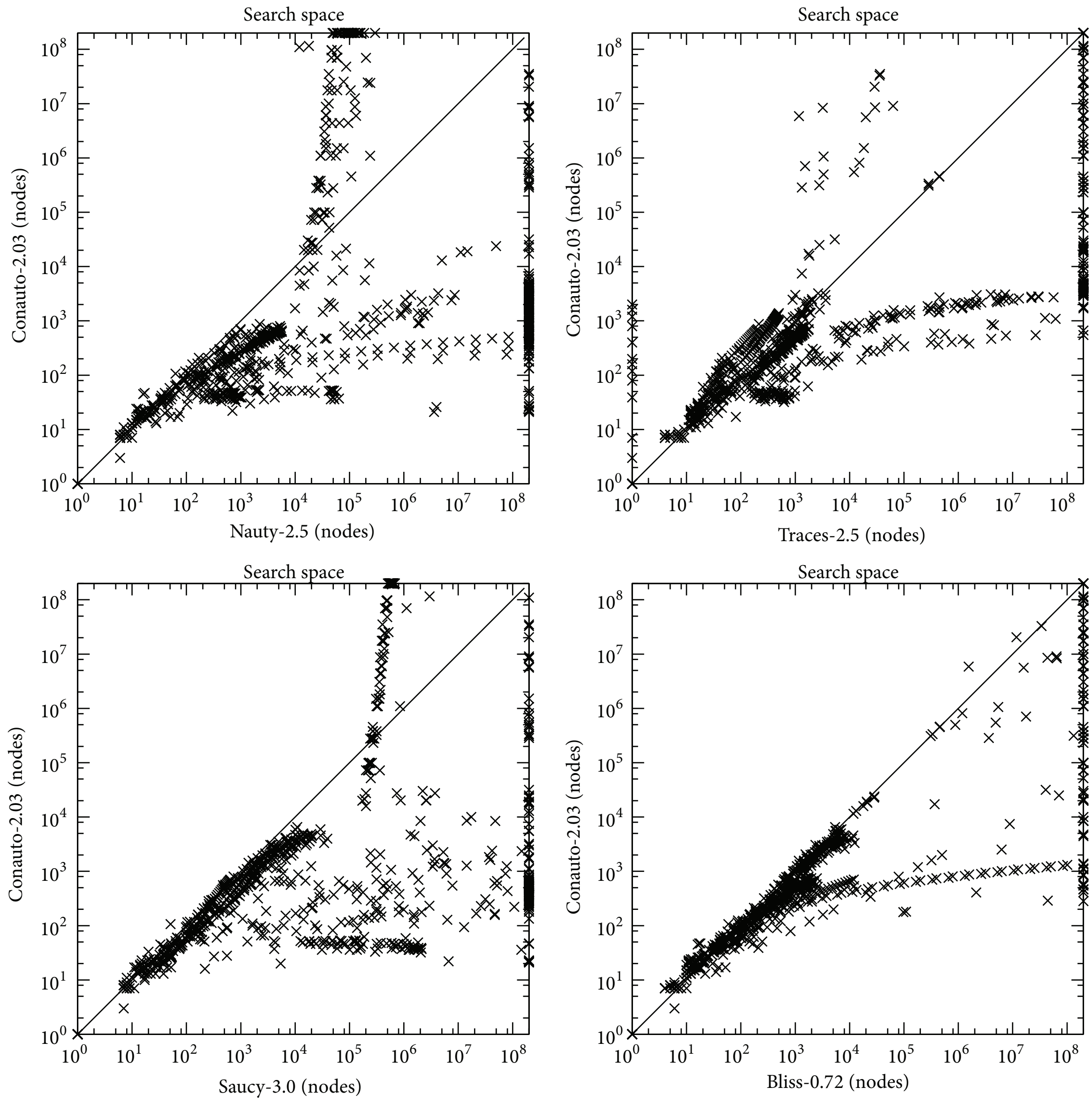

FIGURE 7: Comparison of the search-space of conauto-2.03 against nauty-2.5, Traces-2.5, saucy-3.0 and bliss-0.72.

A.4. Other Graph Families. Affine Geometries [AG2]. This family is the "ag" series of the bliss benchmark [23]. It contains point-line graphs of affine geometries $\mathrm{AG}_2(q)$.

Complete [COM]. This family contains simple undirected graphs, in which every pair of distinct vertices is connected by an edge.

Desarguesian Projective Planes [PG2]. This family contains the point-line graphs of Desarguesian projective planes $\mathrm{PG}_2(q)$.

F-Lex [F-LEX]. These graphs have been built by Petteri Kaski following a construction due to Pascal Schweitzer.

Hadamard [HAD]. This family contains graphs defined in terms of a Hadamard Matrix. It also includes the "had" series of the bliss benchmark [23].

Hadamard-Switched [HSW]. This family is the "had-sw" series of the bliss benchmark [23].

Kronecker Eye Flip Graphs [KEF]. This family comes from the "kef" series of the bliss benchmark [23].

Line Graphs of Complete Graphs [LKG]. This family contains the line-graphs of complete graphs (COM).

Paley Tournaments [PTO]. This family contains Paley tournaments (digraphs). The vertices of a paley tournament are the elements of the finite field F_q. There is an arc from vertex a to vertex b if and only if $a - b$ is a quadratic residue in F_q.

Projective Planes (Order 16) [PP16]. This family contains projective planes of order 16 [24].

Random [R1N]. This family comes from the SIVALab benchmark [25]. These are graphs in which there is an arc from a vertex u to a vertex v with probability 0.1.

Two-Dimensional Grids [G2N]. This family comes from the SIVALab benchmark [25]. These are the two-dimensional meshes in that benchmark.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This paper is supported in part by the Comunidad de Madrid Grant S2009TIC-1692, Spanish MINECO/MICINN Grant TEC2011-29688-C02-01, and National Natural Science Foundation of China grant 61020106002. A preliminary version of this work was presented at SEA 2013.

References

[1] J. L. Faulon, "Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs," *Journal of Chemical Information and Computer Sciences*, vol. 38, no. 3, pp. 432–444, 1998.

[2] G. Tinhofer and M. Klin, "Algebraic combinatorics in mathematical chemistry. Methods and algorithms III. Graph invariants and stabilization methods," Tech. Rep. TUM -M9902, Technische Universitat Munchen, 1999.

[3] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Graph matching applications in pattern recognition and image processing," in *Proceedings of the 2003 International Conference on Image Processing (ICIP '03)*, vol. 2, pp. 21–24, IEEE Computer Society Press, Barcelona, Spain, September 2003.

[4] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[5] B. D. McKay, *The Nauty Page*, Computer Science Department, Austra lian National University, 2010, http://cs.anu.edu.au/~bdm/nauty/.

[6] T. A. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX '07)*, pp. 135–149, January 2007.

[7] T. Junttila and P. Kaski, "Conflict propagation and component recursion for canonical labeling," in *Theory and Practice of Algorithms in (Computer) Systems*, vol. 6595 of *Lecture Notes in Computer Science*, pp. 151–162, Springer, Berlin, Germany, 2011.

[8] A. Piperno, "Search space contraction in canonical labeling of graphs (preliminary version)," CoRR, http://arxiv.org/abs/0804.4881.

[9] G. Tener and N. Deo, "Attacks on hard instances of graph isomorphism," *Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 64, pp. 203–226, 2008.

[10] G. Tener, *Attacks on difficult instances of graph isomorphism: sequential and parallel algorithms [Ph.D. thesis]*, University of Central Florida, 2009.

[11] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[12] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting structure in symmetry detection for CNF," in *Proceedings of the 41st Design Automation Conference*, pp. 530–534, ACM, June 2004.

[13] H. Katebi, K. A. Sakallah, and I. L. Markov, "Symmetry and satisfiability: an update," in *SAT*, vol. 6175 of *Lecture Notes in Computer Science*, pp. 113–127, Springer, 2010.

[14] H. Katebi, K. A. Sakallah, and I. L. Markov, "Conflict anticipation in the search for graph automorphisms," in *Logic for Programming, Artificial Intelligence, and Reasoning*, N. Bjorner and A. Voronkov, Eds., vol. 7180 of *Lecture Notes in Computer Science*, pp. 243–257, Springer, Heidelberg, Germany, 2012.

[15] P. Codenotti, H. Katebi, K. A. Sakallah, and I. L. Markov, "Conict analysis and branching heuristics in the search for graph automorphisms," in *Proceedings of the IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI '13)*, pp. 907–914, November 2013.

[16] H. Katebi, A. K. Sakallah, and L. I. Markov, "Graph symmetry detection and canonical labeling: differences and synergies," in *Turing-100, volume 10 of EPiC Series*, A. Voronkov, Ed., vol. 10 of *EPiC Series*, pp. 181–195, EasyChair, Manchester, UK, 2012.

[17] B. Weisfeiler, *On Construction and Identification of Graphs*, vol. 558 of *Lecture Notes in Mathematics*, Springer, Berlin, Germany, 1976.

[18] T. Miyazaki, "The complexity of McKay's canonical labeling algorithm," in *Groups and Computation II*, vol. 28, pp. 239–256, American Mathematical Society, 1997.

[19] J. L. Lopez-Presa and A. F. Anta, "Fast algorithm for graph isomorphism testing," in *Experimental Algorithms*, vol. 5526 of *Lecture Notes in Computer Science*, pp. 221–232, Springer, Berlin, Germany, 2009.

[20] J. L. López-Presa, *Efficient algorithms for graph isomorphism testing [Ph.D. thesis]*, La Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, Madrid, Spain, 2009, http://www.diatel.upm.es/jllopez/tesis/thesis.pdf.

[21] T. Czajka and G. Pandurangan, "Improved random graph isomorphism," *Journal of Discrete Algorithms*, vol. 6, no. 1, pp. 85–92, 2008.

[22] J. L. Lopez-Presa, *Benchmark Graphs for Evaluating Graph Isomorphism Algorithms*, Conauto Website by Google sites, 2011, http://sites.google.com/site/giconauto/home/benchmarks.

[23] T. Junttila, *Benchmark Graphs for Evaluating Graph Automorphism and Canonical Labeling Algorithms*, Laboratory for Theoretical Computer Science, Helsinki University of Technology, 2009, http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml.

[24] G. E. Moorhouse, "Projective planes of small order," Department of Mathematics, University of Wyoming, 2005, http://www.uwyo.edu/moorhouse/pub/planes/.

[25] M. de Santo, P. Foggia, C. Sansone, and M. Vento, "A large database of graphs and its use for benchmarking graph isomorphism algorithms," *Pattern Recognition Letters*, vol. 24, no. 8, pp. 1067–1079, 2003.