Fast Algorithm for Graph Isomorphism Testing*

José Luis López-Presa[1] and Antonio Fernández Anta[2,**]

[1] DIATEL, Universidad Politécnica de Madrid, 28031 Madrid, Spain
[2] LADyR, GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain

Abstract. In this paper we present a novel approach to the graph isomorphism problem. We combine a direct approach, that tries to find a mapping between the two input graphs using backtracking, with a (possibly partial) automorphism precomputing that allows to prune the search tree. We propose an algorithm, *conauto*, that has a space complexity of $O(n^2 \log n)$ bits. It runs in time $O(n^5)$ with high probability if either one of the input graphs is a $G(n,p)$ random graph, for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$. We compare the practical performance of conauto with other popular algorithms, with an extensive collection of problem instances. Our algorithm behaves consistently for directed, undirected, positive, and negative cases. Additionally, when it is slower than any of the other algorithms, it is only by a small factor.

1 Introduction

The Graph Isomorphism problem (GI) tests whether there is a one-to-one mapping between the vertices of two graphs, preserving the arcs. This is of both theoretical and practical interest. In practice, it has applications in many fields, like pattern recognition, computer vision, information retrieval, data mining, VLSI layout validation, and chemistry. Its main theoretical interest comes from the fact that, while GI is clearly in NP, it is not known if it is in P or NP-complete.

Previous work. As could be expected, GI has been extensively studied[1]. On the theoretical side, there is much work trying to place GI into a complexity class. There is strong evidence that GI is not NP-complete since, otherwise, the polynomial time hierarchy would collapse to its second level ($\Sigma_2^p = \Pi_2^p = \mathrm{AM}$) [4,16] and because it would be the only NP-complete problem to be polynomial-time equivalent in its decision and counting versions [10]. Recently, Arvind and Kurur [1] have shown that GI is in SPP ("Stoic PP"). GI is known to be solvable in polynomial time for some restricted classes of graphs, like trees or planar graphs [9]. However there are graph families that are specially hard, like certain families of strongly regular graphs (SRG) and projective planes. As far as we

* Partially supported by grants MICINN TIN2008-06735-C02-01, CAM S-0505/TIC/0285, and MEC PR2008-0015.

** Done in part while on leave at Alcatel-Lucent Bell Laboratories, Murray Hill, NJ.

[1] This review of the literature is necessarily incomplete. The reader can see the surveys of Reed and Corneil [15], Fortin [7], Goldberg [9], and Gati [8]. See also [14].

J. Vahrenhold (Ed.): SEA 2009, LNCS 5526, pp. 221–232, 2009.
© Springer-Verlag Berlin Heidelberg 2009

know, the best bound for general graphs up to now is due to Babai and Luks [3], whose canonical labeling (see below) algorithm runs in $\exp(n^{1/2+o(1)})$ time. GI has also been studied on random graphs $G(n,p)$. For $p = 1/2$, Babai et al. [2] proposed a canonical labeling algorithm that labels all graphs in expected linear time. Recently, Czajka and Padurangan [6] have given a linear time algorithm that canonically labels a $G(n,p)$ random graph with high probability[2], for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$.

GI algorithms use mainly two approaches. The *direct* approach tries to find an isomorphism between the two input graphs directly with a classical backtracking algorithm, possibly using heuristics to prune the search tree. Examples of direct algorithms are Ullman's [18] or vf2 [5]. The major drawback of these algorithms is that they are slow when the graphs being tested have many automorphisms, since they usually do not detect them. The *canonical labeling* approach applies some function $C()$ to each graph, which returns a *certificate* (canonical labeling) of the graph, such that $C(G) = C(H)$ if and only if graphs G and H are isomorphic. Nauty [11,12] is a canonical labeling algorithm that is currenly considered the fastest GI algorithm. The main problem of nauty, and any other complete canonical labeling algorithm, is that it needs to compute the whole automorphism group (which is hard). Not surprisingly, Miyazaki [13] has found a family of graphs with exponential lower time bounds for nauty.

Contributions. We propose an algorithm for GI that combines the best of the two approaches. Our algorithm, which we call *conauto*, is a direct algorithm since it tries to find a mapping between the two input graphs using backtracking. However, to drastically prune the search tree, it looks for automorphisms in the graphs, as canonical labeling algorithms do, but without necessarily computing the whole automorphism group. We show that our algorithm has a space complexity of $O(n^2 \log n)$ bits when run with n-node graphs. Additionally, using results of Czajka and Padurangan [6], we show that conauto runs in time $O(n^3)$ w.h.p. if either one of the input graphs is a $G(n,p)$ random graph, for $p \in [\omega(\ln^4 n/n \ln\ln n), 1-\omega(\ln^4 n/n \ln\ln n)]$.

We claim that conauto is very practical. To back this claim we compare it with other algorithms, namely nauty [12] and vf2 [5]. The former is included because it is considered to be the fastest practical GI algorithm, while the latter is included as a modern example of a direct algorithm. The comparison is done by running programs implementing the algorithms on an extensive benchmark that we have built [14], with positive and negative isomorphism cases, and directed and undirected graphs from several families. The benchmark used combines simple graph families, like random graphs, with other families that are known to be hard to handle by most GI algorithms, like some SRG families or the point-line graphs of Desarguesian projective planes. The comparison concludes that, when conauto is not able to handle a family of graphs (it cannot finish in 10,000 seconds), none of the other two can, while there are families that are handled easily by conauto and not by the others. Additionally, when it is slower than any of the other algorithms, it is only by a small factor. In general,

[2] W.h.p., probability at least $1 - O(n^{-c})$, for some $c > 0$ and large enough n.

conauto behaves more consistently in all cases (directed versus undirected, and positive versus negative). It is worth mentioning that an early version of conauto was recoded by Johannes Singler and included in the LEDA C++ class library of algorithms [17]. As noted in [17], both implementations (LEDA's and ours) of that early version of conauto have a very uniform behavior, but the LEDA implementation was found to be slower than ours. The version of conauto we present in this paper has a more complete search for automorphisms and uses them more exhaustively than the one included in LEDA.

Paper structure. In Section 2 we give basic definitions and notation. In Section 3 we describe the theoretic concepts on which the algorithm is based, while the algorithm is presented in Section 4. In Section 5 the asymptotic complexity of the algorithm is evaluated, and in Section 6 its practical performance is compared with nauty and vf2.

2 Definitions and Notation

A *directed graph* $G = (V, R)$ consists of a finite non-empty set V of vertices and a binary relation $R \subseteq V \times V$. An *arc* $(u, v) \in R$ is considered to be directed from u to v. R can be represented by an *adjacency matrix* $Adj(G) = A$ with size $|V| \times |V|$ in the following way.

$$A_{uv} = \begin{cases} 0 \text{ if } (u,v) \notin R \wedge (v,u) \notin R \\ 1 \text{ if } (u,v) \notin R \wedge (v,u) \in R \\ 2 \text{ if } (u,v) \in R \wedge (v,u) \notin R \\ 3 \text{ if } (u,v) \in R \wedge (v,u) \in R \end{cases}$$

Let $V_1 \subseteq V$, the *available degree* of v in V_1 under G, denoted by $ADg(v, V_1, G)$, is the 3-tuple (D_3, D_2, D_1) where $D_i = |\{u \in V_1 : A_{vu} = i\}|$ for $i \in \{1, 2, 3\}$. Extending the notation, we use $ADg(V_1, V_2, G) = d$ to denote that $\forall u, v \in V_1, ADg(u, V_2, G) = ADg(v, V_2, G) = d$, for $V_1, V_2 \subseteq V$. Let $ADg(V_1, V_2, G) = (D_3, D_2, D_1)$, then we define $Neigh(V_1, V_2, G) = D_3 + D_2 + D_1$ (i.e. the number of neighbors *each vertex* of V_1 has in V_2), and the predicate $Lnkd(V_1, V_2, G) = (Neigh(V_1, V_2, G) > 0)$. We say that $(D_3, D_2, D_1) \prec (E_3, E_2, E_1)$ when the first 3-tuple precedes the second one in lexicographic order. This notation will be used to order the available degrees of both vertices and sets.

Definition 1. *Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$. An* isomorphism *of G and H is a one-to-one mapping $m : V_G \longrightarrow V_H$ such that for all $u, v \in V_G$ $(v, u) \in R_G \iff (m(v), m(u)) \in R_H$.*

Graphs G and H are *isomorphic*, written $G \simeq H$, if there is at least one isomorphism of them. An *automorphism* of G is an isomorphism of G and itself.

Like other GI algorithms, conauto relies on vertex classification. This is performed using the available degree of the vertices, and refining the successive partitions in an iterative process. A *partition* of a set S is a sequence $\mathcal{S} = (S_1, ..., S_r)$ of disjoint nonempty subsets of S such that $S = \bigcup_{i=1}^{r} S_i$. The sets S_i are called

the *cells* of partition $\mathcal{S}$. The empty partition is denoted $\emptyset$. If $\mathcal{S} = (S_1, ..., S_r)$ and $\mathcal{T} = (T_1, ..., T_s)$ are partitions of two disjoint sets S and T, the *concatenation* of $\mathcal{S}$ and $\mathcal{T}$, denoted $\mathcal{S} \circ \mathcal{T}$, is the partition $(S_1, ..., S_r, T_1, ..., T_s)$. Clearly, $\emptyset \circ \mathcal{S} = \mathcal{S} = \mathcal{S} \circ \emptyset$.

Partitions may be refined by two means: vertex and set refinements. A vertex refinement classifies the vertices in each cell using the adjacency type they have with a *pivot vertex*. This way, each cell may be split into up to four subcells. A set refinement classifies the vertices in each cell using their available degree with respect to a *pivot set* (cell). Let $V_1, V_2 \subseteq V$, $SetPart(V_1, V_2, G)$ is the *set partition* of V_1 by V_2, which is a partition $(S_1, ..., S_r)$ of V_1 such that $\forall i, j \in \{1, ..., r\}$, $i < j$ implies $ADg(S_i, V_2, G) \succ ADg(S_j, V_2, G)$. If $V_2 = \{v\} \not\subseteq V_1$ we have the *vertex partition* of V_1 by v, denoted $VtxPart(V_1, v, G)$. Let $V_1 \subseteq V$, $\mathcal{S} = (S_1, ..., S_r)$ be a partition of V_1, and $P = S_x$ for some $x \in \{1, ..., r\}$ be a pivot set, then

1. The *vertex refinement* of $\mathcal{S}$ by the pivot vertex $v \in P$, denoted $VtxRef(\mathcal{S}, v, G)$, is the partition $\mathcal{T} = \mathcal{T}_1 \circ ... \circ \mathcal{T}_r$ such that $\forall i \in \{1, ..., r\}$, $\mathcal{T}_i = \emptyset$ if $\neg Lnkd(S_i, V_1, G)$, and $\mathcal{T}_i = VtxPart(S_i \setminus \{v\}, v, G)$ otherwise.
2. The *set refinement* of $\mathcal{S}$ by P, denoted $SetRef(\mathcal{S}, P, G)$ is the partition $\mathcal{T} = \mathcal{T}_1 \circ ... \circ \mathcal{T}_r$ such that $\forall i \in \{1, ..., r\}$, $\mathcal{T}_i = \emptyset$ if $\neg Lnkd(S_i, V_1, G)$, and $\mathcal{T}_i = SetPart(S_i, P, G)$ otherwise.

Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two graphs. Let $\mathcal{S} = (S_1, ..., S_r)$ and $\mathcal{T} = (T_1, ..., T_s)$ be partitions of $V_1 \subseteq V_G$ and $V_2 \subseteq V_H$ respectively, $\mathcal{S}$ and $\mathcal{T}$ are *compatible* under G and H, denoted $Comp(\mathcal{S}, \mathcal{T}, G, H)$, if $r = s$, and $\forall i \in \{1, ..., r\}$, $|S_i| = |T_i|$ and $ADg(S_i, V_1, G) = ADg(T_i, V_2, H)$.

A sequence of partitions starts with an initial partition (e.g., the degree partition) and each subsequent partition is obtained by applying some refinement to the previous one. A set refinement is labeled SET, and a vertex refinement is labeled VTX (from *vertex*) when the pivot set has only one vertex, and BTK (from *backtrack*) when it has more than one. More formally, a *sequence of partitions* for a graph $G = (V, R)$ is a tuple $(\mathsf{S}, \mathsf{R}, \mathsf{P})$, where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, are the partitions, $\mathsf{R} = (R^0, ..., R^{t-1})$ indicate the type of each refinement applied, and $\mathsf{P} = (P^0, ..., P^{t-1})$ are the pivot sets used. For all $i \in \{0, ..., t\}$, let $\mathcal{S}^i = (S^i_1, ..., S^i_{r_i})$, $V^i = \bigcup_{j=1}^{r_i} S^i_j$. Then the following statements must hold:

1. $\forall i \in \{0, ..., t-1\}$, $R^i \in \{\mathrm{VTX}, \mathrm{SET}, \mathrm{BTK}\}$, and $P^i \in \{1, ..., |\mathcal{S}^i|\}$.
2. $\forall i \in \{0, ..., t-1\}$, $R^i = \mathrm{SET} \Rightarrow \mathcal{S}^{i+1} = SetRef(\mathcal{S}^i, S^i_{P^i}, G)$.
3. $\forall i \in \{0, ..., t-1\}$, $R^i \neq \mathrm{SET} \Rightarrow \mathcal{S}^{i+1} = VtxRef(\mathcal{S}^i, v, G)$ for some $v \in S^i_{P^i}$.
4. $\forall x \in \{1, ..., r_t\}$, $\neg Lnkd(S^t_x, V^t, G) \vee |S^t_x| = 1$.

For convenience, for any $l \in \{1, ..., t-1\}$, we refer to the tuple $(\mathcal{S}^l, R^l, P^l)$ as *level* l. Level t is identified by $\mathcal{S}^t$, since R^t and P^t are not defined. Note that, at each refinement step, from the definitions of vertex and set refinements, the relative order of the vertices is preserved, and the vertices with no links are discarded. It is hence possible to define a (partial) order of the vertices of a graph, induced by a sequence of partitions, in the following way. Let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions for graph $G = (V, R)$. $\forall i \in \{0, ..., t\}$, let $\mathcal{S}^i = (S^i_1, ..., S^i_{r_i})$, and $V^i = \bigcup_{j=1}^{r_i} S^i_j$. Q induces a (partial) *order* $\prec_{\mathsf{Q}}$ in V as follows.

1. $\forall i \in \{0,...,t\}, \forall x,y \in \{1,...,r_i\}, x < y \Rightarrow \forall u \in S_x^i, \forall v \in S_y^i, u \prec_{\mathsf{Q}} v$.
2. $\forall i \in \{0,...,t-1\}, \forall x \in \{1,...,r_i\}, \forall u \in (S_x^i \setminus V^{i+1}), \forall v \in (S_x^i \cap V^{i+1}), u \prec_{\mathsf{Q}} v$.

An *Order* $<_{\mathsf{Q}}$ *induced by a sequence of partitions* Q is any total order that extends the order $\prec_{\mathsf{Q}}$. The i^{th} vertex with respect to $<_{\mathsf{Q}}$ is denoted $\omega_{\mathsf{Q}}(i)$.

Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two graphs. Let $\mathsf{Q}_G = (\mathsf{S}_G, \mathsf{R}_G, \mathsf{P}_G)$, and $\mathsf{Q}_H = (\mathsf{S}_H, \mathsf{R}_H, \mathsf{P}_H)$ be two sequences of partitions for graphs G and H respectively. Q_G and Q_H are said to be *compatible* if $|\mathsf{S}_G| = |\mathsf{S}_H| = t, |\mathsf{R}_G| = |\mathsf{R}_H| = t-1, |\mathsf{P}_G| = |\mathsf{P}_H| = t-1$, and they satisfy all the following. Let $\mathsf{R}_G = (R_G^0, ..., R_G^{t-1})$, $\mathsf{R}_H = (R_H^0, ..., R_H^{t-1})$, $\mathsf{P}_G = (P_G^0, ..., P_G^{t-1})$, $\mathsf{P}_H = (P_H^0, ..., P_H^{t-1})$, $\mathsf{S}_G = (\mathcal{S}^0, ..., \mathcal{S}^t)$, and $\mathsf{S}_H = (\mathcal{T}^0, ..., \mathcal{T}^t)$. Then

1. $\forall i \in \{0,...,t-1\}$, $R_G^i = R_H^i$, and $P_G^i = P_H^i$.
2. $\forall i \in \{0,...,t\}$, $Comp(\mathcal{S}^i, \mathcal{T}^i, G, H)$.
3. Let $\mathcal{S}^t = (S_1^t, ..., S_r^t)$, $\mathcal{T}^t = (T_1^t, ..., T_r^t)$, then $\forall x,y \in \{1,...,r\}$, $ADg(S_x^t, S_y^t, G) = ADg(T_x^t, T_y^t, H)$.

As will be seen, finding compatible sequences of partitions for two graphs gives an isomorphism between them, by just mapping the vertices in any of the orders induced by the sequences.

3 Theoretical Foundations

The algorithm conauto solves GI by trying to find compatible sequences of partitions for the input graphs. The following theorem shows that this in fact solves GI. All the proofs can be found in [14].

Theorem 1. *Two graphs G and H are isomorphic if and only if there are two compatible sequences of partitions Q_G and Q_H for graphs G and H respectively.*

Basically, conauto first constructs a sequence of partitions for one of the graphs, and then tries to find a compatible one for the other. Reproducing in the second sequence a refinement labeled SET or VTX is direct, since there is only one possible pivot set or vertex. However, a refinement labeled BTK implies several potential pivot vertices, what may lead to backtracking. The rest of this section explores how a limited automorphism search in the first graph can avoid some of this backtracking, transforming BTK into VTX for some refinements.

Two vertices $u, v \in V$ of a graph $G = (V, R)$ are *equivalent*, denoted $u \equiv v$, if there is an automorphism π of G such that $\pi(u) = v$. A vertex $w \in V$ is *fixed* by π if $\pi(w) = w$. When two vertices are equivalent, they belong to the same *orbit*. The set of all the orbits of a graph is called the *orbit partition*. Our algorithm performs a partial computation of the orbit partition incrementally, starting from the singleton partition. Since only a limited search for automorphisms is done, it is possible to stop before the orbit partition is really found. Then, only a semiorbit partition is obtained. A *semiorbit partition* of G is any partition $\mathsf{O} = \{O_1, ..., O_k\}$ of V, such that all vertices in O_i are equivalent, for all i.

Lemma 1. *At any level l of a sequence of partitions Q_G, all the vertices in a cell with no remaining links are mutually equivalent.*

Using this lemma, some equivalences are detected using only one sequence of partitions. However, conauto generates two sequences of partitions to detect most equivalences. From Theorem 1 and the definition of automorphism, it follows that two compatible sequences of partitions for a graph G define an automorphism of G. Let l be a backtracking level of a sequence of partitions Q_G (i.e., $R^l = \text{BTK}$), let $S^l_{P^l}$ be the pivot cell and $p \in S^l_{P^l}$ the pivot vertex used for the vertex refinement at level l. Consider any $p' \in S^l_{P^l}, p \neq p'$. Let Q'_G be a sequence of partitions compatible with Q_G, generated using p' as pivot instead of p at level l. Note that Q_G and Q'_G are equal up to level l. Let $<_{\mathsf{Q}_G}$ be an order induced by Q_G on the vertices of V, and let $<_{\mathsf{Q}'_G}$ be an order induced by Q'_G on the same set of vertices V. Then,

Lemma 2. *The mapping π induced by $<_{\mathsf{Q}_G}$ and $<_{\mathsf{Q}'_G}$, defined as $\pi(\omega_{\mathsf{Q}_G}(i)) = \omega_{\mathsf{Q}'_G}(i)$, $\forall i \in \{1, ..., |V|\}$, is an automorphism of G.*

Let k be such that $p = \omega_{\mathsf{Q}_G}(k)$, then $\pi(p) = p' = \omega_{\mathsf{Q}'_G}(k)$; $\forall j \in \{k, ..., |V|\}$, $\omega_{\mathsf{Q}_G}(j) \equiv \omega_{\mathsf{Q}'_G}(j)$; and π fixes vertices $\omega_{\mathsf{Q}_G}(1), ..., \omega_{\mathsf{Q}_G}(k-1)$. Two vertices $u, v \in V$ of a graph $G = (V, R)$ are *equivalent at level l*, denoted $u \equiv_l v$, if there is an automorphism of G that permutes them, and fixes all the vertices in $V \setminus V^l$ (i.e., those discarded in previous levels). Note that p and p' are equivalent at level l.

Lemma 3. *If $u \equiv_l v$, then $u \equiv_i v$, $\forall i \in \{0, ..., l-1\}$.*

Let $u \equiv_l v$, if $u \equiv_l p$, then $v \equiv_l p$, and if $u \not\equiv_l p$, then $v \not\equiv_l p$. This implies that when $u \equiv_l v$, their semiorbits can be merged at level l. Let us now extend the sequence of partitions to include a semiorbit partition.

Definition 2. *An* extended sequence of partitions E *for a graph $G = (V, R)$ is a tuple (Q, O), where Q is a sequence of partitions, denoted as SeqPart(E), and O is a semiorbit partition of G, denoted as Orbits(E).*

We observe now that when all the vertices in a pivot set used at a backtracking level l ($R^l = \text{BTK}$) are proved to be equivalent, R^l can be set to VTX, eliminating the backtracking point. This follows from the fact that automorphisms are preserved under isomorphisms, as stated in the following lemma.

Lemma 4. *If the vertices of a pivot set in a sequence of partitions Q_G for graph G are equivalent, then in a compatible sequence of partitions Q_H for graph H, the vertices in the corresponding pivot set must also be equivalent.*

The only information conauto stores about automorphisms is the semiorbit partition. Hence, with an extended sequence of partitions, it knows that two vertices are equivalent (but it does not know all the vertices that are fixed by an automorphism that permutes them). Nevertheless, for each two vertices u and v that belong to the same semiorbit in a semiorbit partition, there is at least one automorphism that fixes all the vertices that belong to singleton semiorbits, and permutes u and v.

Algorithm 1. Test whether G and H are isomorphic (*conauto*)

Iso(G, H) : boolean
1 **if** degree partitions do not match **then return** FALSE
2 $\mathsf{Q}_G \leftarrow$ *GenSeqOfPart*(G) ; $\mathsf{Q}_H \leftarrow$ *GenSeqOfPart*(H)
3 $\mathsf{E}_G \leftarrow$ *FindAuto*(G, Q_G) ; $\mathsf{E}_H \leftarrow$ *FindAuto*(H, Q_H)
4 **if** *BtkAmount*(*SeqPart*$(\mathsf{E}_G)) \leq$ *BtkAmount*(*SeqPart*$(\mathsf{E}_H))$
5 **then return** *Match*$(0, G, H,$ *SeqPart*$(\mathsf{E}_G),$ *Orbits*$(\mathsf{E}_H))$
6 **else return** *Match*$(0, H, G,$ *SeqPart*$(\mathsf{E}_H),$ *Orbits*$(\mathsf{E}_G))$

Algorithm 2. Generate a sequence of partitions for a graph G

GenSeqOfPart(G) : sequence of partitions
1 Start with the degree partition
2 **while** there are non-singleton cells with links **do**
3 **if** there is a singleton cell $\{v\}$ with links **then** label VTX; Refine by vertex v
4 **else** label SET; Refine by set exhaustively
5 **if** no set refinement succeeded **then** relabel BTK; Refine by vertex
6 label FIN
7 **return** the computed sequence of partitions

4 Algorithm *conauto*

In this section we present the algorithm *conauto* (Algorithm 1) which applies the previous theoretical discussion. If both graphs have the same vertex degrees, first it generates a sequence of partitions for each graph, and then tries to eliminate potential backtracking points looking for vertex equivalences at these backtracking points. Then, it chooses the graph with less backtracking levels (*BtkAmount*(), i.e., number of levels l with R^l = BTK) in its sequence of partitions as the target, and tries to find a compatible sequence of partitions for the other graph. If one such sequence of partitions is found, it returns TRUE. Otherwise it returns FALSE.

Algorithm 2, *GenSeqOfPart*, starts from the degree partition of the vertex set, and generates a sequence of partitions iteratively as follows.

1. If there are singleton cells in the partition, one of them is chosen as the pivot set, and a vertex refinement is performed to obtain the next partition in the sequence (Line 3).
2. Otherwise, the algorithm performs set refinements using different cells in the partition as pivot sets, until one of them is able to split at least one cell (maybe itself), or all of them have been tried unsuccessfully (Line 5).
3. If no cell meeting the conditions of Cases 1 and 2 has been found, then some cell is chosen as the pivot set, and a vertex in that cell is used as the pivot vertex to generate the new partition performing a vertex refinement (Line 6).

The search for automorphisms is performed by Algorithm 3. First it uses algorithm *ProcCellsWithNoLinks* to apply Lemma 1. Then traverses the sequence

Algorithm 3. Look for automorphisms

```
FindAuto(G, Q) : extended sequence of partitions
 1 O ← the singleton partition of V
 2 ProcCellsWithNoLinks(O)
 3 for each level l labeled BTK, in decreasing order of l do
 4     for each non-pivot vertex v in the pivot cell do
 5         Generate an alternative sequence of partitions using v
 6         if the sequences of partitions are compatible then
 7             ProcCompSeqsOfPart(O)
 8     if all the vertices in the pivot cell are equivalent then
 9         relabel original partition VTX
10 return (Q, O)
```

Algorithm 4. Find a sequence of partitions compatible with the target

```
Match(l, G, H, Q_G, O_H) : boolean
 1 if partition labeled VTX then
 2     success ⟵ Ref. by vertex are compat. and Match(l + 1, G, H, Q_G, O_H)
 3 else if partition labeled SET then
 4     success ⟵ Ref. by set are compat. and Match(l + 1, G, H, Q_G, O_H)
 5 else if partition labeled BTK then
 6     for each vertex v in the pivot cell, while not success do
 7         if v may not be discarded according to O_H then
 8             success ⟵ Ref. by vertex are compat. and Match(l + 1, G, H, Q_G, O_H)
 9 else (i.e. partition labeled FIN)
10     success ← adjacencies in both partitions match
11 return success
```

of partitions upwards looking for vertex equivalences among the vertices in the pivot sets at the levels labeled BTK, applying Lemma 2. This way, Lemma 3 will be applicable, so the automorphisms already found may be used when processing previous partitions in the sequence. The generation of an alternative sequence of partitions is performed in a straightforward way, avoiding bactracking. If this alternative sequence of partitions is compatible with the original one, then new vertex equivalences have been found, and they are used to iteratively compute the semiorbit partitions of the graphs using algorithm *ProcCompSeqsOfPart*.

When, at a backtracking point, all the vertices in the pivot cell are found to be equivalent, that level is relabeld from BTK to VTX. Recall that, from Lemma 4, this equivalence must hold for the other graph, so only one vertex in the corresponding pivot cell will need to be tested during the search for an equivalent sequence of partitions.

Algorithm 4 (*Match*) is a recursive algorithm that receives a level l to process in the sequence of partitions, the graphs G and H to test, the sequence of partitions Q_G for graph G, and the semiorbit partition $\mathsf{O_H}$ previously obtained for graph H. It returns TRUE if it is able to find a sequence of partitions for graph H that is compatible with Q_G, and FALSE otherwise. Algorithm 4 starts

with a partition that is compatible with the original (e.g., both start with the degree partition). Then, if the current level is labeled VTX, it applies a vertex refinement to the current partition. If the new partition generated is compatible with the original, it recursively calls itself to process the next partition in the sequence. Levels labeled SET are processed in a similar way, but applying a set refinement. If the current level is labeled BTK, it applies Lemma 2 to prune the search space. More sophisticated automorphism management may help here, but we have discarded for now that possibility in favor of simplicity. Hence, vertex equivalence will only be applied when all the previously fixed vertices belong to singleton semiorbits. At the last level (labeled FIN), Condition 3 from the definition of compatibility between sequences of partitions is tested.

The algorithm conauto directly applies the theoretical results from the previous section. Hence, the following theorem.

Theorem 2. *Two graphs G and H are isomorphic iff $Iso(G, H)$ returns* TRUE.

5 Complexity Analysis

Algorithm conauto requires to store the adjacency matrices and the sequences of partitions for each of the graphs. The matrices need $O(n^2)$ words for graphs of n vertices. (We assume words of $O(\log n)$ bits, since they need to store vertex identifiers.) Each partition may be represented using $O(n)$ words. It is not hard to see that a sequence of partitions has at most $2n$ partitions. Then, a sequence of partitions requires $O(n^2)$ words. Since at most three sequences have to be stored at any time (those of the graphs and a temporary sequence to find automorphisms), the sequences of partitions take $O(n^2)$ words. This yields a total amount of space required by conauto of $O(n^2)$ words, or $O(n^2 \log n)$ bits.

Regarding time, a careful analysis of each type of refinement gives that generating a new partition in a sequence takes at most time $O(n^2)$. Then, a sequence of partitions is built in time $O(n^3)$. In order to find automorphisms at most $O(n^2)$ sequences are created. Hence, creating a target sequence of partitions requires time $O(n^5)$. Now, the time to find a sequence of partitions compatible with the target directly depends on the number of backtracking points in the target sequence. If there are no backtracking points it is just the time to generate a sequence, $O(n^3)$ time. In general, let α be the number of backtracking points; then the time complexity is $O(n^{\max(\alpha+3,5)})$.

Finally, let us consider a random graph $G(n, p)$ for $p \in [\omega(\ln^4 n/n \ln\ln n), 1 - \omega(\ln^4 n/n \ln\ln n)]$. Sort the degrees of the neighbors of a vertex into its degree vector. Czajka and Pandurangan [6] have shown that, with high probability, no two vertices have the same degree vector, and that a canonical labeling for the graph is obtained from the lexicographic ordering of the degree vectors. If no two vertices have the same degree vector, conauto will generate a sequence of partitions without backtracking points, first obtaining the degree partition and then by repeatedly applying set refinements. Then, our algorithm will finish in time $O(n^5)$ with high probability if any of the graphs is a random graph $G(n, p)$.

6 Performance Evaluation

In this section we compare the performance of an implementation of conauto with the two other programs of reference: *nauty* and *vf2*. The tests have been carried out in a Pentium III at 1.0 GHz with 256 MB of main memory, under Linux RedHat 9.0. All the programs have been compiled with the same compiler, GNU's gcc, and using the same optimization flags. The execution time considered is the real time (not CPU time) consumed by the programs, excluding loading time (the time needed by the programs to load from disk the graphs being tested). The CPU time limit for each program run was set to $10,000$ seconds. If a program was unable to finish within this CPU time limit for a pair of graphs of some size, all its tests for that and bigger sizes were discarded. Some of the curves obtained have been omitted due to space restrictions. They can be found, with a detailed description of the benchmark used in the evaluation, at [14].

The first graphs considered are random graphs $G(n, 0.1)$ (only isomorphic cases). As expected, all algorithms run very fast with these graphs, finishing in less than a second even for graphs of $1,000$ nodes. However, vf2 is one order of magnitude worse than the other programs, nauty being the fastest. The second family of graphs are 2D-meshes. In this case, for undirected graphs all algorithms behave similarly, finishing in, at most, a few seconds (for $1,000$ nodes). A difference in behavior is observed for directed graphs. While conauto behaves as with undirected 2D-meshes, the time of nauty increases and the time of vf2 decreases, both in about one order of magnitude. The next family of graphs considered are Paley graphs, a subclass of SRGs. In this case all programs run in reasonable time (at most tens of seconds). It may be worth to note that vf2 is more than 2

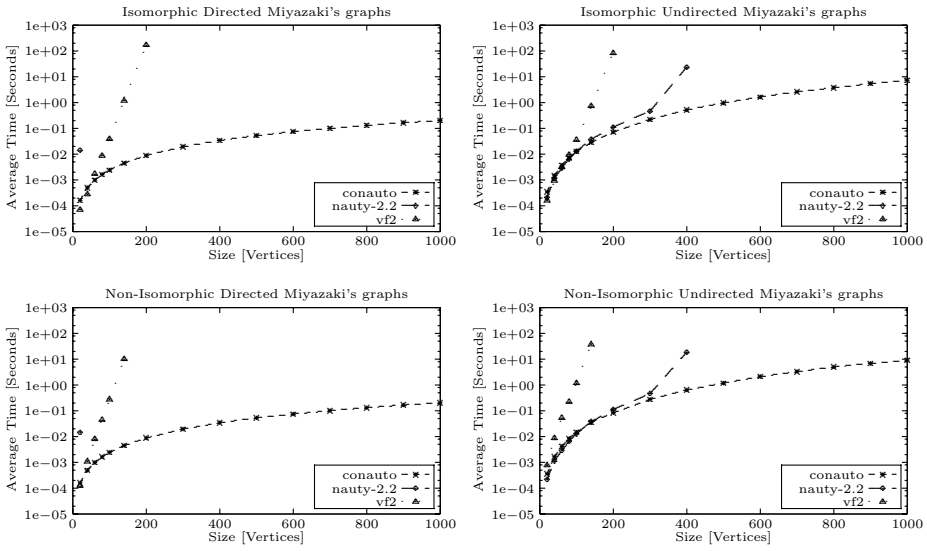

Fig. 1. Performance of conauto with Miyazaki's graphs

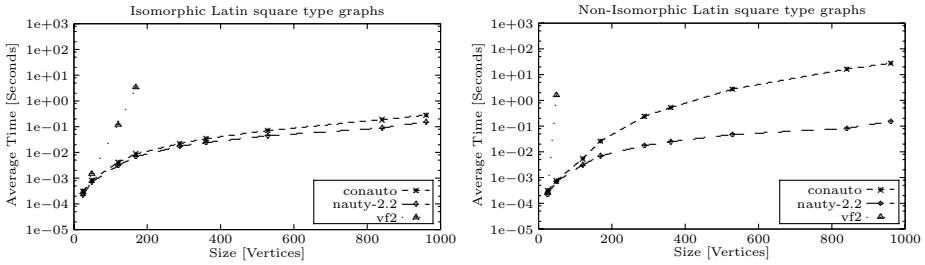

Fig. 2. Performance of conauto with Latin square graphs

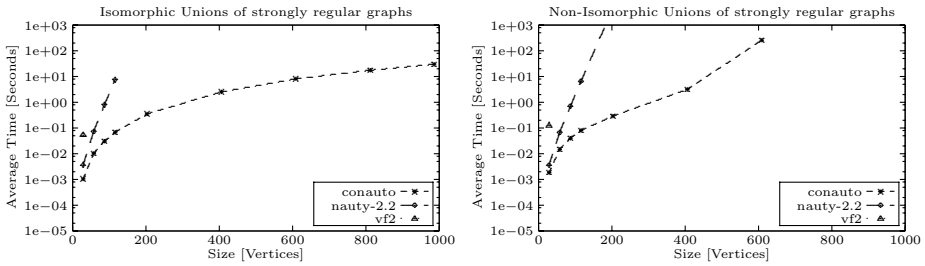

Fig. 3. Performance of conauto with unions of strongly regular graphs

orders of magnitude slower than conauto and nauty. For triangular graphs and lattice graphs, also subclasses of SRGs, we observe a symmetric phenomenon: all programs run fast (at most a few seconds) and vf2 is about one order of magnitude faster.

The first family of graphs in which a substantial difference in behavior can be observed are Miyazaki's graphs (see Figure 1). These are known to be very hard graphs for nauty [13] (e.g., with the directed version, it is not able to label graphs of 40 vertices in $10,000$ s.). As can be seen in the figure, this family of graphs is only handled fast by conauto, which always finishes in a few seconds. The other algorithms cannot go beyond 400-node graphs (200 nodes if directed). A second interesting family are Latin square graphs, which are SRGs. For this family vf2 is not able to finish beyond graphs of 200 nodes (see Figure 2). Additionally, while nauty has the same low running time for positive and negative cases, conauto shows good (similar to nauty) running times for positive cases but about 2 orders of magnitude more for negative cases. The third interesting family of graphs are those obtained as unions of SRGs with the same parameters $(29, 14, 6, 7)$ (see the results in Figure 3). These graphs are already known to make nauty exponential in time (cf. [13]). For vf2, they are so hard, that it can only finish within time with graphs of one component. On the other hand, conauto runs reasonably fast for positive cases, and faster than the others for the negative cases. However, it can not find an answer for graphs above 600 vertices for non-isomorphic pairs of graphs. The hardest family we have in our benchmark are point-line graphs of

Desarguesian projective planes. For this family none of the programs is able to deal with graphs of more than 200 vertices.

References

1. Arvind, V., Kurur, P.P.: Graph isomorphism is in SPP. In: FOCS 2002 (2002)
2. Babai, L., Kučera, L.: Canonical labeling of graphs in linear average time. In: FOCS 1979 (1979)
3. Babai, L., Luks, E.M.: Canonical labeling of graphs. In: STOC 1983 (1983)
4. Boppana, R., Hastad, J., Zachos, S.: Does co-NP have short interactive proofs? Information Processing Letters 25, 27–32 (1987)
5. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In: Proc. of the 3rd IAPR-TC-15 Int'l Workshop on Graph-based Representations (2001)
6. Czajka, T., Pandurangan, G.: Improved random graph isomorphism. J. of Discrete Algorithms 6(1), 85–92 (2008)
7. Fortin, S.: The graph isomorphism problem. Technical Report TR 96-20, U. of Alberta, Edmonton, Alberta, Canada (July 1996)
8. Gati, G.: Further annotated bibliography on the isomorphism disease. J. of Graph Theory 3(2), 95–109 (2006)
9. Goldberg, M.: The graph isomorphism problem. In: Gross, J.L., Yellen, J. (eds.) Handbook of graph theory. Discrete Mathematics and its Applications, ch. 2.2, pp. 68–78. CRC Press, Boca Raton (2003)
10. Mathon, R.: A note on the graph isomorphism counting problem. Information Processing Letters 8(3), 131–132 (1979)
11. McKay, B.D.: Practical graph isomorphism. Congr. Numer. 30, 45–87 (1981)
12. McKay, B.D.: The nauty page. CS Dept., Australian National U. (2004), `http://cs.anu.edu.au/~bdm/nauty/`
13. Miyazaki, T.: The complexity of McKay's canonical labeling algorithm. In: Finkelstein, L., Kantor, W.M. (eds.) Groups and Computation II. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 28 (1997)
14. López Presa, J.L.: Efficient Algorithms for Graph Isomorphism Testing. Ph.D thesis, ETSI Telecomunicación, U. Rey Juan Carlos, Madrid, Spain (March 2009), `http://www.diatel.upm.es/jllopez/tesis/thesis.pdf`
15. Read, R.C., Corneil, D.G.: The graph isomorphism disease. J. of Graph Theory 1, 339–363 (1977)
16. Schöning, U.: Graph isomorphism is in the low hierarchy. J. of Computer and System Sciences 37(3), 312–323 (1988)
17. Singler, J.: Graph isomorphism implementation in LEDA 5.1. Technical report, Algorithmic Solutions Software GmbH (December 2005)
18. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM 23(1) (1976)