UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Universitaria
de
Ingeniería Técnica de Telecomunicaciones

PROYECTO FIN DE CARRERA

Use of Automorphisms in Conauto-2.0

Luis Felipe Núñez Chiroque

October, 2011

Acknowledgements

I would like to thank my tutor José Luis López Presa for all his help and collaboration in the realization of this final project. I would also thank Professor Antonio Fernández Anta for his help with the theoretical part of this work.

I also thank each and every one of the teachers I have had throughout my training since I was a child, of which I have always learned something.

Thank you to my colleagues, with whom I have been progressing side by side, helping and supporting each others.

And last but not least, I need to thank my family, all of them, especially my parents, who always have given me all the help and support I ever needed.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Graphs are mathematical structures used to model pairwise relations between objects from a certain collection, which are one of the prime objects of study in discrete mathematics. Many problems of practical interest can be represented by graphs. Graph theory is the study of graphs and it has several open problems. One of these problems is the subgraph isomorphism problem from which one can extract a special case of it: the Graph Isomorphism problem (GI), which is of both theoretical an practical interest. GI tests whether there is a one-to-one mapping between the vertices of two graphs that preserves the arcs. This problem has applications in many fields, like pattern recognition and computer vision [3], data mining [32], VLSI layout validation [1], and chemistry [6, 30]. At the theoretical level, its main theoretical interest is that it is not known whether GI is in P or whether it is NP-complete.

1.2 State of the Art

For the last three decades, *nauty* [17, 18] has been the most widely used tool for graph isomorphism testing and canonical labeling. However, Miyazaki proved [19] that nauty required exponential time for a family of colored graphs. McKay noted also that nauty would also require exponential time for unions of strongly regular graphs. All this has encouraged researchers to develop new tools, to try to overcome this drawback. Some of them are based, like nauty, on canonical labeling. Examples are *bliss* [10, 9], *Traces* [21], and *nishe* [29, 28]. Another tool, named *saucy* [5, 11] solves a related problem: computing the automorphism group of a graph. It is specially designed to efficiently process big sparse graphs.

A different way to tackle the GI problem was suggested in [15]. The tool developed in that work, called *conauto*, does not generate a canonical labeling of the graphs being tested, but instead looks for similar sequences of vertex partitions. To do so, it uses a limited search for automorphisms on the graphs. This algorithm has good performance in practice for different families of graphs, like for example the graphs of Miyazaki. However, the inability of conauto to compute the whole automorphism group restricts the benefit obtained from known automorphisms. That reduced its performance with some families of graphs, like Latin square graphs or the point-line graphs of Desarguesian projective planes. One major contribution of the followup work [22] was a way to avoid backtracking when processing union graphs, which could be applied in a more general field. Some ways to improve algorithm conauto were also suggested as open problems; in particular, computing the full automorphism group of the graphs, and recording and using information on non-isomorphisms to prune the search space.

1.3 Objectives

The aim of this dissertation is to improve the *conauto* algorithm performing a complete automorphism group computation, preserving its approach. The automorphism group computation will consist in obtaining a set of generators, the size of the automorphism group itself, and the orbit partition of the set of vertices. This task must be carried out during the search for automorphisms, so that it does not affect the basic structure of the algorithm. In order to reduce the time required for the automorphism group computation, techniques that reduce the search space are needed. For example, the generation of the initial sequence of partitions will be revised for changes, and the matching process will also be modified to make profit of the information obtained during the search for automorphisms in both of the graphs being tested. A new pivot cell selection algorithm is designed, which is quite time-consuming. However, it is worth, since it will reduce the time needed for the search for automorphisms, which would be otherwise much more time-consuming. With all this in mind, the resulting algorithm must be general purpose, complete (always giving an answer, yes or no), and must preserve the advantages of the main approaches (canonical labeling, and direct backtrack algorithms) avoiding their disadvantages, such as *conauto* did.

To test our algorithm, and as an additional contribution, we will construct a benchmark for testing GI practical algorithms. Thus, we will build a graph database with different families of graphs which, we believe, are somewhat relevant for the tests. Some of them are graphs that are handled easily by most graph isomorphism algorithms, like random graphs. Other families are known to be especially hard for nauty, though they are conceptually simple. Finally, there is a family of graphs (the point-line graphs of Desarguesian Projective Planes) for which the pivot cell selection policy used is determinant on the performance of every GI algorithm we know of. For some families, we will perform only positive tests, i.e., in which both graphs are isomorphic (if negative tests do not apply or are not relevant). In our benchmark, both directed and undirected graphs are included, since nauty suffers from a special difficulty to deal with digraphs.

We will compare our algorithm with other algorithms by means of our benchmark. We include graph charts to show the practical performance of our algorithm in comparison with the other algorithms. For this purpose, we will use nauty, bliss, Traces, and saucy as algorithms of reference. The choice of nauty is obvious, since it is the referent for practical graph isomorphism algorithms. Bliss and Traces have shown to have a better performance than nauty for some graph families. Finally, saucy is a tool specially tuned for big sparse graphs, but only computes the automorphism group of a graph. Hence, it can not be used for canonical labeling, nor isomorphism testing. However, it will be useful test our algorithm against saucy to evaluate the performance of our complete search for automorphisms.

Our algorithm, although not designed for graphs with colored vertices or colored arcs, can be extended in a naive way to handle them. Besides, it does not include sophisticated invariants that could be added at the users choice (like nauty does), but again, it would not be difficult to add that functionality.

1.4 Methodology

First of all, a detailed study of the Ph.D thesis [22], where *conauto* algorithm is described, was carried out in order to understand and analyze the concepts and the theoretical background used by the algorithm. This dissertation studies GI and some graph families along with their complexity for GI. Once the problem is presented, the algorithm is explained for stages adding functionalities. Moreover, practical examples are shown to help for the understanding of the algorithm and its performance. Then, an study of the automorphism group of a graph, relying on [12], was done, since the automorphism group computation of a graph is the aim of this dissertation. Next, a review of the state of the art tools for GI was carried out in order to see their main features, performance and contributions.

Finally, once the theoretical study was finished, a modular design of the algorithm source code (the starting version was conauto-1.02) was done for having specific functional parts of code divided. Then, the implementation phase began.

1.5 Structure

The rest of this dissertation is organized as follows. In the next chapter, all the theoretical background is described, which will be used by the algorithm. Chapter 3 describes the algorithm and its functionalities as well as the use of the theoretical background previously explained. We also prove the correctness of the algorithm. In Chapter 4, we give a description of the families of graphs chosen for the tests, and justify why we believe they are appropriate to evaluate GI algorithms. Moreover, we show an example of automorphism group computation and graph isomorphism testing, and perform some tests with different families of graphs, showing the results obtained with the algorithm. Finally, in Chapter 5 we summarize the conclusions of the work described here, and how this work might be extended in the future.

Chapter 2

Theoretical Background

In this chapter, we introduce some notation and definitions that will be used throughout this work. First we recall some basic concepts of graph theory, and redefine others in a way that bests serves our purpose. Then, we introduce some specific definitions that are to be used in the development of our algorithms.

2.1 Basic Definitions

A *directed graph* $G = (V, R)$ consists of a finite non-empty set V of vertices and a binary relation R, i.e. a subset $R \subseteq V \times V$. The elements of R are called *arcs*. An arc $(u, v) \in R$ is considered to be oriented from u to v. An *undirected graph* is a graph whose arc set R is symmetrical, i.e. $(u, v) \in R$ iff $(v, u) \in R$. From now on, we will use the term *graph* to refer to a *directed graph*. Undirected graphs are just a particular case of directed graphs.

Given a graph $G = (V, R)$, R can be represented by an *adjacency matrix* $Adj(G) = A$ with size $|V| \times |V|$ in the following way:

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

Note the difference with the traditional definition of the adjacency matrix where $A_{uv} = 1$ if $(u, v) \in R$ and $A_{uv} = 0$ if $(u, v) \notin R$. Our definition gives, in one matrix element A_{uv}, the information in two elements of the traditional adjacency matrix (elements A_{uv} and A_{vu}). Furthermore, it can be easily generalized to colored arcs (each type or color of an arc may be denoted by a different value in the adjacency matrix).

Definition 2.1 *Given a graph $G = (V, R)$ and its adjacency matrix $Adj(G) = A$. Let $V_1 \subseteq V$, the* available degree *of v in V_1 under G, denoted by $ADeg(v, V_1, G)$, is the 3-tuple (D_3, D_2, D_1) where $D_i = |\{u \in V_1 : A_{vu} = i\}|$ for $i \in \{1, 2, 3\}$.*

Definition 2.2 *Given a graph $G = (V, R)$ and its adjacency matrix $Adj(G) = A$, the* degree *of a vertex $v \in V$ under graph G, denoted by $Deg(v, G)$, is the 3-tuple (D_3, D_2, D_1) where $D_i = |\{u \in V : A_{vu} = i\}|$, for $i \in \{1, 2, 3\}$.*

Observe that $Deg(v, G) = ADeg(v, V, G)$. Again, note the difference with the traditional definition of degree. Our degree is a combination of the in-degree, the out-degree, and the number of neighbors of a vertex. (Observe also that if we colored the arcs with k colors, the degree would be a k-tuple.)

Extending the notation, we use $ADeg(V_1, V_2, G) = d$ for some $V_1, V_2 \subseteq V$ to denote that $\forall u, v \in V_1, ADeg(u, V_2, G) = ADeg(v, V_2, G) = d$. The same extension can be applied to the degree. Let $V_1 \subseteq V$ such that $\forall u, v \in V_1, Deg(u, G) = Deg(v, G) = d$. Then, we denote $Deg(V_1, G) = d$.

Let $G = (V, R)$ be a graph with $Adj(G) = A$, and $V_1, V_2 \subseteq V$. Let $ADeg(V_1, V_2, G) = (D_3, D_2, D_1)$, then we define the function $NumLinks(V_1, V_2, G) = D_3 + D_2 + D_1$ (i.e. the number of neighbors *each vertex* of V_1 has in V_2 under graph G), and the predicate $HasLinks(V_1, V_2, G) = (NumLinks(V_1, V_2, G) > 0)$.

We will say a 3-tuple $(D_3, D_2, D_1) \prec (E_3, E_2, E_1)$ when the first one precedes the second one in lexicographic order and $(D_3, D_2, D_1) \succ (E_3, E_2, E_1)$ when the second one precedes the first one in lexicographic order. This notation will be used to order the degrees and the available degrees of both vertices and sets.

Definition 2.3 *Let $G = (V, R)$ be a graph. Let $W \subseteq V$. Then the* subgraph induced *by W on G, denoted G_W, is the graph $H = (W, R')$ such that $R' = \{(u, v) : u, v \in W \wedge (u, v) \in R\}$.*

Definition 2.4 *Let $G = (V, R)$ be a graph. A* permutation *$\pi : V \longrightarrow V$ acting on the finite set V is a one-to-one mapping from V onto itself. The image of an element $v \in V$ with respect to the permutation π is denoted by v^π. A vertex $v \in V$ is* fixed *by π if $v^\pi = v$ and it is* permuted *if it is not fixed by π. G^π denotes the graph in which vertices x^π and y^π are adjacent if and only if x and y are adjacent in G. Let $W \subseteq V$, then $W^\pi = \{w^\pi : w \in W\}$. Let $\mathcal{S} = (S_1, ..., S_r)$ be a partition of V, then $\mathcal{S}^\pi = (S_1^\pi, ..., S_r^\pi)$.*

Definition 2.5 *Let $G = (V, R_G)$ and $H = (V, R_H)$ be two graphs with the same vertex set. A permutation π of V is called an* isomorphism *of G and H if $\forall u, v \in V$, we have that $(v, u) \in R_G \iff (v^\pi, u^\pi) \in R_H$ (i.e., the adjacency matrix is preserved).*

G and H are called *isomorphic*, written $G \simeq H$, if there is at least one isomorphism π of them. An *automorphism* of G is an isomorphism of G and itself, i.e., $G^\pi = G$.

Definition 2.6 *Let $G = (V, R)$ be a graph. The* automorphism group *of G is $Aut(G) = \{\pi : G^\pi = G\}$ (i.e., the set of all automorphisms of G). Let $\mathcal{S} = (S_1, ..., S_r)$ be a partition of V. Analogously, $Aut(G, \mathcal{S}) = \{\pi : \pi \in Aut(G)$ and $\forall i \in \{1, ..., r\}, \forall v \in S_i, v^\pi \in S_i\}$.*

Observe that $Aut(G, \mathcal{S}) \subseteq Aut(G)$. When $S = \{V\}$ (i.e., $|S| = 1$), $Aut(G, \mathcal{S}) = Aut(G)$.

Definition 2.7 *Let $\mathcal{G}$ be a family of graphs. An* invariant *on $\mathcal{G}$ is a function Φ with domain $\mathcal{G}$ such that: $\Phi(G_1) = \Phi(G_2)$ if G_1 is isomorphic to G_2. A complete graph invariant is one such that $\Phi(G_1) = \Phi(G_2)$ if and only if G_1 is isomorphic to G_2.*

The number of vertices, and the number of arcs are some examples of graph invariants which are easy to compute. However, none of these graph invariants is a complete graph invariant, and no known complete graph invariant is computable in polynomial time.

Definition 2.8 *Let $\mathcal{G}$ be a family of graphs on vertex set V and $\mathbb{S}$ the set of partitions of V. A* vertex-invariant *is a function Φ with domain $\mathcal{G} \times \mathbb{S} \times V$ such that $\Phi(G^\pi, \mathcal{S}^\pi, v^\pi) = \Phi(G, \mathcal{S}, v)$ for every graph $G \in \mathcal{G}$, partition $\mathcal{S} \in \mathbb{S}$, vertex $v \in V$ and any permutation π of V.*

Informally, previous definition says that the values of Φ are independent of the labeling of the graph. Observe that G and G^π are isomorphic graphs, and if $G = G^\pi$ then $\pi \in Aut(G)$, i.e., π is an automorphism of G.

Observation 2.1 Available degree *and* degree *are vertex-invariants.*

Proof: Let $G = (V, R)$ be a graph, $A = Adj(G)$ the adjacency matrix of G, π a permutation of V, and $W \subseteq V$. From the definition of G^π it is easy to see that $A_{uv} = A_{u^\pi v^\pi}$ for all $u, v \in V$. Thus, we obtain that $ADeg(v, W, G) = ADeg(v^\pi, W^\pi, G^\pi)$ and $Deg(v, G) = Deg(v^\pi, G^\pi)$ from their respective definitions. ■

2.2 Sequences of Partitions

In this section we will define what a partition is, the procedures that will be used to refine partitions, and the basic structures we use for isomorphism testing: the sequences of partitions.

Definition 2.9 *A* partition *of a set V is a sequence $\mathcal{S} = (S_1, ..., S_r)$ of disjoint nonempty subsets of V such that $V = \bigcup_{i=1}^{r} S_i$. The sets S_i are called the* cells *of $\mathcal{S}$. The empty partition will be denoted by $\emptyset$.*

The *unit partition* is the partition with only one cell. The *discrete partition* is that in which every cell is a singleton.

Definition 2.10 *Let $\mathcal{S} = (S_1, ..., S_r)$ and $\mathcal{T} = (T_1, ..., T_s)$ be partitions of two disjoint sets S and T, respectively. 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, $\emptyset \circ \mathcal{S} = \mathcal{S} = \mathcal{S} \circ \emptyset$.*

Usually, the partition of the vertex set according to the degree of each vertex is used as the starting point for vertex classification in graph isomorphism testing algorithms. It is easy to see that it is necessary for two graphs to be isomorphic, that the number of vertices of each degree is the same in both graphs (recall that the degree is a vertex-invariant of a graph). Let us formally define the degree partition of a graph.

Definition 2.11 *Let $G = (V, R)$ be a graph. The* degree partition *of G, which will be denoted as $DegreePartition(G)$, is a partition $\mathcal{S} = (S_1, ..., S_r)$ of V such that for all $i, j \in \{1, ..., r\}$, $i < j$ implies $Deg(V_i, G) \succ Deg(V_j, G)$.*

Partitions may be further refined by different invariants, but we only consider two applications of the available degree invariant. The first one is to classify the vertices in the cells of a partition considering the adjacency type they have with a certain *pivot vertex* in the graph considered. This way, cells may be split into up to four distinct cells (or k if we use k-colored arcs). We call this process a vertex refinement. The second refinement classifies the vertices in the cells using their available degree in a given *pivot set* (cell). This leads to what we call a set refinement. In fact, vertex refinement is a special case of set refinement, which occurs when the pivot set is singleton. We distinguish them for implementation reasons.

Definition 2.12 *Let $G = (V, R)$ be a graph, $v \in V$, $W \subseteq V \setminus \{v\}$. The* vertex partition *of W by v, denoted $PartitionByVertex(W, v, G)$, is a partition $(S_1, ..., S_r)$ of W such that for all $i, j \in \{1, ..., r\}$, $i < j$ implies $ADeg(S_i, \{v\}, G) \succ ADeg(S_j, \{v\}, G)$.*

Definition 2.13 *Let $G = (V, R)$ be a graph, and $\mathcal{S} = (S_1, ..., S_r)$ a partition of V. Let $v \in S_x$ for some $x \in \{1, ..., r\}$. The* vertex refinement *of $\mathcal{S}$ by v, denoted $VertexRefinement(\mathcal{S}, v, G)$ is the partition $\mathcal{T} = \mathcal{T}_1 \circ ... \circ \mathcal{T}_r$ such that for all $i \in \{1, ..., r\}$, $\mathcal{T}_i$ is the empty partition $\emptyset$ if $\neg HasLinks(S_i, V, G)$ and $PartitionByVertex(S_i \setminus \{v\}, v, G)$ otherwise. S_x is the* pivot set *and v is the* pivot vertex.

Definition 2.14 *Let $G = (V, R)$ be a graph, and $V_1, V_2 \subseteq V$. The* set partition *of V_1 by V_2, denoted $PartitionBySet(V_1, V_2, G)$, is a partition $(S_1, ..., S_r)$ of V_1 such that for all $i, j \in \{1, ..., r\}$, $i < j$ implies $ADeg(S_i, V_2, G) \succ ADeg(S_j, V_2, G)$.*

Definition 2.15 *Let $G = (V, R)$ be a graph, and $\mathcal{S} = (S_1, ..., S_r)$ a partition of V. Let $P = S_x$ for some $x \in \{1, ..., r\}$ be a given pivot set. The* set refinement *of $\mathcal{S}$ by P, denoted $SetRefinement(\mathcal{S}, P, G)$ is the partition $\mathcal{T} = \mathcal{T}_1 \circ ... \circ \mathcal{T}_r$ such that for all $i \in \{1, ..., r\}$, $\mathcal{T}_i$ is the empty partition $\emptyset$ if $\neg HasLinks(S_i, V, G)$ and $PartitionBySet(S_i, P, G)$ otherwise.*

Once we have presented the possible partition refinements that may be applied to partitions, we can build sequences of partitions, following the *individualization-refinement* technique, in which an initial partition of a graph is taken (for example the degree partition) and subsequent partitions are generated, each from its previous one, by applying one of the refinements defined above. Vertex refinements are tagged as VERTEX (if the pivot set has only one vertex), SET (if a set refinement is possible with some pivot set), or BACKTRACK otherwise (when a vertex refinement is performed with a pivot set with more than one vertex), what is called a *vertex individualization.*

Definition 2.16 *Let $\mathcal{G}$ be a family of graphs on vertex set V and $\mathbb{S}$ the set of partitions of V. A* pivot cell selector *is a function σ with domain $\mathcal{G} \times \mathbb{S}$ which returns a cell, such that $\sigma(G, \mathcal{S})^\pi = \sigma(G^\pi, \mathcal{S}^\pi)$ for every graph $G \in \mathcal{G}$, partition $\mathcal{S} \in \mathbb{S}$, and any permutation π of V.*

Definition 2.17 *Let $G = (V, R)$ be a graph. A* sequence of partitions *for graph G is a tuple $(\mathsf{S}, \mathsf{R}, \mathsf{P})$, where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, are the partitions themselves, $\mathsf{R} = (R^0, ..., R^{t-1})$ indicate the type of refinement applied at each step, and $\mathsf{P} = (P^0, ..., P^{t-1})$ is the pivot set used for each refinement step, such that all the following holds. Let $\mathcal{S}^i = (S^i_1, ..., S^i_{r_i})$, $V^i = \bigcup_{j=1}^{r_i} S^i_j$, for all $i \in \{0, ..., t\}$. Then:*

1. *$R^i \in \{\text{VERTEX}, \text{SET}, \text{BACKTRACK}\}$, $P^i \in \{1, ..., r_i\}$ and $HasLinks(S^i_{Pi}, V^i, G)$.*
2. *$R^i = \text{SET}$ implies that $\mathcal{S}^i$ is not equitable and $\mathcal{S}^{i+1} = SetRefinement(\mathcal{S}^i, S^i_{Pi}, G)$.*
3. *$R^i = \text{VERTEX}$ implies that $S^i_{Pi} = \{v\}$ is a singleton and $\mathcal{S}^{i+1} = VertexRefinement(\mathcal{S}^i, v, G)$.*
4. *$R^i = \text{BACKTRACK}$ implies that $\mathcal{S}^i$ is equitable, S^i_{Pi} is not a singleton, and, for some $v \in S^i_{Pi}$, $\mathcal{S}^{i+1} = VertexRefinement(\mathcal{S}^i, v, G)$.*

If a sequence of partitions Q satisfies that, for all $x \in \{1, ..., r_t\}$, either $\neg HasLinks(S^t_x, V^t, G)$, or S^t_x is a singleton, then we say that Q is a *complete* sequence of partitions.

For convenience, for all $l \in \{1, ..., t-1\}$, by *level l* we refer to the tuple $(\mathcal{S}^l, R^l, P^l)$ in a sequence of partitions. Level t is identified by $\mathcal{S}^t$, since R^t and P^t are not defined.

Now we will show how to derive a permutation on the vertex set of a graph G, from a sequence of partitions for that graph. To do so, we define the order induced by a sequence of partitions on the vertices of a graph. Once we have such an order, it is easy to derive a permutation on the vertex set of the graph (a relabeling of the vertex set).

Definition 2.18 *Let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions for graph $G = (V, R)$ where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, and $\mathsf{P} = (P^0, ..., P^{t-1})$. For all $i \in \{0, ..., t\}$, let $\mathcal{S}^i =$*

$(S^i_1, ..., S^i_{r_i})$*, and* $V^i = \bigcup_{j=1}^{r_i} S^i_j$*. The* order induced *by* Q *on the vertex set* $(V \setminus V^t) \cup \{v \in V^t : \neg \mathit{HasLinks}(v, V^t, G)\})$*, denoted by* $<_{\mathsf{Q}}$*, is that which satisfies that:*

1. *For all* $i \in \{0, ..., t-1\}$*:*
   (a) *For all* $u \in V^i \setminus V^{i+1}, v \in V^{i+1}$*,* $u <_{\mathsf{Q}} v$*.*
   (b) $\mathcal{S}^{i+1} = \mathit{VertexRefinement}(\mathcal{S}^i, u, G)$ *implies that for all* $v \in V^i \setminus V^{i+1}$*,* $u <_{\mathsf{Q}} v$*.*
2. *For all* $i \in \{0, ..., t\}$*:*
   (a) *For all* $x, y \in \{1, ..., r_i\}$ *such that* $\neg \mathit{HasLinks}(S^i_x, V^i, G)$ *and* $\neg \mathit{HasLinks}(S^i_y, V^i, G)$*,* $x < y$ *implies that for all* $u \in S^i_x, v \in S^i_y$*,* $u <_{\mathsf{Q}} v$*.*
   (b) *For all* $x \in \{1, ..., r_i\}$ *such that* $\neg \mathit{HasLinks}(S^i_x, V^i, G)$ *and* $|S^i_x| > 1$*, for all* $u, v \in S^i_x$*,* $u <_{\mathsf{Q}} v$ *if and only if* u *precedes* v *in lexicographic order.*

If a sequence of partitions is complete, then it induces a complete order on the vertices of the whole vertex set V*, adding the following statement:*

3. *For all* $u, v \in V^t$ *such that* $\mathit{HasLinks}(u, V^t, G)$ *and* $\mathit{HasLinks}(v, V^t, G)$*, there are* $x, y \in \{1, ..., r_t\}$ *so that* $\{u\} = S^t_x$ *and* $\{v\} = S^t_y$*. Then,* $x < y$ *implies that* $u <_{\mathsf{Q}} v$*.*

The *labeling induced* by a complete sequence of partitions Q is a mapping $\lambda : \{1, ..., |V|\} \longrightarrow V$ such that, for all $i \in \{1, ..., |V|\}$ and for all $v \in V$, $\lambda(i) = v$ if and only if v is the i^{th} vertex w.r.t. the total order $<_{\mathsf{Q}}$ defined. If the sequence of partitions is not complete, then the labeling applies to the vertices in $(V \setminus V^t) \cup \{v \in V^t : \neg \mathit{HasLinks}(v, V^t, G)\}$. The *inverse mapping* $\lambda^{-1} : V \longrightarrow \{1, ..., |V|\}$ is such that $\lambda^{-1}(v) = i$ if and only if $\lambda(i) = v$. The function *InducedOrder*(Q) returns the labeling induced by Q.

Now, we will introduce the concept of compatibility between two sequences of partitions.

Definition 2.19 *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 sequences of partitions *if they satisfy all the following:*

1. $|\mathsf{S}_G| = |\mathsf{S}_H| = t, |\mathsf{R}_G| = |\mathsf{R}_H| = t-1, |\mathsf{P}_G| = |\mathsf{P}_H| = t-1$*.*
2. *Let* $\mathsf{R}_G = (R^0_G, ..., R^{t-1}_G)$*, and* $\mathsf{R}_H = (R^0_H, ..., R^{t-1}_H)$*, then for all* $i \in \{0, ..., t-1\}$*,* $R^i_G = R^i_H$*.*
3. *Let* $\mathsf{P}_G = (P^0_G, ..., P^{t-1}_G)$*, and* $\mathsf{P}_H = (P^0_H, ..., P^{t-1}_H)$*, then for all* $i \in \{0, ..., t-1\}, P^i_G = P^i_H$*.*
4. *Let* $\mathsf{S}_G = (\mathcal{S}^0, ..., \mathcal{S}^t)$*,* $\mathsf{S}_H = (\mathcal{T}^0, ..., \mathcal{T}^t)$*, then for all* $i \in \{0, ..., t\}$*,* $|\mathcal{S}^i| = |\mathcal{T}^i|$*.*
5. *For all* $i \in \{0, ..., t\}$*, let* $\mathcal{S}^i = (S^i_1, ..., S^i_{r_i})$*,* $\mathcal{T}^i = (T^i_1, ..., T^i_{r_i})$*. Then, if* $\mathcal{S}^i$ *is equitable, for all* $x, y \in \{1, ..., r_i\}$*,* $\mathit{ADeg}(S^i_x, S^i_y, G) = \mathit{ADeg}(T^i_x, T^i_y, H)$*. Otherwise, let* $V^i = \bigcup_{j=1}^{r_i} S^i_j$*,* $W^i = \bigcup_{j=1}^{r_i} T^i_j$*, then for all* $x \in \{1, ..., r_i\}$*,* $\mathit{ADeg}(S^i_x, V^i_y, G) = \mathit{ADeg}(T^i_x, W^i, H)$*.*

Once we have defined the compatibility between sequences of partitions, we can state the relationship between compatibility of sequences of partitions and graph isomorphism in the following theorem, whose proof follows from the compatibility of sequences of partitions.

Theorem 2.1 *Let* $G = (V_G, R_G)$ *and* $H = (V_H, R_H)$ *be two graphs. Let* Q_G *and* Q_H *be two sequences of partitions, for graphs* G *and* H *respectively, which are compatible. Let* $W_G = (V_G \setminus V^t_G) \cup \{v \in V^t_G : \neg \mathit{HasLinks}(v, V^t_G, G)\}$ *and* $W_H = (V_H \setminus V^t_H) \cup \{v \in V^t_H : \neg \mathit{HasLinks}(v, V^t_H, H)\}$*. Let* λ *and* λ' *be the labelings induced by* Q_G *and* Q_H *on the set of vertices* W_G *and* W_H *respectively. Then, the mapping* $m : W_G \longrightarrow W_H$ *such that* $m(\lambda(i)) = \lambda'(i)$*,* $i \in \{1, \ldots, |W_G|\}$*, is an isomorphism of* G_{W_G} *and* H_{W_H}*. If the sequences of partitions* Q_G *and* Q_H *are complete, the mappging* m *is the isomorphism of* G *and* H *induced by the sequences of partitions* Q_G *and* Q_H*.*

Corollary 2.1 *Two graphs* G *and* H *are isomorphic if and only if there are two complete sequences of partitions* Q_G *and* Q_H*, for graphs* G *and* H *respectively, which are compatible.*

Corollary 2.2 *Let $\mathsf{Q}_G = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ and $\mathsf{Q}_H = (\mathsf{T}, \mathsf{R}, \mathsf{P})$ two compatible sequences of partitions. Let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$ and $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$. Let $\mathcal{S}^i = \{S^i_1, ..., S^i_{r_i}\}$ and $\mathcal{T}^i = \{T^i_1, ..., T^i_{r_i}\}$ for all $i \in \{0, ..., t\}$. Then, for each $x \in \{1, \ldots, |V|\}$, $\lambda(x) \in S^j_k$ for some $j \in \{0, ..., t\}$ and some $k \in \{1, ..., r_j\}$ if and only if $\lambda'(x) \in T^j_k$.*

One parameter of a sequence of partitions that will be used by our algorithms to choose the target partition to be reproduced (as it will be shown in next section), is the number of refinement steps where backtracking will be needed.

Definition 2.20 *Let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions, and let $\mathsf{R} = (R^0, ..., R^{t-1})$. The* amount of backtracking *induced by Q is BacktrackAmount$(\mathsf{Q}) = |\{i : i \in \{1, ..., t-1\} \wedge R^i =$ BACKTRACK$\}|$.*

In a sequence of partitions, a backtrack level arises when a partition does not have singleton cells (suitable for a vertex refinement) and it is not possible to refine such partition by means of a set refinement. Let us introduce now the concept of *equitable partition*, which will be useful in the following discussion.

Definition 2.21 *Let $G = (V, R)$ be a graph, and let $\mathcal{S} = (S_1, ..., S_r)$ be a partition of V. $\mathcal{S}$ is said to be* equitable *(with respect to G) if for all $i \in \{1, ..., r\}$, for all $u, v \in S_i$, for all $j \in \{1, ..., r\}$, $ADeg(u, S_j, G) = ADeg(v, S_j, G)$.*

Observation 2.2 *The partition at a backtrack level is equitable.*

Proof: Assume otherwise. Then, there exists some S_j such that there are two vertices u, v in some S_i, such that $ADeg(u, S_j, G) \neq ADeg(v, S_j, G)$. Therefore, it would be possible to perform a set refinement on the partition, using S_j as the pivot cell, and vertices u and v would be distinguished by this refinement, and cell S_i would be split. This is not possible since, at a backtrack level, no set refinement has succeeded. ■

Observation 2.3 *Let l be a backtracking level. Let $\mathcal{S}^l = (S^l_1, ..., S^l_r)$ be the partition at that level. Then, for all $i \in \{1, ..., r\}$, $G_{S^l_i}$ is regular.*

Proof: From Observation 2.2, $\mathcal{S}^l$ is equitable. Fix $i \in \{1, ..., r\}$, then, from Definition 2.21, for all $u, v \in S^l_i$, $ADeg(u, S^l_i, G) = ADeg(v, S^l_i, G)$. Therefore, $G_{S^l_i}$ is regular, for all $i \in \{1, ..., r\}$. ■

2.3 Automorphism Group

When we look for automorphisms of a graph, we actually find equivalences among vertices. The notion of *equivalence* among vertices in a graph is essential for automorphism group management. Instead of using the Schreier-Sims representation of the automorphism group [25], only a set of (at most $n - 1$) generators is stored (for a graph on n vertices). The equivalence between vertices will be used to eliminate backtrack levels (if all the vertices of the pivot cell at some level are found to be equivalent, then the backtrack point is removed) , to prune the search for automorphisms, and to prune the search for a sequence of partitions compatible with the target. When two vertices are equivalent, they are said to belong to the same *orbit*. The set of all its orbits defines a partition of the vertices of a graph. When the whole automorphism group of a graph has been computed (the set of generators in our case), all the vertices are correctly classified into their respective orbits, and the resulting partition is called the *orbit partition.* If a graph is vertex transitive, then all its vertices belong to the same orbit, so the orbit partition has only one orbit. The orbit partition of a graph G could be obtained from its automorphism group $Aut(G)$, when the automorphism group acts on the set of vertices V.

Definition 2.22 *Let $G = (V, R)$ be a graph, and let $u, v \in V$. Vertices u and v are* equivalent *if there is an automorphism π of G that permutes u and v, i.e. $u^\pi = v$.*

Definition 2.23 *Let $G = (V, R)$ be a graph. The* generating set *of the automorphism group $Aut(G)$ of G is a set of generators $\Gamma = \{\gamma_1, ..., \gamma_g\}$, such that $\Gamma \subseteq Aut(G)$ and $Aut(G) = <\Gamma>$ (i.e., Γ generates $Aut(G)$).*

Observation 2.4 *Let $\pi_1, \pi_2 \in Aut(G)$ be two permutations of V. Then, $G^{\pi_1} = G^{\pi_2}$ (i.e., both graphs are automorphic), and the autmorphism itself is the mapping $m(v^{\pi_1}) = v^{\pi_2}$.*

Definition 2.24 *Let $G = (V, R)$ be a graph, $Aut(G)$ the automorphism group of the graph G. The* orbit *of a vertex $v \in V$ is defined as the set of all images $v^{Aut(G)} = \{v^\pi : \pi \in Aut(G)\}$.*

Let $\Gamma = \{\gamma_1, ..., \gamma_g\}$ a generating set, such that $<\Gamma> = Aut(G)$. Thus, we define the function $OrbitOf(v, \Gamma) = \{v^\pi : \pi \in <\Gamma>\}$, for any $v \in V$.

Definition 2.25 *Let $G = (V, R)$ be a graph. The* orbit partition *of G is a partition $\mathsf{O} = \{O_1, ..., O_n\}$ of V, such that for all $i \in \{1, ..., n\}$, $v, u \in O_i$ implies that v and u are equivalent.*

Observation 2.5 *Observe that, when have not the whole generating set yet, $<\Gamma> \subset Aut(G)$, the function $OrbitOf(v, \Gamma)$ returns the orbit of vertex v under graph G_W, where $W \subseteq V$ is the set of those vertices that are not fixed in any permutation $\gamma \in \Gamma$; the rest of vertices will be in a singleton orbit. Both cases, the function $Orbits(\Gamma)$ returns the orbit partition, as the concatenation of each different orbit of any vertex $v \in V$.*

Now, we will show some ways to infer vertex equivalence during the generation of a sequence of partitions for a graph, and how the compatibility of sequences of partitions implies equivalence of vertices.

Lemma 2.1 *Let $\mathsf{Q}_G = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions for graph G, such that $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, and $\mathsf{P} = (P^0, ..., P^{t-1})$. Let $l \in \{1, ..., t\}$, $\mathcal{S}^l = (S^l_1, ..., S^l_{r_l})$, and $V^l = \bigcup_{i=1}^{r_l} S^l_i$, such that there is some $k \in \{1, ..., r_l\}$ with $NumLinks(S^l_k, V^l, G) = 0$ and $|S^l_k| > 1$. Then, for all $u, v \in S^l_k$, u and v are equivalent.*

Proof: If u and v belong to the same cell S^l_k, none of the vertices previously discarded in the sequence of partitions has been able to distinguish them. Hence, their adjacencies are the same with all the previously discarded vertices. Besides, since they have no remaining links, they are not adjacent to any vertex in V^l, and they are discarded at this stage in the refinement process. Therefore, permuting u and v and fixing all the other vertices of graph G, we obtain an automorphism of G. Hence, u and v are equivalent. ■

This way, some equivalences may be detected using only one sequence of partitions. However, most equivalences are detected using two sequences of partitions. From Corollary 2.1 and the definition of automorphism, it follows that two compatible sequences of partitions for a graph G define an automorphism of G.

During the generation of a sequence of partitions for a graph G, backtracking points may arise. Let $\mathsf{Q}_G = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions for graph G. Let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$. Let us assume that R^l = BACKTRACK for some $l \in \{0, ..., t-1\}$. Then, let $\mathcal{S}^l = (S^l_1, ..., S^l_{r_l})$ and let $u \in S^l_{Pl}$ be the pivot vertex used for the vertex refinement at stage l. Let $v \in S^l_{Pl}, u \neq v$, be another vertex in the pivot set. Let Q'_G be a sequence of partitions compatible with Q_G, generated using vertex v instead of vertex u, at stage l. Note that Q_G and Q'_G are equal up to level l. Let $\leq_{\mathsf{Q}_G}$ be the order induced by Q_G on the vertices of V, and let $\leq_{\mathsf{Q}'_G}$ be the order induced by Q'_G on the same set of vertices V. Let $\omega_{\mathsf{Q}_G}(i)$ denote the i^{th} vertex with respect to $\leq_{\mathsf{Q}_G}$, and $\omega_{\mathsf{Q}'_G}(i)$ denote the i^{th} vertex with respect to $\leq_{\mathsf{Q}'_G}$. Then, mapping m, defined as $m(\omega_{\mathsf{Q}_G}(i)) = \omega_{\mathsf{Q}'_G}(i)$ for all $i \in \{1, ..., |V|\}$, is an automorphism of G. Mapping m satisfies that $u = \omega_{\mathsf{Q}_G}(k), v = \omega_{\mathsf{Q}'_G}(k)$, for some $k \in \{1, ..., |V|\}$. Then:

Lemma 2.2 *For all $j \in \{k, ..., |V|\}$, $\omega_{\mathsf{Q}_G}(j)$ and $\omega_{\mathsf{Q}'_G}(j)$ are equivalent, and m fixes vertices $\omega_{\mathsf{Q}_G}(1), ..., \omega_{\mathsf{Q}_G}(k-1)$, i.e. they are pairwise equal to $\omega_{\mathsf{Q}'_G}(1), ..., \omega_{\mathsf{Q}'_G}(k-1)$.*

While the equivalence stated in Lemma 2.1 is somewhat universal, i.e., the automorphism discovered fixes the rest of the vertices in the graph, the equivalences stated in Lemma 2.2 rely on the fact that only the vertices previously discarded are fixed by the automorphism discovered, and it is not known if fixing other vertices the equivalence still holds. Nevertheless, already found vertex equivalences may be used to prune future searches provided that backtracking points are explored in a certain order.

Lemma 2.3 *If two vertices u and v are equivalent at level l, then they are equivalent at any level $i \in \{0, ..., l-1\}$.*

Proof: Let u and v be two vertices that are equivalent at level l. This means that the two compatible sequences of partitions that determined their equivalence share the first l levels. Therefore, from Lemma 2.2, the vertices in $V \setminus V^l$ are fixed by the automorphism induced by these sequences of partitions. Let us call this automorphism m. Since $V^l \subseteq V^i$ for all $i \in \{0, ..., l-1\}$, $(V \setminus V^i) \subseteq (V \setminus V^l)$. Hence, there is an automorphism that maps u and v, and fixes the vertices in $V \setminus V^i$, e.g. m. ■

Remark 2.1 *Let u and v be two vertices that are equivalent at level l. If u is equivalent to p at level l, then v and p are also equivalent at level l, and if u is not equivalent to p at level l, then v and p are not equivalent at level l.*

Proof: If u and v are equivalent at level l, that is because there is an automorphism m that permutes u and v, and fixes all the vertices in $V \setminus V^l$. If u and p are equivalent at level l, then there is an automorphism m' that permutes u and p, and fixes all the vertices in $V \setminus V^l$. Since $m'(u) = p$, $m'(m(v)) = p$. Hence, the composition of m and m' yields an automorphism of v and p that fixes all the vertices in $V \setminus V^l$, since both automorphisms fixed them.

By the reverse argument, if there is no automorphism that fixes the vertices in $V \setminus V^l$ and permutes u and p, then one can conclude that there is no automorphism that fixes the vertices in $V \setminus V^l$ and permutes v and p. Otherwise, if there were such an automorphism m' such that $m'(p) = v$, then we could apply automorphism m, to get that $m(m'(p)) = m(v)$, i.e. $m(m'(p)) = u$, and all the vertices in $V \setminus V^l$ would be fixed, since they were fixed by both automorphisms. Thus we reach a contradiction. ■

Similarly, if at some level, v and u are equivalent and so are w and x, then, if v and w are equivalent at this same level, then u and x are also equivalent at this level. It is easy to see that a simple composition of automorphisms, as in the previous cases, yields this result. Thus, during the computation of semiorbit partitions, the basic operation performed on the semiorbit partitions is the merging of semiorbits. When two vertices u and v are found equivalent, their semiorbits are merged.

Definition 2.26 *Let $G = (V, R)$ be a graph, an let $\mathsf{O} = \{O_1, ..., O_n\}$ be a partition of V. Then, merge$(\mathsf{O}, O_i, O_j) = (\mathsf{O} \setminus \{O_i, O_j\}) \cup \{O_i \cup O_j\}$.*

Lemma 2.4 *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.*

Making use of this lemma in our algorithm, probably not all the backtracking points will be eliminated, but a significant improvement may be achieved for graphs with a symmetric structure. This search for equivalence among the vertices in the pivot cells will be performed just after the generation of the sequences of partitions, and before the search for the compatible sequence of partitions.

Equivalence among vertices in a graph may be used during the search for the compatible sequence of partitions for the graph, thus reducing the number of vertices to try at a backtracking point, what will also help pruning the search space. However, note that the only information we consider about automorphisms is our semiorbit partition. Hence, with an extended sequence of partitions, we know that two vertices are equivalent, but we do not know which vertices are fixed by an automorphism that permutes them. Nevertheless, we can state the following observation:

Observation 2.6 *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.*

Proof: In fact, if there are vertices in singleton semiorbits, that is because all known automorphisms fix them. ■

Storing the full automorphism group of a graph, or at least all the automorphisms discovered, would be much more powerful. Some ways to represent an automorphism group feasible for our purpose are exposed in [12, Chapter 6]. However, more space or more computing would be needed than in the proposed algorithm. Hence, we have chosen to manage vertex equivalence the easy way. A future improvement to our algorithm might be to add a more powerful way to manage automorphisms. If our algorithm is modified to compute the automorphism group of a graph, this would be a compulsory feature.

2.4 Subpartitions Theorems

In this section we will define the concept of subpartition in a sequence of partitions. Then, we will state two theorems that can be used to significantly improve the performance of any algorithm based on the individulization-refinement process that computes the automorphism group of a graph. These results are specially powerful when dealing with graphs built from regularly connected (uniformly joined) components, but are not restricted to this case, since they are not based on the recognition of components.

Definition 2.27 *Let $G = (V_G, R_G)$ be a graph. Let $\mathcal{S} = (S_1, ..., S_r)$ be an equitable partition of a set $V \subseteq V_G$, and $\mathcal{T} = (T_1, ..., T_s)$ be an equitable partition of $W \subseteq V$. $\mathcal{T}$ is a* subpartition *of $\mathcal{S}$ if and only if for all $i \in \{1, ..., r\}$, there are no $j, k \in \{1, ..., s\}, j \neq k$ such that $T_j \subseteq S_i$,*

$T_k \subseteq S_i$, $HasLinks(T_j, W, G)$ and $HasLinks(T_k, W, G)$. (I.e., each cell with links of $\mathcal{T}$ is included in a different cell of $\mathcal{S}$.)

Let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a sequence of partitions for graph $G = (V, R)$ where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, and $\mathsf{P} = (P^0, ..., P^{t-1})$. Let us consider the case in which there are two partitions $\mathcal{S}^k$ and $\mathcal{S}^l$ such that both are equitable, $0 \leq k < l \leq t$, $\mathcal{S}^l$ is a subpartition of $\mathcal{S}^k$, and there is no $k < i < l$ such that $\mathcal{S}^i$ is a subpartition of $\mathcal{S}^k$. Let $p \in S^k_{P^k}$ be the pivot vertex used for the vertex refinement at level k. Assume that taking a vertex $q \in S^k_{P^k}, q \neq p$ at level k as the pivot vertex, an alternative sequence of partitions $\mathsf{Q}' = ((\mathcal{S}^0, ..., \mathcal{S}^k, \mathcal{T}^{k+1}, ..., \mathcal{T}^l), (R^0, ..., R^{l-1}), (P^0, ..., P^{l-1}))$ is generated, which is compatible with the original subsequence $((\mathcal{S}^0, ..., \mathcal{S}^l), (R^0, ..., R^{l-1}), (P^0, ..., P^{l-1}))$.

Let $V^k = \bigcup_{j=1}^{r_k} S^k_j$. From partition $\mathcal{S}^k$, we derive a new partition $\hat{\mathcal{S}}^k$ which contains the cells $S^k_i \in \mathcal{S}^k$, such that $HasLinks(S^k_i, V^k, G)$, in the same order. Analogously, we derive partitions $\hat{\mathcal{S}}^l$ and $\hat{\mathcal{T}}^l$, from $\mathcal{S}^l$ and $\mathcal{T}^l$ respectively. Note that $|\hat{\mathcal{S}}^l| = |\hat{\mathcal{T}}^l|$ and $|\hat{\mathcal{S}}^l| \leq |\hat{\mathcal{S}}^k|$. To simplify the notation, let us assume, in the following, that $|\hat{\mathcal{S}}^k| = |\hat{\mathcal{S}}^l| = |\hat{\mathcal{T}}^l| = r$, in which case some cells of $\hat{\mathcal{S}}^l$ and $\hat{\mathcal{T}}^l$ might be empty.

Let $\hat{V}^k = \bigcup_{j=1}^{r} \hat{S}^k_j$, $\hat{V}^l = \bigcup_{j=1}^{r} \hat{S}^l_j$, and $\hat{W}^l = \bigcup_{j=1}^{r} \hat{T}^l_j$. For each $i \in \{1, ..., r\}$, let $E_i = \hat{S}^k_i \setminus \hat{V}^l$, $E'_i = \hat{S}^k_i \setminus \hat{W}^l$, $A_i = E_i \cap E'_i$, $B_i = E_i \setminus A_i$, $C_i = E'_i \setminus A_i$, and $D_i = \hat{S}^k_i \cap \hat{V}^l \cap \hat{W}^l$. Observe that $|E_i| = |E'_i|$, and hence $|B_i| = |C_i|$. Let $E = \hat{V}^k \setminus \hat{V}^l$, $E' = \hat{V}^k \setminus \hat{W}^l$, $A = E \cap E'$, $B = E \setminus A$, $C = E' \setminus A$, and $D = \hat{V}^l \cap \hat{W}^l$. Observe that $E = A \cup B$, $E' = A \cup C$, and $X = \bigcup_{i=1}^{r} X_i$, for all $X \in \{E, E', A, B, C, D\}$. Now we analyze the adjacencies among the vertices of these sets.

Lemma 2.5 *Let α be the labeling induced by Q, and β the labeling induced by Q'. For each $u \in V \setminus \hat{V}^l$, let $u = \alpha(i)$ and $u' = \beta(i)$. Then, for each $j \in \{1, ..., r\}$, there is some $\delta \in \{0, 1, 2, 3\}$ such that, for all $v, w \in \hat{S}^l_j$, $v', w' \in \hat{T}^l_j$, $M_{uv} = M_{uw} = M_{u'v'} = M_{u'w'} = \delta$.*

Proof: If for some $u \in V \setminus \hat{V}^l$, $M_{uv} \neq M_{uw}$ for $u, w \in \hat{S}^l_j$, that cell would have been split at some refinement up to level l. The same argument applies to the vertices $u' \in E'$ and $v', w' \in \hat{T}^l_j$. Besides, from the compatibility of Q and Q', $M_{uv} = M_{uw} = M_{u'v'} = M_{u'w'} = \delta$, for each $j \in \{1, ..., r\}$. ■

Observe that, from Lemma 2.5, $M_{ab} = M_{ad}$ for all $b \in B_j$ and $d \in D_j$. Also from Lemma 2.5, $M_{ac} = M_{ad}$ for all $c \in C_j$ and $d \in D_j$. Hence,

Corollary 2.3 *Let $a \in A$, for each $j \in \{1, ..., r\}$, for all $b \in B_j$, $c \in C_j$, $M_{ab} = M_{ac}$.*

Lemma 2.6 *For each $i, j \in \{1, ..., r\}$, there is some $\delta \in \{0, 1, 2, 3\}$ such that for all $u \in B_i$, $v \in C_i$, $w \in D_i$, $u' \in B_j$, $v' \in C_j$, and $w' \in D_j$, $M_{uv'} = M_{uw'} = M_{vu'} = M_{vw'} = M_{wu'} = M_{wv'} = \delta$.*

Definition 2.28 *Let α be the labeling induced by Q. Let β be the labeling induced by Q' on the vertices of $V \setminus \hat{V}^l$. Then, the labeling $\gamma : \{1, ..., |V|\} \longrightarrow V$ is defined as follows:*

$$\gamma(i) = \begin{cases} \beta(i) & \forall i,\ 1 \leq i \leq (|V| - |\hat{V}^l|) \\ f(i) & \forall i,\ (|V| - |\hat{V}^l|) < i \leq |V| \end{cases} \quad \textit{where} \quad f(i) = \begin{cases} \alpha(i) & \textit{if } \alpha(i) \notin E' \\ f(\beta^{-1}(\alpha(i))) & \textit{if } \alpha(i) \in E'. \end{cases}$$

From Corollary 2.2, it follows that $\alpha(i)$ and $\gamma(i)$ are in the same cell of $\mathcal{S}^k$ for $(|V| - |\hat{V}^k|) < i \leq |V|$.

Lemma 2.7 *Let α and γ be as defined in Definition 2.28. Then, the mapping $m : V \longrightarrow V$ defined as $m(\alpha(i)) = \gamma(i)$ is an automorphism of G_C and G_B.*

Proof: If for all $b = \alpha(i) \in B$, $\gamma(i) \in C$, then the inverse mapping defined for the vertices in C is an isomorphism of G_C and G_B. Consider now the case in which there is some $b = \alpha(i) \in B$ such that $a_1 = \gamma(i) \in A$. Let $\{a_1, ..., a_n\}$ such that $a_x \in A$ and $a_x = m(a_{(x-1)})$ for all $x \in \{2, ..., n\}$, and $m(a_n) = c \in C$. Then, from Lemma 2.5, for each $j \in \{1, ..., r\}$, for all $u \in B_j$, $v \in C_j$, $M_{bu} = M_{a_1 v}$. Besides, from Corollary 2.3, $M_{a_1 u} = M_{a_1 v}$. Assume that for all $x \in \{1, ..., n-1\}$, for each $j \in \{1, ..., r\}$, for all $u \in B_j$, $v \in C_j$, $M_{bu} = M_{a_x u} = M_{a_x v}$. Then, since $m(a_{(n-1)}) = a_n$, from Lemma 2.5, $M_{a_n v} = M_{a_{n-1} u}$ for all $u \in B_j$, $v \in C_j$. Hence, from Corollary 2.3, $M_{a_n v} = M_{a_{n-1} u} = M_{bu}$. Finally, since $m(a_n) = c$, using the same argument, $M_{cv} = M_{a_n u} = M_{bu}$, for all $u \in B_j$, $v \in C_j$. Since this argument applies to every vertex of B, we can conclude that m defines an isomorphism of G_C and G_B. ■

The following theorem allows to detect automorphisms without having to generate a complete sequence of partitions.

Theorem 2.2 (Early Automorphism Detection) *Let α and γ be as defined in Definition 2.28. Then, the mapping $m : V \longrightarrow V$ defined as $m(\alpha(i)) = \gamma(i)$ is an automorphism of G.*

Proof: Let $M = Adj(G)$. To prove that mapping m is an automorphism of G, it is enough to prove the following properties, since they include all the cases.

1. m defines an isomorphism of $G_{(V \setminus \hat{V}^l)}$ and $G_{(V \setminus \hat{W}^l)}$. This property follows directly from Theorem 2.1 and the fact that, for all $i \in \{1, ..., |(V \setminus \hat{V}^l)|\}$, $\gamma(i) = \beta(i)$.
2. m defines an automorphism of G_D: the trivial automorphism. It holds because, for all $\alpha(i) \in D$, m maps it to $\gamma(i) = \alpha(i)$, since $D \cap E' = \emptyset$.
3. m defines an isomorphism of G_C and G_B. Proven as Lemma 2.7.
4. For all $\alpha(i) \in (V \setminus \hat{V}^l), \alpha(j) \in \hat{V}^l$, $M_{\alpha(i)\alpha(j)} = M_{\gamma(i)\gamma(j)}$ and $M_{\alpha(j)\alpha(i)} = M_{\gamma(j)\gamma(i)}$. This follows from Lemma 2.5, since $\alpha(j)$ and $\gamma(j)$ are in the same cell at level k.
5. For all $\alpha(i) \in D, \alpha(j) \in C$, $M_{\alpha(i)\alpha(j)} = M_{\gamma(i)\gamma(j)}$ and $M_{\alpha(j)\alpha(i)} = M_{\gamma(j)\gamma(i)}$. This follows from Lemma 2.6, since $\gamma(j) \in B$, and $\alpha(j)$ and $\gamma(j)$ are in the same cell at level k. ■

The following theorem shows how to prune the search for compatible sequences of partitions.

Theorem 2.3 (Backjumping) *Let $G = (V_G, R_G)$ be a graph, and $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ be a complete sequence of partitions for graph G, where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$. Let $H = (V_H, R_H)$ be a graph, and $((\mathcal{T}^0, ..., \mathcal{T}^l), (R^0, ..., R^{l-1}), (P^0, ..., P^{l-1}))$ a sequence of partitions for graph H, such that $l \leq t$ and $\mathcal{T}^l$ is equitable, which is compatible with the subsequence of partitions $((\mathcal{S}^0, ..., \mathcal{S}^l)$, $(R^0, ..., R^{l-1})$, $(P^0, ..., P^{l-1}))$ for graph G. Assume that there is no complete sequence of partitions, for graph H, that starts with $((\mathcal{T}^0, ..., \mathcal{T}^l)$, $(R^0, ..., R^{l-1})$, $(P^0, ..., P^{l-1}))$ and is compatible with Q. Let $0 \leq k < l$ such that $\mathcal{T}^l$ is a subpartition of $\mathcal{T}^k$. Then, no complete sequence of partitions, for graph H, that starts with $((\mathcal{T}^0, ..., \mathcal{T}^k)$, $(R^0, ..., R^{k-1})$, $(P^0, ..., P^{k-1}))$ can be compatible with Q.*

Proof: From the previous theorem, any complete sequence of partitions for H that starts with $(\mathcal{T}^0, ..., \mathcal{T}^k)$ and is compatible with Q up to level l, is equivalent to some complete sequence that starts with $(\mathcal{T}^0, ..., \mathcal{T}^l)$. Since, as assumed, no complete sequence of partitions compatible with Q that starts with $(\mathcal{T}^0, ..., \mathcal{T}^l)$ could be found, the claim follows. ■

2.5 Failure Recording

This techinque is based on the vertex invariant of the non-automorphic paths of the automorphism search tree. That means that within an automorphic path of the search tree, from a backtrack level, the non-automorphic paths are the same for every automorphism of a graph. So that, if we record this non-automorphic paths of the search tree, we prune the search space if appears a non-automorphic path that would not be exist during the automorphisms search.

Definition 2.29 *Let $\mathcal{G}$ be a family of graphs on vertex set V and $\mathbb{S}$ the set of partitions of V. Let $G = (V, R)$ be a graph and $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$ a sequence of partitions for graph G, where $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, and $\mathsf{P} = (P^0, ..., P^{t-1})$. Then, a* fail *of a sequence of partitions Q on level $l \in \{0, ..., t-1\}$, such that $R^l = \text{BACKTRACK}$, is given by the (vertex invariant) function $\Phi : \mathcal{G} \times \mathbb{S} \times V \longrightarrow \mathbb{N}$.*

Observation 2.7 *Let $\mathsf{Q}' = (\mathsf{T}, \mathsf{R}, \mathsf{P})$ be a complete sequence of partitions compatible with $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, where $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$. Then, $\Phi(G, \mathcal{S}^l, v) = \Phi(G^\pi, \mathcal{T}^l, v^\pi)$ if and only if $\pi = InducedOrder(\mathsf{Q}')$.*

The return value of the function Φ, from Definition 2.29, is an integer value, because at the implementation, the fail is given by a hash value (integer), as a result of apply a hash function to some information of the further non-automorphic path in the search tree (the partition of the last generated level in this path).

Note from Observation 2.7 that two sequences of partitions incompatible between them will have a different set of failures, although these sets have not be disjoint necessarily. Thus, the function Φ is injective, but not necessary bijective.

2.6 Extending Sequences of Partitions

Once we have presented the theorems, lemmas, and invariants that will use for our algorithm, we obtain extra information about the sequence of partitions. This information has to be added to the sequence of partitions as complementary. So that, we have to define what an extended sequence of partitions is.

Definition 2.30 *Let $G = (V, R)$ be a graph. An* extended sequence of partitions E *for a graph G is a tuple $(\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma)$, where Q is a sequence of partitions, denoted $SeqPart(\mathsf{E})$, $\mathsf{B} = (B^0, ..., B^{t-1})$ indicate the backtrack level for each level, $\mathsf{L} = (L^0, ..., Lt-1)$ indicate the automorphism search tree limit level for each level, $\mathsf{F} = (\mathcal{F}^0, ..., \mathcal{F}^{t-1})$ are the multisets of the automorphism search tree failures for each level, and Γ is the generating set of $Aut(G)$.*

We can obtain the orbit partition of a graph G from the extended sequence of partitions of G, because the automorphism group $Aut(G)$ of G is derived from the generating set Γ (Definition 2.23), and it is used for compute each orbit (Definition 2.24). Then, the function $Orbits(\mathsf{E})$ return this orbit partition from an extended sequence of partitions of a graph G.

Chapter 3

Algorithms

In this chapter we will present the main algorithm of *Conauto-2.0*, that is called *AreIsomorphic*. This is a graph isomorphism testing algorithm, which is complete, i.e., given two graphs, it always returns an answer (positive or negative).

3.1 Main Algorithm of *Conauto-2.0*

Conauto-2.0 preserves the basic approach of its predecessor *conauto* [22, 15], but adds some new functionalities which will be explained in the following sections. It receives two graphs G and H as parameters and returns TRUE if both graphs are isomorphic, and FALSE if they are not.

First of all, this algorithm tests if both graphs have the same number of vertices and arcs which are both (non complete) graph invariants (see Definition 2.7). That is easy to test, and it is a necessary condition for isomorphism. Then, it generates initial partitions of the vertices of both graphs based on their degrees; $\mathcal{D}_G$ is the degree partition of G, and $\mathcal{D}_H$ the degree partition of H. If these partitions are not compatible (G and H differ in the number of vertices of some degree), the graphs cannot be isomorphic. Generating the degree partitions and checking for their compatibility is fast and can simplify the search for an isomorphism between G and H, since vertices in one cell of $\mathcal{D}_G$ can only be mapped to vertices in the corresponding cell of $\mathcal{D}_H$ (they can only be mapped to vertices with their same degree). Unfortunately, for regular graphs, this degree partition has only one cell, what means that each vertex in one partition (or graph) can be mapped to any one in the other partition (or graph).

If the degree partitions $\mathcal{D}_G$ and $\mathcal{D}_H$ are compatible, Algorithm 2, *GenerateSequenceOfPartitions*, is used to generate the sequence of partitions Q_G for graph G. After that, Algorithm 7, *FindAutomorphisms*, performs a complete search for automorphisms, while computing the automorphism group of the graph (a set of generators), and obtains the corresponding extended sequence of partitions E_G for graph G. If the sequence of partitions *SeqPart*(E_G) has no backtrack levels, Algorithm 15, *Match*, will attempt to find a sequence of partitions for graph H which is compatible with *SeqPart*(E_G). That is because Algorithm 15, *Match*, will not use backtracking and will be polynomial time (it is explained later, in Section 3.4). Otherwise, if the sequence of partitions *SeqPart*(E_G) has backtrack levels, the algorithm proceeds to compute graph H, in the same way it did for graph G. The sequence of partitions Q_H is generated, and then an extended sequence of partitions E_H is obtained while looking for automorphisms by means of Algorithm 7.

Algorithm 1 Test whether G and H are isomorphic (*conauto-2.0*).

```
AreIsomorphic(G, H) : boolean
 1 - - let $G = (V_G, R_G)$ and $H = (V_H, R_H)$
 2 if $(|V_G| \neq |V_H|) \vee (|R_G| \neq |R_H|)$ then
 3     return FALSE
 4 else
 5     $\mathcal{D}_G \leftarrow DegreePartition(G)$
 6     $\mathcal{D}_H \leftarrow DegreePartition(H)$
 7     if $\mathcal{D}_G$ and $\mathcal{D}_H$ are not compatible under $G$ and $H$ respectively then
 8         return FALSE
 9     else
10         $\mathsf{Q}_G \leftarrow GenerateSequenceOfPartitions(G, \mathcal{D}_G)$
11         $\mathsf{E}_G \leftarrow FindAutomorphisms(G, \mathsf{Q}_G)$
12         if $BacktrackAmount(SeqPart(\mathsf{E}_G)) = 0$ then
13             return $(0 \leq Match(0, G, H, \mathsf{E}_G, \mathcal{D}_H, \{\Gamma_H\})$
14         end if
15         $\mathsf{Q}_H \leftarrow GenerateSequenceOfPartitions(H, \mathcal{D}_H)$
16         $\mathsf{E}_H \leftarrow FindAutomorphisms(H, \mathsf{Q}_H)$
17         - - let $Aut(G) = <\Gamma_G>$, from $\mathsf{E}_G$
18         - - let $Aut(H) = <\Gamma_H>$, from $\mathsf{E}_H$
19         if $|Aut(G)| \neq |Aut(H)| \vee |Orbits(\mathsf{E}_G)| \neq |Orbits(\mathsf{E}_H)|$ then
20             return FALSE
21         end if
22         if $BacktrackAmount(SeqPart(\mathsf{E}_G)) \leq BacktrackAmount(SeqPart(\mathsf{E}_H))$ then
23             return $(0 \leq Match(0, G, H, \mathsf{E}_G, \mathcal{D}_H, \{\Gamma_H\}))$
24         else
25             return $(0 \leq Match(0, H, G, \mathsf{E}_H, \mathcal{D}_G, \{\Gamma_G\}))$
26         end if
27     end if
28 end if
```

Once both extended sequence of partitions, E_G and E_H, are complete we can obtain the automorphism group (from Definition 2.23) for graphs G and H. Then, automorphism group sizes of graphs G and H are compared to be equal, and in the same way, their number of orbits. These are necessary conditions for isomorphism (non complete graph invariants).

Now, the sequence of partitions with less backtrack levels is chosen as the target sequence and a new sequence of partitions for the other graph, compatible with the target, is searched for by Algorithm 15. If graphs G and H are isomorphic, it will return a positive value (the level in which the compatibility between partitions was found), otherwise it returns a negative value. Note that the partition with larger number of backtrack levels (either Q_G or Q_H) is dropped after being generated. However, generating a sequence of partitions takes (only) polynomial time, (and it is not guaranteed to be optimal in the number of backtrack levels). Besides, the sequences of partitions may be very different. Therefore, in practical it is worth generating a sequence of partitions for each graph, and then choosing the one which will generate less backtrack levels during the search process.

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

```
GenerateSequenceOfPartitions$(G, \mathcal{D})$ : sequence of partitions
 1 - - let $G = (V, R)$
 2 - - for all $l > 0$, if $\mathcal{S}^l$ is defined, let $\mathcal{S}^l = (S^l_1, ..., S^l_{r_l})$, $V^l = \bigcup_{j=1}^{r_l} S^l_j$
 3 $\mathcal{S}^0 \leftarrow \mathcal{D}$
 4 for each $S^0_x \in \mathcal{S}^0$ do
 5     $\mathit{Valid}(S^0_x) \leftarrow (|\mathcal{S}^0| > 1) \wedge \mathit{HasLinks}(S^0_x, V^0, G)$
 6 end for
 7 $l \leftarrow 0$
 8 while $\exists S^l_x \in \mathcal{S}^l : (|S^l_x| > 1) \wedge \mathit{HasLinks}(S^l_x, V^l, G)$ do
 9     $P^l \leftarrow \mathit{IndexBestPivot}(\mathcal{S}^l, G)$
10     if $|S^l_{P^l}| = 1$ then
11         $R^l \leftarrow$ VERTEX
12         $v \leftarrow$ the only vertex in $S^l_{P^l}$
13         $\mathcal{S}^{l+1} \leftarrow \mathit{VertexRefinement}(\mathcal{S}^l, v, G_{V^l})$
14     else
15         $\mathit{success} \leftarrow$ FALSE
16         while $\mathit{Valid}(S^l_{P^l}) \wedge \neg \mathit{success}$ do
17             $\mathit{Valid}(S^l_{P^l}) \leftarrow$ FALSE
18             $R^l \leftarrow$ SET
19             $\mathcal{S}^{l+1} \leftarrow \mathit{SetRefinement}(\mathcal{S}^l, S^l_{P^l}, G_{V^l})$
20             $\mathit{success} \leftarrow \exists S^{l+1}_x, S^{l+1}_{x+1} \in \mathcal{S}^{l+1} : S^{l+1}_x, S^{l+1}_{x+1} \subset S^l_y$ for some $S^l_y \in \mathcal{S}^l$
21             if $\neg \mathit{success}$ then
22                 $P^l \leftarrow \mathit{IndexBestPivot}(\mathcal{S}^l, G)$
23             end if
24         end while
25         if $\neg \mathit{success}$ then
26             $R^l \leftarrow$ BACKTRACK
27             $v \leftarrow$ any vertex in $S^l_{P^l}$
28             $\mathcal{S}^{l+1} \leftarrow \mathit{VertexRefinement}(\mathcal{S}^l, v, G_{V^l})$
29         end if
30     end if
31     $l \leftarrow l + 1$
32     for each $S^l_x \in \mathcal{S}^l$ do
33         - - let $S^{l-1}_y \in \mathcal{S}^{l-1} : S^l_x \subseteq S^{l-1}_y$
34         $\mathit{Valid}(S^l_x) \leftarrow \mathit{HasLinks}(S^l_x, V^l, G) \wedge (\mathit{Valid}(S^{l-1}_y) \vee (|S^l_x| < |S^{l-1}_y|))$
35     end for
36 end while
37 $t \leftarrow l$
38 $\mathsf{S} \leftarrow (\mathcal{S}^0, ..., \mathcal{S}^t)$; $\mathsf{R} \leftarrow (R^0, ..., R^{t-1})$; $\mathsf{P} \leftarrow (P^0, ..., P^{t-1})$
39 return $(\mathsf{S}, \mathsf{R}, \mathsf{P})$
```

3.2 Generation of a Sequence of Partitions

In this section we will explain all the algorithms related with the generation of a sequence of partitions for a graph. The approach of the earlier version of *conauto* for the generation of this sequence of partitions is the same, because instead of finding a specific canonical form of both graphs to be compared, we take any easy to reproduce form (ordering of the vertices) of one of the graphs. Furthermore, we also use partition (vertex or set) refinement to classify vertices. Partition refinement has been traditionally performed by splitting cells according to the adjacencies their vertices have with all the cells in a partition (see for example [33, 17, 1]). However, this can be quite costly. Therefore, we do things the other way round; i.e., we take

Algorithm 3 Find the best set $S_i \in \mathcal{S}$ to be used as a pivot.

```
IndexBestPivot($\mathcal{S}, G$) : integer
1 $b \leftarrow$ IndexBestValidPivot($\mathcal{S}, G$)
2 if $\neg$Valid($S_b$) $\wedge |\mathcal{S}| > 1$ then
3     $b \leftarrow$ IndexBestIndividualizedCell($\mathcal{S}, G$)
4 end if
5 return $b$
```

cells, not to try to have them split, but to try to split other cells (or itself) using vertex or set refinement. This approach is much less costly in terms of time and, on the short term, also space, but, on the long term, it needs more space, and leads to the same stable (or equitable) partition. Furthermore, it is not necessary to consider singleton cells (cells with only one vertex) more than once [1]. Hence, we discard singleton cells once they have been used for a vertex refinement. This reduces the complexity of the problem, and reduces the memory used for the algorithm. The main feature of this version of *conauto* for the generation of the sequence of partitions is the way we choose the pivot cell for vertex individualization, in order to obtain the minimum number of backtrack levels in the final partition.

Algorithm 2, *GenerateSequenceOfPartitions*, starts from the degree partition $\mathcal{D}$ of a graph G, and generates successive partition refinements, until it finds a partition such that the vertices in cells with more than one vertex have no adjacencies with the remaining vertices in that partition. New partitions are generated from their previous ones in the following way:

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 (lines 10-13).

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 (lines 15-24).

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 (lines 25-29).

Valid is an attribute of the cells, used to improve the performance in Case 2. A cell S_x^l is invalid if it has been proved to be unable to split any cells in partition $\mathcal{S}^l$, and valid otherwise. Thus, new cells are valid unless they have no remaining links, since a cell without links will never be able to split any cell. Before a cell is used as the pivot set for a refinement by set, it is marked invalid in advance, because, if it is not able to split any cell, it will be proved invalid, and if it is able to split some cell, once it has been used, it has split all the cells to its best, and it will never be able to split any of the subcells it has generated (otherwise, it would have split them at this point). Then, if it does not split itself with this refinement, it will remain invalid, whilst if it does, its subcells will be valid, since they are new, and they are not known to be invalid yet.

The task of choosing the pivot set among a set of cells is done by Algorithm 3, *IndexBestPivot*. This algorithm behaves as follows:

- It chooses the smallest valid cell with more remaining links (than those of same size), if such a cell exists. If this cells is a singleton one, this corresponds to Case 1 above. Otherwise, it correspond to Case 2 above.

- If there are no valid cells and there are more than one cell in the current partition, it

Algorithm 4 Find the best valid set $S_i \in \mathcal{S}$ to be used as a pivot.

```
IndexBestValidPivot($\mathcal{S}, G$) : integer
 1 - - let $\mathcal{S} = (S_1, ..., S_r)$ and $V = \bigcup_{i=1}^{r} S_i$
 2 $b \leftarrow 1$
 3 for $i \leftarrow 2$ to $r$ do
 4     if $Valid(S_i)$ then
 5         if $\neg Valid(S_b) \vee (|S_b| > |S_i|) \vee (|S_i| = |S_b| \wedge NumLinks(S_i, V, G) > NumLinks(S_b, V, G))$ then
 6             $b \leftarrow i$
 7         end if
 8     end if
 9 end for
10 return $b$
```

Algorithm 5 Find the best set $S_i \in \mathcal{S}$ to be used as a pivot to individualize.

```
IndexBestIndividualizedCell($\mathcal{S}, G$) : integer
 1 - - let $\mathcal{S} = (S_1, ..., S_r)$ and $V = \bigcup_{i=1}^{r} S_i$
 2 $C = \{i : (\nexists j, j < i : NumLinks(S_j, V, G) = NumLinks(S_i, V, G) \wedge |S_j| = |S_i|)\}$
 3 $b \leftarrow 1$
 4 $n \leftarrow 0$
 5 for each $c \in C$ do
 6     $\mathcal{T} \leftarrow NextEquitablePartition(\mathcal{S}, G, c)$
 7     - - let $\mathcal{T} = \{T_1, ..., T_{r'}\}, W = \bigcup_{i=1}^{r'} T_i$
 8     if $\nexists T_x \in \mathcal{T} : (|T_x| > 1) \wedge HasLinks(T_x, W, G)$ then (e.i. LAST partition reached)
 9         return $c$
10     else if $\mathcal{T}$ is subpartition of $\mathcal{S}$ then
11         return $c$
12     else if $r' + (|V| - |W|) > n$ then
13         $b \leftarrow c$
14         $n \leftarrow r' + (|V| - |W|)$
15     else if $r' + (|V| - |W|) = n \wedge |S_b| > |S_c|$ then
16         $b \leftarrow c$
17     end if
18 end for
19 return $b$
```

chooses one among them using Algorithm 5, *IndexBestIndividualizedCell*, which returns an optimal cell for individualization. This corresponds to Case 3 above.

The pivot set (for vertex and set refinements) is chosen according to three different criteria: first, it is better a pivot set which has links than one without links, since a pivot set with no links will not be able to split any cell. Among cells with links, a valid one is preferred, since an invalid cell will not be used for a set refinement, and would lead to a backtrack level, which is the algorithm's worst option. Finally, a smaller cell is preferred, since it will be faster to process than a bigger one. Among invalid cells (it would lead to a backtrack level), Algorithm 5 is responsible for choosing the pivot set. It is explained in next section.

3.2.1 Choosing a Cell for Individualization

Algorithm 5, *IndexBestIndividualizedCell*, starts from the partition $\mathcal{S}$ and the graph G. It attempts to choose the optimal pivot cell for vertex individualization in this partition. For each possible combination of cell size and available degree, the first cell in the partition, with that

Algorithm 6 Generate the next equitable level of the partition $\mathcal{S}$.

```
NextEquitablePartition($\mathcal{S}, G, p$) : Partition
 1 - - let $\mathcal{S} = (S_1, ..., S_r)$ and $V = \bigcup_{i=1}^{r} S_i$
 2 $v \leftarrow$ the first vertex of $S_p$
 3 $\mathcal{T} \leftarrow VertexRefinement(\mathcal{S}, v, G_V)$
 4 - - let $\mathcal{T} = (T_1, ..., T_s)$ and $W = \bigcup_{i=1}^{s} T_i$
 5 for each $T_x \in \mathcal{T}$ do
 6     - - let $S_y \in \mathcal{S} : T_x \subseteq S_y$
 7     $Valid(T_x) \leftarrow HasLinks(T_x, W, G) \wedge (Valid(S_y) \vee (|T_x| < |S_y|))$
 8 end for
 9 $P \leftarrow IndexBestValidPivot(\mathcal{T}, G)$
10 $success \leftarrow FALSE$
11 while $Valid(T_P) \wedge \exists T_x \in \mathcal{T} : (|T_x| > 1) \wedge HasLinks(T_x, W, G)$ do
12     if $|T_P| = 1$ then
13         $v \leftarrow$ the only vertex in $T_P$
14         $\mathcal{T}' \leftarrow VertexRefinement(\mathcal{T}, v, G_W)$
15         $success \leftarrow TRUE$
16     else
17         $Valid(T_P) \leftarrow FALSE$
18         $\mathcal{T}' \leftarrow SetRefinement(\mathcal{T}, T_P, G_W)$
19         - - let $\mathcal{T}' = (T'_1, ..., T'_{s'})$ and $W = \bigcup_{i=1}^{s'} T'_i$
20         $success \leftarrow \exists T'_x, T'_{x+1} \in \mathcal{T}' : T'_x, T'_{x+1} \subset T_y$ for some $T_y \in \mathcal{T}$
21     end if
22     if $success$ then
23         $success \leftarrow FALSE$
24         - - let $\mathcal{T}' = (T'_1, ..., T'_{s'})$ and $W = \bigcup_{i=1}^{s'} T'_i$
25         for each $T'_x \in \mathcal{T}'$ do
26             - - let $T_y \in \mathcal{T} : T'_x \subseteq T_y$
27             $Valid(T'_x) \leftarrow HasLinks(T'_x, W, G) \wedge (Valid(T_y) \vee (|T'_x| < |T_y|))$
28         end for
29         $\mathcal{T} \leftarrow \mathcal{T}'$
30     end if
31     $P \leftarrow IndexBestValidPivot(\mathcal{T}, G)$
32 end while
33 return $\mathcal{T}$
```

size and available degree, is considered (line 2). Then, for every one of these considered cells, algorithm 6, *NextEquitablePartition*, returns the next equitable refined partition $\mathcal{T}$ from the partition $\mathcal{S}$. Let c be the cell chose. If the partition $\mathcal{T}$ is the last partition in the sequence, cell c will be the pivot cell returned (lines 8-9). If the partition $\mathcal{T}$ is subpartition of $\mathcal{S}$, cell c will be the pivot cell returned (lines 10-11). Unless one of this two criteria were fulfilled, the algorithm will behave as follows:

- We add the number of discarded vertices with the number of cells of $\mathcal{T}$. Then, the cell of $\mathcal{S}$ which yields the biggest such sum is chosen as the pivot cell.

- If the such sum of two considered cells of partition $\mathcal{S}$ is equal, the smaller cell is chosen, since it will be faster to process than a bigger one.

Note that, since only one vertex in each cell is considered (the first one), the choice is not isomorphism invariant and, hence, cannot be used for canonical labeling.

Algorithm 7 Look for automorphisms.

```
$FindAutomorphisms(G, \mathsf{Q})$ : extended sequence of partitions
1  - - let $G = (V, R)$, $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$
2  - - let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathcal{S}^l = (S^l_1, ...S^l_{r_l})$, $V^l = \bigcup_{i=1}^{r_l} S^l_i$, for all $l \in \{0, ..., t\}$
3  $\mathsf{B} \leftarrow ComputeBacktrackLevels(\mathsf{S}, \mathsf{R}, G)$
4  $\mathsf{L} \leftarrow ComputeLimitLevels(\mathsf{S}, \mathsf{R}, G)$
5  $\mathsf{F} \leftarrow \emptyset$
6  $\Gamma \leftarrow ProcessCellsWithNoLinks(G, \mathsf{Q})$
7  - - let $\mathsf{E} = (\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma)$, $\Gamma = \{\gamma_1, ..., \gamma_{|\Gamma|}\}$
8  $l \leftarrow t - 1$
9  while $l \geq 0$ do
10     if $R^l$ = BACKTRACK then
11         $\mathsf{E}' \leftarrow \mathsf{E}$
12         $\mathsf{E} \leftarrow CheckAutomorphisms(G, \mathsf{E}', l)$
13     end if
14     $l \leftarrow l - 1$
15 end while
16 return $\mathsf{E}$
```

Algorithm 6, *NextEquitablePartition*, starts from the partition $\mathcal{S}$, the graph G, and a pivot cell index p of the partition. The first vertex v of this pivot cell is individualized (line 3) and the obtained partition is subsequently refined until an equitable partition is reached. This algorithm is similar to Algorithm 2, but does not compute a complete sequence of partitions and the pivot cell selection is carried out by Algorithm 4. That is because we are only interested return a valid cell and not to individualize (equitable level). We correct the choice of the first vertex of the pivot cell (at individualization) instead of any other different vertex in the cell, with the failure management (explained in section 2.5) while looking for automorphisms.

3.3 Search For Automorphisms

In this section we will show our automorphisms searching algorithm, which computes the automorphism group of a graph (i.e., set of generators, size of the automorphism group, and orbits). Because we use a backtrack algorithm, this implies a computationally hard work, so then we need some techniques in order to prune the automorphism search space. The use of these techniques are described in this chapter. Since the detection of automorphisms consist on reproduce (if exists, and consequently exists an automorphism) an alternative and compatible sequences of partitions from the original one, this same techniques (or part of them) will be used in next section (Section 3.4), towards finding an isomorphism of a graph.

Algorithm 7, *FindAutomorphisms*, performs the search of automorphisms. It starts from the graph G and its sequence of partitions Q. First of all, it computes the backtrack levels and the limit-search levels, in order to use both Theorems 2.3 and 2.2 respectively. Next, it uses Algorithm 10, *ProcessCellsWithNoLinks*, which attempt to use Lemma 2.1 at each level in $\{0, ..., t\}$ in order to find the trivial automorphisms. Then, the search for automorphisms at each backtrack level begins from the last partition in the sequence, traversing it up to the first. This way, Lemma 2.3 will be applicable, so the automorphisms already found may be used when processing previous partitions in the sequence. The searching for automorphisms itself, at each backtrack level, is performed through Algorithm 11, *CheckAutomorphisms*.

Algorithm 8 Compute the backtrack level of each backtrack level of a Partition.

```
ComputeBacktrackLevels$(\mathsf{S}, \mathsf{R}, G)$ : sequence of integers
 -- let $\mathsf{S} = \{\mathcal{S}^0, ..., \mathcal{S}^t\}$, $\mathsf{R} = \{R^0, ..., R^{t-1}\}$, $G = (V, R)$
 for each $l \in \{0, ..., t-1\}$ do
     if $R^l = BACKTRACK$ then
         $k \leftarrow (l-1)$
         while $k \geq 0 \wedge (R^k \neq BACKTRACK \vee \mathcal{S}^l$ is subpartition of $\mathcal{S}^k$ do
             $k \leftarrow (k-1)$
         end while
         $B^l \leftarrow k$
     else
         $B^l \leftarrow (l-1)$
     end if
 end for
 $\mathsf{B} \leftarrow (B^0, ..., B^{t-1})$
 return $\mathsf{B}$
```

Algorithm 9 Compute the automorphism-search limit level of each backtrack level of a Partition.

```
ComputeLimitLevels$(\mathsf{S}, \mathsf{R}, G)$ : sequence of integers
 -- let $\mathsf{S} = \{\mathcal{S}^0, ..., \mathcal{S}^t\}$, $\mathsf{R} = \{R^0, ..., R^{t-1}\}$, $G = (V, R)$
 for each $k \in \{0, ..., t-1\}$ do
     if $R^k = BACKTRACK$ then
         $l \leftarrow (k+1)$
         while $l < t \wedge (R^l \neq BACKTRACK \vee \mathcal{S}^l$ is not subpartition of $\mathcal{S}^k$ do
             $l \leftarrow (l+1)$
         end while
         $L^k \leftarrow l$
     else
         $L^k \leftarrow t$
     end if
 end for
 $\mathsf{L} \leftarrow (L^0, ..., L^{t-1})$
 return $\mathsf{L}$
```

Algorithm 10 Process cells with no links.

```
ProcessCellsWithNoLinks$(G, \mathsf{Q})$ : Set of generators
 -- let $G = (V, R)$
 -- let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathcal{S}^l = (S^l_1, ...S^l_{r_l})$, $V^l = \bigcup_{i=1}^{r_l} S^l_i$, for all $l \in \{0, ..., t\}$
 $\pi \leftarrow InducedOrder(\mathsf{Q})$
 $\Gamma \leftarrow \emptyset$
 for each $l \in \{0, ..., t\}$ do
     for each $S^l_x \in \mathcal{S}^l : \neg HasLinks(S^l_x, V^l, G)$ do
         $u \leftarrow$ any vertex of $S^l_x$
         for each $v \in S^l_x : u \neq v$ do
             $\pi' \leftarrow Exchange(\pi, u, v)$
             $\Gamma \leftarrow \Gamma \cup \pi'$
         end for
     end for
 end for
 return $\Gamma$
```

Algorithm 11 Look for automorphisms at level l.

```
$CheckAutomorphisms(G, \mathsf{E}, l)$ : extended sequence of partitions
1  - - let $G = (V, R)$, $\mathsf{E} = (\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma)$
2  - - let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, $\mathsf{B} = (B^0, ..., B^{t-1})$, $\mathsf{L} = (L^0, ..., L^{t-1})$, $\mathsf{F} = \{F^0, ..., F^{t-1}\}$, $\Gamma = \{\gamma_1, ..., \gamma_{|\Gamma|}\}$
3  - - let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$
4  - - let $\mathcal{S}^i = (S^i_1, ...S^i_{r_i}), V^i = \bigcup_{j=1}^{r_i} S^i_j$, for all $i \in \{0, ..., t\}$
5  - - let $p$ the pivot vertex used to generate partition $\mathcal{S}^{l+1}$
6  for each $v \in (S^l_{P^l} \setminus \{p\})$ do
7      $Valid(OrbitOf(v, \Gamma)) \leftarrow TRUE$
8  end for
9  $success \leftarrow TRUE$
10 $F^l \leftarrow \emptyset$
11 for each $v \in (S^l_{P^l} \setminus \{p\})$ do
12     - - let $\Delta = \{\Gamma\}$, $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$
13     - - for all $i \in \{0, ..., l\}$, let $\mathcal{T}^i = \mathcal{S}^i$
14     - - for all $i \in \{l+1, ..., t\}$, if $\mathcal{T}^i$ is defined, let $\mathcal{T}^i = (T^i_1, ..., T^i_{r_i})$, $W^i = \bigcup_{j=1}^{r_i} T^i_j$
15     if $OrbitOf(p, \Gamma) \neq OrbitOf(v, \Gamma) \wedge Valid(OrbitOf(v, \Gamma))$ then
16         $T^{l+1} \leftarrow VertexRefinement(S^l, v, G_{V^l})$
17         $compatible \leftarrow$ if $S^{l+1}$ and $T^{l+1}$ are comptible under $G_{V^{l+1}}$ and $G_{W^{l+1}}$ respectively
18         if $compatible$ then
19             $(l', \Gamma', f) \leftarrow SubtreeCompatible(l+1, \mathsf{T}, G, \mathsf{E}, \Delta, l)$
20             $\Gamma \leftarrow \Gamma'$
21             $compatible \leftarrow l' > l$
22         end if
23         if $\neg compatible$ then
24             $Valid(OrbitOf(v, \Gamma)) \leftarrow FALSE$
25             $success \leftarrow FALSE$
26             $F^l \leftarrow F^l \uplus f$
27         end if
28     end if
29     if $\neg Valid(OrbitOf(v, \Gamma))$ then
30         $F^l \leftarrow F^l \uplus GetOldFailure(\mathsf{S}, v, F^l, l)$
31     end if
32 end for
33 if $success$ then
34     $R^l \leftarrow VERTEX$
35 end if
36 return $(\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma')$
```

Algorithm 11, *CheckAutomorphisms*, looks for vertices equivalent to the pivot vertex, from the same cell.

It starts from a graph G, an extended sequence of partitions E, and the level l in which it have to search for automorphisms. First of all, the orbits of all the vertices in the pivot cell, except vertex p used in the original sequence of partitions, are marked valid. Observe that p does not need to be stored since it can be identified as the only vertex with links in $\mathcal{S}^l$ that is not in $\mathcal{S}^{l+1}$. Each vertex v of the pivot cell, except p, that belongs to an orbit different from the pivot vertex, and which is valid, is processed. Then, vertex v is individualized, and it is obtained a new partition $\mathcal{T}^{l+1}$, which is checked to be compatible with $\mathcal{S}^{l+1}$. If it is, the algorithm tries to generate an alternative sequence of partitions compatible with the original one by means of Algorithm 12, *SubtreeCompatible*. If it success, the returned level by this algorithm will be the level in which the compatibility (and consequently, and automorphism) was found, so that

$l' > l$. Then, the set of generators Γ will be updated. Otherwise, the returned level will be the backtrack level (so that, a value lower or equal than l), and the set of generator Γ will be preserved. Last, in any case the vertex v does not yield an automorphism, its orbit is marked as non valid, in order not to process any equivalent vertex, which will yield a similar result, and the returned failure is recorded in the multiset of failures F^l. On the other hand, if the vertex was not valid, is known that not yields any automorphism and the failure produced by any vertex which belongs the same orbit (recall that it is an invariant) is recorded in F^l, but the vertex will be not even attempt to individualize.

When, at a backtracking point, all the vertices in the pivot cell are found to be equivalent, R^l is changed form BACKTRACK to VERTEX. Recall that, from Lemma 2.4, 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.

The generation of the alternative sequences of partitions is done using Algorithm 12, which is a recursive algorithm. First of all, before generating the partition $\mathcal{T}^{l+1}$ as an alternative to $\mathcal{S}^{l+1}$, the following conditions are checked (note that a pre-condition of this algorithm is that partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ are compatible):

1. If partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ are identical (lines 8-14), then it is not necessary to go further, since the rest of both sequences must be the same. It is easy to complete the alternative sequence of partitions T and obtain the automorphism π, which is added into the set of generators Γ, and algorithm returns with return level value l notifing success up on.
2. If limit level L^k is reached (lines 15-18), then Theorem 2.2 is applied by means of the function *ProcessSubpartition*. Such that function implements Defintion 2.28, obtaining the automorphism π, which is added into the set of generators Γ, and algorithm returns with return level value l, notifying success upward.
3. If T is a complete alternative sequence of partitions (lines 19-21), then the automorphism π is obtained from T, which is added into the set of generators Γ, and algorithm returns with return level value l, notifying success upward.

If the algorithm finds $R^k = \text{BACKTRACK}$ at some level, it might be possible to find a sequence of partitions compatible with the original one, for some vertex in the pivot set. Algorithm 13, *DeepeningInBacktrack*, performs this work.

Finally, the partition $\mathcal{T}^{l+1}$ is generated using the same kind of refinement R^l with the corresponding pivot cell $T^l_{p^l}$. If $\mathcal{T}^{l+1}$ and $\mathcal{S}^{l+1}$ are compatible the generation of an alternative sequence of partitions follows with algorithm *SubtreeCompatible* again. If the returned value level l' is not the same that the current level l, then this value l' is returned up on. Note that if $l' > l$ then an automorphism will be reported. Otherwise ($l' < l$), an unsuccessful generation of an alternative sequence of partitions will be reported and the search will attempt to resume from (at most) level l', which may performs a backjumping depends on the l' value. Furthermore, the failure is set in f, for being recorded upward. Observe that, if a fail arise at an erased backtrack level (now vertex), then the fail is set to negative value, because no fail supose to be there, since all vertices in $T^l_{p^l}$ are equivalent. Thus, the fail value should be recorded in the past backtrack level l, but it did not (has not failures) and then, it is especially marked with negative value for recording at level k. In any other case (either level $l+1$ not compatible or $l' = l$), the value B^l is returned up on.

The backtracking in the search for automorphisms is performed by Algorithm 13, which is used when a backtrack level is reached in the search for automorphisms. In this case, some vertices

Algorithm 12 Try to generate a compatible sequence of partitions.

```
SubtreeCompatible$(l, \mathsf{T}, G, \mathsf{E}, \Delta, k)$ : tuple (integer, Set of generators, integer)
 1 - - let $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$
 2 - - for all $i \in \{0, ..., l\}$, let $\mathcal{T}^i = (T_1^i, ..., T_{r_i}^i), W^i = \bigcup_{j=1}^{r_i} T_j^i$
 3 - - for all $i \in \{l+1, ..., t\}$, if $\mathcal{T}^i$ is defined, let $\mathcal{T}^i = (T_1^i, ..., T_{r_i}^i), W^i = \bigcup_{j=1}^{r_i} T_j^i$
 4 - - let $G = (V, R)$, $\mathsf{E} = (\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma)$
 5 - - let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, $\mathsf{B} = (B^0, ..., B^{t-1})$, $\mathsf{L} = (L^0, ..., L^{t-1})$, $\mathsf{F} = \{F^0, ..., F^{t-1}\}$, $\Gamma = \{\gamma_1, ..., \gamma_{|\Gamma|}\}$
 6 - - let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$,
 7 - - for all $i \in \{0, ..., t\}$, let $\mathcal{S}^i = (S_1^i, ..., S_{r_i}^i), V^i = \bigcup_{j=1}^{r_l} S_j^i$
 8 $f \leftarrow 0$
 9 if $\mathcal{T}^l = \mathcal{S}^l$ then
10     for each $i \in \{l+1, ..., t\}$ do $\mathcal{T}^i \leftarrow \mathcal{S}^i$
11     $\pi \leftarrow$ InducedOrder$((\mathsf{T}, \mathsf{R}, \mathsf{P}))$
12     return $(l, \Gamma \cup \pi, f)$
13 end if
14 if $l = L^k \wedge l \neq t$ then
15     $\pi \leftarrow$ ProcessSubpartition$(\mathsf{Q}, \mathsf{T}, l)$
16     return $(l, \Gamma \cup \pi, f)$
17 end if
18 if $l = t \wedge \mathcal{S}^l$ and $\mathcal{T}^l$ are compatible under $G_{V^l}$ and $G_{W^l}$ respectively then
19     $\pi \leftarrow$ InducedOrder$((\mathsf{T}, \mathsf{R}, \mathsf{P}))$
20     return $(l, \Gamma \cup \pi, f)$
21 else if $R^l$ = BACKTRACK then
22     return DeepeningInBacktrack$(l, \mathsf{T}, G, \mathsf{E}, \Delta, k)$
23 else
24     if $R^l$ = VERTEX then
25         $v \leftarrow$ any vertex in $T_{P^l}^l$
26         $\mathcal{T}^{l+1} \leftarrow$ VertexRefinement$(\mathcal{T}^l, v, G_{W^l})$
27     else (i.e. $R^l$ = SET)
28         $\mathcal{T}^{l+1} \leftarrow$ SetRefinement$(\mathcal{T}^l, T_{P^l}^l, G_{W^l})$
29     end if
30     if $S^{l+1}$ and $T^{l+1}$ are compatible under $G_{V^{l+1}}$ and $G_{W^{l+1}}$ respectively then
31         $(l', \Gamma', f') \leftarrow$ SubtreeCompatible$(l+1, \mathsf{T}, G, \mathsf{E}, \Delta, k)$
32         if $l' \neq l$ then
33             return $(l', \Gamma', f')$
34         end if
35     end if
36     - - let $p$ the pivot vertex used for refining level $k$
37     $f \leftarrow$ GetFailure$(G, \mathcal{T}^l, p)$
38     if $R^l$ = VERTEX $\wedge |T_{P^l}^l| > 1$ then
39         $f \leftarrow (-1)$
40     end if
41 end if
42 return $(B^l, \Gamma, f)$
```

of the pivot cell (maybe all of them) will be individualized and the looking for an alternative sequence of partitions will go on.

It starts computing the partial orbits O using Algorithm 14, *ComputePartialOrbits*, and also a set of set of generators is obtained in Δ' from the previous Δ, used for the automorphism group management. Then, this partial orbits (may be the trivial orbits) are marked valid and each vertex v from the pivot cell $T_{P^l}^l$ whose orbit is valid is attempt to individualize and generate an alternative sequence of partitions. When the orbit of the current vertex v is valid, inmediatly

Algorithm 13 Deepen into a backtrack level searching a compatible sequence of partitions.

```
$DeepeningInBacktrack(l, \mathsf{T}, G, \mathsf{E}, \Delta, k)$ : tuple (integer, Set of generators, integer)
- - let $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$
- - for all $i \in \{0, ..., l\}$, let $\mathcal{T}^i = (T_1^i, ..., T_{r_i}^i), W^i = \bigcup_{j=1}^{r_i} T_j^i$
- - for all $i \in \{l+1, ..., t\}$, if $\mathcal{T}^i$ is defined, let $\mathcal{T}^i = (T_1^i, ..., T_{r_i}^i), W^i = \bigcup_{j=1}^{r_i} T_j^i$
- - let $G = (V, R)$, $\mathsf{E} = (\mathsf{Q}, \mathsf{B}, \mathsf{L}, \mathsf{F}, \Gamma)$
- - let $\mathsf{Q} = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, $\mathsf{B} = (B^0, ..., B^{t-1})$, $\mathsf{L} = (L^0, ..., L^{t-1})$, $\mathsf{F} = \{F^0, ..., F^{t-1}\}$, $\Gamma = \{\gamma_1, ..., \gamma_{|\Gamma|}\}$
- - let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$,
- - for all $i \in \{0, ..., t\}$, let $\mathcal{S}^i = (S_1^i, ..., S_{r_i}^i), V^i = \bigcup_{j=1}^{r_l} S_j^i$
- - let $k'$ the previous equitable level, such that $k \leq k' < l$
$F \leftarrow \emptyset$
$(\mathsf{O}, \Delta') \leftarrow ComputePartialOrbits(\mathsf{T}, G, \Delta, k', l)$
for each $v \in T_{P^l}^l$ do
    $Valid(Orb(v, \mathsf{O})) \leftarrow$ TRUE
end for
for each $v \in T_{P^l}^l$ do
    if $Valid(Orb(v, \mathsf{O}))$ then
        $Valid(Orb(v, \mathsf{O})) \leftarrow$ FALSE
        $\mathcal{T}^{l+1} \leftarrow VertexRefinement(\mathcal{T}^l, v, G_{W^l})$
        if $S^{l+1}$ and $T^{l+1}$ are compatible under $G_{V^{l+1}}$ and $G_{W^{l+1}}$ respectively then
            $(l', \Gamma', f) \leftarrow SubtreeCompatible(l+1, \mathsf{T}, G, \mathsf{E}, \Delta', k)$
            if $l' \neq l$ then
                return $(l', \Gamma', f)$
            end if
        end if
        $F \leftarrow F \uplus \{\biguplus_{i=1}^{|Orb(v,\mathsf{O})|}\{f\}\}$
        if $F \nsubseteq F^l$ then
            return $(B^l, \Gamma)$
        end if
    endif
end for
return $(B^l, \Gamma)$

```

this orbit is marked non-valid in order to any other vertex of this orbit will not be attempt to individualize. Next, the partition $\mathcal{T}^{l+1}$ is generated by *VertexRefinement* using the vertex v as pivot vertex, and compatibility between $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$ is checked. If they are compatible, algorithm *SubtreeCompatible* goes on generating a compatible sequence of partitions. If the return level value l' is not equal than the current level l, then either an automorphism was found ($l' > l$) or a backjumping is performed hopping this backtrack level l ($l' < l$) and the algorithm returns with this value l'. Otherwise, neither an automorphism was found nor backjumping is performed, so then a failure f arises and is added $|Orb(v, \mathsf{O})|$ times to the multiset F, which next is checked to be equal or a subset of F^l (necessary condition for being in an automorphism path of the search tree). If $F \nsubseteq F^l$, then the search for a compatible sequence of partitions in this level l stop, because we are not in an automorphism path of the search tree and a backjumping is performed (hopping this current level up on) returning with level value B^l. Otherwise, the algorithm goes on with next vertex of $T_{P^l}^l$.

The automorphism group management is performed by Algorithm 14, *ComputePartialOrbits*. We start with a set of set of generators Δ, whose each set will be attempt to split (removing the singleton sets) in subsets of generators which fix the discarded vertices up to level l. First of all, the trivial orbits are created. And the goal is to create the partial orbits applicable at level

Algorithm 14 Compute partial orbits.

```
$ComputePartialOrbits(\mathsf{T}, G, \Delta, k, l)$ : tuple (Orbits, Generators multiset)
1  - - let $\mathsf{T} = (\mathcal{T}^0, ..., \mathcal{T}^t)$
2  - - for all $i \in \{0, ..., l\}$, let $\mathcal{T}^i = (T^i_1, ..., T^i_{r_i}), V^i = \bigcup_{j=1}^{r_i} T^i_j$
3  - - let $G = (V, R)$
4  - - let $\Delta = \{\Gamma^1, ..., \Gamma^n\}$, $\Gamma^j = \{\gamma^j_1, ..., \gamma^j_{m^j}\}$
5  $\Delta' \leftarrow \emptyset$
6  $\mathsf{O} \leftarrow \{\{v_i\} : v_i \in V\}$
7  for each $\Gamma^j \in \Delta : |\Gamma^j| > 1$ do
8      $\Gamma' \leftarrow \gamma^j_1$
9      for each $\gamma' \in \{\gamma^j_2, ..., \gamma^j_{m^j}\}$ do
10         if $\gamma^j_1$ and $\gamma'$ fixed vertices from $|V \setminus V^k|$ to $|V \setminus V^l|$ then
11             $\Gamma' \leftarrow \Gamma' \cup \{\gamma'\}$
12             for each $u, v : (\gamma^j_1)^{-1}(u) = (\gamma')^{-1}(v)$ do
13                 $\mathsf{O} \leftarrow merge(\mathsf{O}, Orb(u, \mathsf{O}), Orb(v, \mathsf{O}))$
14             end for
15         end if
16     end for
17     if $|\Gamma'| > 1$ then
18         $\Delta' \leftarrow \Delta' \cup \Gamma'$
19     end if
20     $\Gamma^j \leftarrow \Gamma^j \setminus \Gamma'$
21 end for
22 return $(\mathsf{O}, \Delta')$
```

l, for the set of partitions T, as follows:

- Each subset of generators $\Gamma' \in \Delta$ is split into other subsets that fix the discarded vertices, from level k to l, of T. Note that, we only need to check discarded vertices from level k, because the subsets were updated at level k from its previous corresponding backtrack level.
- Each pair of generators of these news subsets, are used for merge orbits (Definition 2.26) supported by Observation 2.4.

3.4 Matching

Algorithm 15, *Match*, is used to look for a sequence of partitions Q_H for a graph H, compatible with the target Q_G, for a graph G, starting from its degree partition $\mathcal{D}_H$. *Match* is a recursive backtracking algorithm which generates a new partition each time it is run, until it reaches the last partition in the sequence. It starts from the partition $\mathcal{T}$, that is compatible with $\mathcal{S}^l$, and then it generates a new partition $\mathcal{T}'$ using the same refinement (R^l) used to generate $\mathcal{S}^{l+1}$ form $\mathcal{S}^l$, with the corresponding pivot set T_{P^l}. If partitions $\mathcal{T}'$ and $\mathcal{S}^{l+1}$ are compatible, then it makes a recursive call to process the new partition.

This algorithm works in the following way: if an isomorphism of G and H is found, it returns t (the number of levels in the sequence of partitions, which is always positive). If it has been impossible to find such an isomorphism, it returns -1. The actual value returned indicates up to which level it is necessary to backtrack to continue the search (if it is smaller than l), that an isomorphism has been found (if it is t), or that another option must be taken at this level, if

Algorithm 15 Find a sequence of partitions compatible with the target.

```
Match($l, G, H, \mathsf{E}_G, \mathcal{T}, \Delta$) : integer
 1 - - let $\mathsf{Q}_G = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, let $\mathsf{S} = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathsf{R} = (R^0, ..., R^{t-1})$, $\mathsf{P} = (P^0, ..., P^{t-1})$
 2 - - let $\mathcal{S}^l = (S^l_1, ...S^l_{r_l}), V^l = \bigcup_{j=1}^{r_l} S^l_j$, for all $l \in \{0, ..., t\}$
 3 - - let $H = (V_H, R_H)$, let $\mathcal{T} = (T_1, ...T_{r_l}), W = \bigcup_{j=1}^{r_l} T_j$
 4 if $l = t$ then
 5     if $\forall x, y \in \{1, ..., r_l\}, ADeg(S^t_x, S^t_y, G) = ADeg(T_x, T_y, H)$ then
 6         return $t$
 7     end if
 8 else
 9     $X \leftarrow T_{P^l}$
10     if $R^l$ = BACKTRACK then
11         $(\mathsf{O}_H, \Delta') \leftarrow ComputePartialOrbits(\mathsf{T}, H, \Delta, 0, l)$
12         for each $v \in X$ do
13             $Valid(Orb(v, \mathsf{O}_H)) \leftarrow$ TRUE
14         end for
15         repeat
16             $v \leftarrow$ any vertex in $X$
17             $X \leftarrow X \setminus \{v\}$
18             if $Valid(Orb(v, \mathsf{O}_H))$ then
19                 $Valid(Orb(v, \mathsf{O}_H)) \leftarrow$ FALSE
20                 $\mathcal{T}' \leftarrow VertexRefinement(\mathcal{T}, v, H_W)$
21                 - - let $\mathcal{T}' = (T'_1, ..., T'_r), W' = \bigcup_{j=1}^{r} T'_j$
22                 if $\mathcal{S}^{l+1}$ and $\mathcal{T}'$ are compatible under $G_{V^{l+1}}$ and $H_{W'}$ respectively then
23                     $m \leftarrow Match(l + 1, G, H, \mathsf{E}_G, \mathcal{T}', \Delta')$
24                     if $l \neq m$ then
25                         return $m$
26                     end if
27                 end if
28             end if
29         until $X = \emptyset$
30     else
31         if $R^l$ = VERTEX then
32             $v \leftarrow$ any vertex in $X$
33             $\mathcal{T}' \leftarrow VertexRefinement(\mathcal{T}, v, H_W)$
34         else (i.e. $R^l$ = SET)
35             $\mathcal{T}' \leftarrow SetRefinement(\mathcal{T}, X, H_W)$
36         end if
37         - - let $\mathcal{T}' = (T'_1, ..., T'_r), W' = \bigcup_{j=1}^{r} T'_j$
38         if $\mathcal{S}^{l+1}$ and $\mathcal{T}'$ are compatible under $G_{V^{l+1}}$ and $H_{W'}$ respectively then
39             $m \leftarrow Match(l + 1, G, H, \mathsf{E}_G, \mathcal{T}', \Delta)$
40             if $l \neq m$ then
41                 return $m$
42             end if
43         end if
44 end if
45 return $B^l$
```

possible (if it is l). To do so, it works in the following way:

- If at level t all the adjacencies are satisfied, then it returns t (an isomorphism has been found).
- If R^l = VERTEX, then a vertex refinement is performed (line 33). If the resulting partition

is compatible with $\mathcal{S}^{l+1}$ (tested at line 38), then a recursive call is made, to proceed to the next level (line 39). Otherwise, since an incompatibility has been found, it returns to a previous level that satisfies the condition of Theorem 2.3 (line 45), computed in B^l. Note that, if the partition at the current level is equitable, then clearly Theorem 2.3 applies, and if the partition at the current level is not equitable, then the value $l-1$ would be returned (stored in B^l).

If a recursive call was made, then the value returned by this call must be evaluated. If the call returned a value which is bigger than l, then it must be t, and an isomorphism has been found, so that value must be returned also by this invocation to the algorithm (lines 40 and 41). If it returned a value smaller than l, it is necessary to backtrack, at least, to that level to be able to find an isomorphism. The reason is that a subsequent level, an incompatibility was found, and, applying Theorem 2.3, it was decided to backtrack to level m). Hence, that same value is returned to the caller (lines 40 and 41).

If the recursive call at line 39 returned a value $m = l$, then, since this is not a backtracking point, it is necessary to backtrack. Line 45 decide to which level it is necessary to backtrack from this level, just as if the incompatibility had been found at this level (computed in B^l).

- If R^l = SET, then a set refinement is performed, and the algorithm behaves the same as in the previous case, where R^l = VERTEX.
- In the case where R^l = BACKTRACK, the algorithm has to try possible matchings for the pivot vertex, until an isomorphism is found or all the choices have been tried or discarded. Observe that it is also used the algorithm *ComputePartialOrbits* in order not to try to use all the vertices in the pivot cell. For that, we use the orbits attribute, valid, which at the begining is set *TRUE* for all orbits. As a vertex is chosen to individualize, inmediatly its orbit is marked as non valid. When the vertex refinement results in a new partition $\mathcal{T}'$ that is compatible with $\mathcal{S}^{l+1}$, a recursive call is made. Otherwise, the next possible choice will be tried in the next iteration. As in the previous cases, this call returns a value m which can be bigger, equal, or less than l.

  If $m = l$, a new choice is tried, just as if the incompatibility were found at this level.

  If $m > l$, then, it must be t, in which case an isomorphism has been found, so that value is returned. This is done in lines 24 and 25.

  If $m < l$, then an incompatibility was found in a subsequent level, and applying the result of Theorem 2.3, it is not necessary to try any other possibility at this level, so m is directly returned. This is accomplished by lines 24 and 25.

  When all the possibilities have been tried unsuccessfully, backtracking is needed. Then, applying Theorem 2.3 in line 45, the algorithm tries to find a previous level where at least two of the cells in the current partition were not differentiated yet. If one such level is found, it is returned to the caller, so that the algorithm backtracks directly to that level, no matter whether there are intermediate backtracking points. If there is no such level, then -1 is returned, since no possible isomorphism can be found.

Observe that, althought the failure recording it is not used in algorithm *Match*, it could be used exactly in the same way as it is used in Algorithm 13. The failures arising, during the search for a compatible sequence of partitions, will be collected at each backtrack level, and compared against F_G. Recall that the failures through a removed backtrack level, have to be set to (-1).

At implementation, it is used as just described.

3.5 Correctness of the Algorithm

In this section we show that the proposed algorithm correctly determines whether two graphs are isomorphic. The algorithm generates compatible sequences of partitions for both graphs being tested. We will prove that compatible sequences of partitions induce an isomorphism between the graphs, and that, if such compatible sequences of partitions exist, our algorithm is able to find them.

Lemma 3.1 *Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two isomorphic graphs. Then, there are two compatible sequences of partitions $\mathsf{Q}_G = (\mathsf{S}_G, \mathsf{R}_G, \mathsf{P}_G)$, and $\mathsf{Q}_H = (\mathsf{S}_H, \mathsf{R}_H, \mathsf{P}_H)$ for graphs G and H, respectively.*

Proof: Let $\mathsf{Q}_G = (\mathsf{S}_G, \mathsf{R}_G, \mathsf{P}_G)$ be any sequence of partitions for graph G. Let $\mathsf{S}_G = (\mathcal{S}^0, ..., \mathcal{S}^t)$, $\mathcal{S}^0 = \mathit{DegreePartition}(G)$, $\mathsf{R}_G = (R_G^0, ..., R_G^{t-1})$, and $\mathsf{P}_G = (P_G^0, ..., P_G^{t-1})$, and for all $i \in \{0, ..., t\}$, let $\mathcal{S}^i = (S_1^i, ..., S_{r_i}^i)$, and $V^i = \bigcup_{j=1}^{r_i} S_j^i$.

Let m be a mapping of the vertices of V_G onto the vertices of V_H that preserves the (m exists since G and H are isomorphic). Then we will generate a sequence of partitions $\mathsf{Q}_H = (\mathsf{S}_H, \mathsf{R}_H, \mathsf{P}_H)$ for graph H which is compatible with Q_G. Let $\mathsf{S}_H = (\mathcal{T}^0, ..., \mathcal{T}^t)$, $\mathsf{R}_H = (R_G^0, ..., R_G^{t-1})$ (the same type of refinement is performed at each step), and $\mathsf{P}_H = (P_G^0, ..., P_G^{t-1})$ (corresponding pivot sets are used at each refinement). Moreover, for all $i \in \{0, ..., t\}$, m maps the vertices in corresponding cells of $\mathcal{S}^i$ and $\mathcal{T}^i$.

Let $\mathcal{T}^0 = \mathit{DegreePartition}(H)$. Note that, since G and H are isomorphic, $\mathcal{S}^0$ and $\mathcal{T}^0$ must be compatible (the number of vertices of each degree must be the same for both graphs). Hence, $|\mathcal{S}^0| = |\mathcal{T}^0|$. Let $\mathcal{S}^0 = (S_1^0, ..., S_r^0)$, and $\mathcal{T}^0 = (T_1^0, ..., T_r^0)$. Then, m maps the vertices in S_i to the vertices in T_i, for all $i \in \{1, ..., r\}$.

Now, by induction, we assume that compatibility exists up to partitions $\mathcal{S}^l$ and $\mathcal{T}^l$, i.e. for all $i \in \{0, ..., l\}$ partitions $\mathcal{S}^i$ and $\mathcal{T}^i$ are compatible, and m maps the vertices in corresponding cells of $\mathcal{S}^i$ and $\mathcal{T}^i$. Then we generate partition $\mathcal{T}^{l+1}$ and prove that it is compatible with $\mathcal{S}^{l+1}$, and that m still maps the vertices in corresponding cells of $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$.

Note first that if a cell S_s^l was discarded when deriving $\mathcal{S}^{l+1}$ from $\mathcal{S}^l$, that was because it had no remaining links. Since $\mathcal{S}^l$ and $\mathcal{T}^l$ are compatible, T_s^l can not have links either, and will also be discarded in $\mathcal{T}^{l+1}$. Then, depending on the value of R^l, three different cases arise in the generation of partition $\mathcal{S}^{l+1}$ from $\mathcal{S}^l$:

1. $R^l =$ VERTEX.
2. $R^l =$ SET.
3. $R^l =$ BACKTRACK.

In Case 1, for graph H, we can generate a new partition $\mathcal{T}^{l+1}$ from $\mathcal{T}^l$ using vertex refinement with the pivot set T_{Pl}^l, which contains a single vertex, image under m of the only vertex in S_{Pl}^l (from the induction hypothesis). Let $\mathcal{S}^l = (S_1^l, ..., S_r^l)$, and $\mathcal{T}^l = (T_1^l, ..., T_r^l)$. Also from the induction hypothesis, the vertices in cell $S_i^l \in \mathcal{S}^l$ are mapped under m to the vertices in cell $T_i^l \in \mathcal{T}^l$ for all $i \in \{1, ..., r\}$. Hence, if the pivot vertex in S_{Pl}^l has a certain kind of link with k vertices in some cell S_i^l, then the vertex in T_{Pl}^l must also have a link of that kind with k vertices in cell T_i^l. Otherwise, there would be vertices in S_i^l which could not be mapped by m to vertices in T_i^l for having different adjacencies with the corresponding pivot vertices. Therefore, the new cells generated will have the same number of vertices, and their vertices will have the same kind of adjacency with the respective pivot vertex, as their corresponding cells in $\mathcal{S}^{l+1}$. Hence, the new partition $\mathcal{T}^{l+1}$ must be compatible with partition $\mathcal{S}^{l+1}$, and the vertices every cell of $\mathcal{S}^{l+1}$ can only be mapped by m, to the vertices in its corresponding cell in $\mathcal{T}^{l+1}$.

In Case 2, we generate partition $\mathcal{T}^{l+1}$ using set refinement with the corresponding pivot set T_{Pl}^l. By the induction hypothesis, cells S_{Pl}^l and T_{Pl}^l must have the same adjacencies with the corresponding cells in partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ respectively. Therefore, the new cells generated will have the same adjacencies with the pivot set in both graphs. Hence, the new cells in $\mathcal{S}^{l+1}$ must be mapped by m to the corresponding new cells in $\mathcal{T}^{l+1}$, and partitions $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$ must be compatible.

In Case 3, let p be the pivot vertex chosen from cell S_{Pl}^l for the vertex refinement applied to partition $\mathcal{S}^l$. This vertex could be mapped by m to any vertex in T_{Pl}^l. However, one of them must be $m(p)$ since $\mathcal{S}^l$ and $\mathcal{T}^l$ are compatible and, from the induction hypothesis, the vertices in S_{Pl}^l can only be mapped to vertices in T_{Pl}^l. Using $m(p)$ as the pivot vertex, it is possible to generate a new partition $\mathcal{T}^{l+1}$ compatible with $\mathcal{S}^{l+1}$ since p and $m(p)$ have the same adjacencies with the corresponding cells in partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ respectively, as in Case 1. Hence, the new partition $\mathcal{T}^{l+1}$ must be compatible with partition $\mathcal{S}^{l+1}$, and m maps the vertices in corresponding cells in $\mathcal{S}^{l+1}$ and $\mathcal{T}^{l+1}$.

This way, we reach partitions $\mathcal{S}^t$ and $\mathcal{T}^t$. Compatibility of these final partitions is straightforward. Since all cells with remaining links are singleton ones, and m maps the vertices in corresponding cells, the adjacency between any two vertices v and w in singleton cells of partition $\mathcal{S}^t$ must be the same as the adjacency between $m(v)$ and $m(w)$ (corresponding cells) in partition $\mathcal{T}^t$. Thus, we complete the proof. ■

Lemma 3.2 *Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two graphs, $|V_G| = |V_H| = n$. 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 compatible sequences of partitions for graphs G and H respectively. Let π_G be the order induced by Q_G on the vertices of V_G, and let π_H be the order induced by Q_H on the vertices of V_H. Then, graphs G and H are isomorphic, and mapping m defined as $m(i^{\pi_G}) = i^{\pi_H}$ for all $i \in \{1, ..., |V_G|\}$ is an isomorphism of G and H.*

Proof: Let $Adj(G) = A$, and $Adj(H) = B$. Let $\mathsf{S}_G = (\mathcal{S}^0, ..., \mathcal{S}^t)$ and $\mathsf{T}_G = (\mathcal{T}^0, ..., \mathcal{T}^t)$. Since Q_G and Q_H are compatible sequences of partitions, their final partitions must also be compatible. Let $\mathcal{S}^t = (S_1^t, ..., S_r^t)$ and $\mathcal{T}^t = (T_1^t, ..., T_r^t)$ be the final partitions, and let $|V_G^t| = |V_H^t| = s$. Then, from Definition 2.19, we know that for all $x, y \in \{1, ..., r\}$, $ADeg(S_x^t, S_y^t, G) = ADeg(T_x^t, T_y^t, H)$. Since all non-singleton cells in the final partitions have no remaining links, this means that for all $i, j \in \{1, ..., s\}$, $A_{(n-s+i)^{\pi_G},(n-s+j)^{\pi_G}} = B_{(n-s+i)^{\pi_H},(n-s+j)^{\pi_H}}$. Hence, subgraphs $G_{V_G^t}$ and $H_{V_H^t}$ are isomorphic, and mapping m restricted to the vertices in V_G^t and V_H^t is an isomorphism of them.

Now, by induction, we assume that subgraphs $G_{V_G^l}$ and $H_{V_H^l}$ are isomorphic, and mapping m restricted to the vertices in V_G^l and V_H^l is an isomorphism of them. Then, we add the vertices in $V_G^{l-1} \setminus V_G^l$ and $V_H^{l-1} \setminus V_H^l$, and prove that subgraphs $G_{V_G^{l-1}}$ and $H_{V_H^{l-1}}$ are isomorphic, and mapping m restricted to the vertices in V_G^{l-1} and V_H^{l-1} is an isomorphism of them.

Note first that the vertices in $V_G^{l-1} \setminus V_G^l$ and $V_H^{l-1} \setminus V_H^l$ come from cells with no remaining links, or they are the pivot vertices used in the refinement, in case partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ are a vertex refinement of partitions $\mathcal{S}^{l-1}$ and $\mathcal{T}^{l-1}$, respectively.

Let $W_G^{l-1} = \{v \in V_G^{l-1} : \neg \mathit{HasLinks}(\{v\}, V_G^{l-1}, G)\}$ the set of vertices with no remaining links in V_G^{l-1}, and $W_H^{l-1} = \{v \in V_H^{l-1} : \neg \mathit{HasLinks}(\{v\}, V_H^{l-1}, H)\}$ the set of vertices with no remaining links in V_H^{l-1}. It is easy to see that $G_{W_G^{l-1} \cup V_G^l}$ and $H_{W_H^{l-1} \cup V_H^l}$ are isomorphic (adding the same number of isolated vertices to two isomorphic graphs yields two isomorphic graphs). Clearly, mapping m restricted to the vertices in $W_G^{l-1} \cup V_G^l$ and $W_H^{l-1} \cup V_H^l$ is an isomorphism of them, since it is when restricted to the vertices in V_G^l and V_H^l (from the induction hypothesis). Note that the vertices in W_G^{l-1} precede all the vertices in V_G^l in order $<_{\mathsf{Q_G}}$, and the vertices in W_H^{l-1} precede all the vertices in V_H^l in order $<_{\mathsf{Q_H}}$. Hence, m maps he vertices in W_G^{l-1} to the vertices in W_H^{l-1}.

In case partitions $\mathcal{S}^l$ and $\mathcal{T}^l$ are a vertex refinement of partitions $\mathcal{S}^{l-1}$ and $\mathcal{T}^{l-1}$ respectively, let v and w be the pivot vertices used in the refinement of partitions $\mathcal{S}^{l-1}$ and $\mathcal{T}^{l-1}$ respectively. Clearly, $v \in V_G^{l-1}$ but $v \notin V_G^l$, and $w \in V_H^{l-1}$ but $w \notin V_H^l$. Besides, since partitions $\mathcal{S}^{l-1}$ and $\mathcal{S}^l$ are compatible with $\mathcal{T}^{l-1}$ and $\mathcal{T}^l$ respectively, for all $x \in \{1, ..., r\}, \mathit{ADeg}(\{v\}, S_x^l, G) = \mathit{ADeg}(\{w\}, T_x^l, H)$. Hence, mapping m restricted to $\{v\} \cup V_G^l$ and $\{w\} \cup V_H^l$ is an isomorphism of $G_{\{v\} \cup V_G^l}$ and $H_{\{w\} \cup V_H^l}$. Note that v precedes all the vertices in V_G^l in order $<_{\mathsf{Q_G}}$, and w precedes all the vertices in V_H^l in order $<_{\mathsf{Q_H}}$. Hence, $m(v) = w$.

Consequently, if m restricted to the vertices in V_G^l and V_H^l is an isomorphism of $G_{V_G^l}$ and $H_{V_H^l}$, it must also be when extended to the vertices in V_G^{l-1} and V_H^{l-1}, thus proving our claim. ■

Theorem 3.1 *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.*

Proof: It follows directly from Lemmas 3.1 and 3.2. ■

Now, to prove that our algorithm correctly determines if two graphs G and H are isomorphic or not, it is enough to prove that it tests every possible sequence of partitions for one of the graphs against one sequence of partitions for the other graph. Thus, if it is not able to find a compatible one, that is because no such sequence of partitions exist.

Theorem 3.2 *Two graphs G and H are isomorphic if and only if $\mathit{AreIsomorphic}(G, H)$ returns* TRUE.

Proof: First, Algorithm *AreIsomorphic* tests some simple necessary conditions for isomorphism: both graphs must have the same number of vertices and the same number of arcs, and their degree partitions must be compatible; otherwise, the can not be isomorphic.

Then, a sequence of partitions is generated for each graph, and these sequences of partitions are searched for automorphisms. This has two effects: some backtracking points may be changed to simple VERTEX refinements, and equivalences among vertices are stored for use during the search for a sequence of partitions compatible with the target.

If a backtracking point is eliminated at level l, that is because all the vertices in the pivot cell are considered equivalent. This equivalence may have been established by three different means:

1. It may have been explicitly found by obtaining equivalent sequences of partitions, in which case, from Lemma 2.2, this equivalence holds.

2. It may have been found at some other level l' such that $l' > l$. In this case, from Lemma 2.3, it also holds at level l.

3. It may have been inferred applying Remark 2.1, in which case it also holds.

Since R^l is changed to VERTEX only at Line 26 in Algorithm 7, when all the vertices in the pivot cell are found equivalent according to the three criteria just mentioned, it is guaranteed that all the vertices in the pivot cell are certainly equivalent. Also, from Lemma 2.4, this equivalence must hold for the other graph, in case an equivalent sequence of partitions exists for that graph. Hence, eliminating this backtracking point is not an impediment for finding an equivalent sequence of partitions if it exists, and it will be enough to try one vertex in the pivot cell at this level. This argument may be applied to all the eliminated backtracking points.

Besides, in Algorithm 15, the orbit partition partition of graph H is used to prune the search at the remaining backtracking points. However, this equivalences among vertices are considered only in the case that all the vertices already discarded belonged to singleton orbits. From Observation 2.6, 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. Hence, there is an automorphism that permutes them and fixes all the vertices already discarded (since they belong to singleton orbits). Thus, if one of them did not lead to a compatible sequence of partitions, none of the members of its orbit will.

Consequently, the pruning achieved by Algorithms 7 and 15 do not eliminate paths in the search tree that might lead to an isomorphism of graphs G and H. Therefore, if Algorithm 15 returns FALSE, there is no sequence of partitions compatible with the target, and, from Lemma 3.1, graphs G and H are not isomorphic. If Algorithm 15 finds a sequence of partitions for one graph which is compatible with the one generated for the other graph, from Lemma 3.2, graphs G and H are isomorphic. Hence, Algorithm *AreIsomorphic* returns TRUE if and only if graphs G and H are isomorphic. ■

Chapter 4

Performance Evaluation

4.1 Graphs Benchmark

Benchmarking is crucial for practical graph isomorphism programs. Depending on the intended use of the algorithm, different families of graphs would be significant. In our case, since we want a general purpose algorithm, we want to test it with very different graph families. For this purpose, we have chosen the following classes of graphs which will be described in detail: random graphs, regular meshes, Point-Line graphs of Desarguesian projective planes, Strongly Regular Graphs, Component-Based Graphs, and Graphs based on Miyazaki's construction.

Moreover, other graph families have been considered such as point-line graphs of affine geometries and Kronecker Eye Flip graphs, from bliss benchmark [8].

4.1.1 Random Graphs

Random graphs are usually very simply tested for isomorphism. Yet, they are the most common graphs found in practice. For this reason, an algorithm that is relatively fast for difficult graphs but has bad performance with random graphs will not be practical. The random graphs included in our benchmark have been taken directly from [24]. They are graphs in which the arcs connect vertices without any structural regularity. The probability of an arc connecting two vertices is independent of the vertices. The generation of these graphs adopted the same model proposed in [31]. This model fixes the probability η of an arc connecting two distinct vertices. For our benchmark, only the graphs in [24] with $\eta = 0.1$ have been used, though there are other alternatives available in their database. These graphs have few, if any, automorphisms, and therefore, they are easy to test. Their vertices are very different from each other and easy to be differentiated.

Since all the graphs in [24] are directed, we have built another family of random graphs, obtained by simply converting these digraphs into undirected graphs. This allows to study changes in the behavior of the algorithms for these slight changes in their structure. In the benchmark, only pairs of isomorphic graphs will be included. It is easy to see that, with very high probability, a simple graph invariant like the degree sequence, will distinguish non-isomorphic random graphs. In [24], it is shown how vf2 is faster than nauty for these non-isomorphic random graphs.

4.1.2 Regular Meshes

Unlike random graphs, meshes do have a structure. This structure makes the graphs to have symmetries, and therefore automorphisms, what should make testing the isomorphism of these graphs harder. The graphs of this class include in the benchmark have also been taken form [24]. We have chosen to include in our benchmark only 2D-meshes, although there are also 3D- and 4D-meshes available, since their results would be similar (they have similar structure). These graphs are all square meshes like the 4×4 mesh showed in Figure 4.1.(a). The meshes of a given size included in the benchmark are all isomorphic. This means that the arcs are always directed rightwards and downwards as in the figure. These graphs have been shown to be very hard for nauty 2.0 [24]. The subsequent version 2.2 of nauty performs much better, though it is still quite slower than vf2.

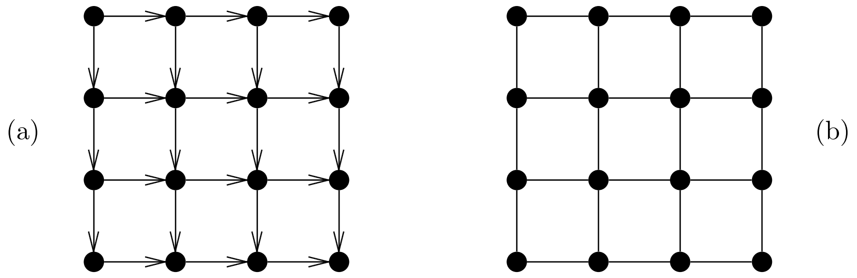

Figure 4.1: 2D-meshes.

Like in the previous case, the corresponding undirected graph family has been derived from the directed version, obtaining graphs like the one shown in Figure 4.1.(b).

4.1.3 Point-Line graphs of Desarguesian Projective Planes

Let $n = q^2 + q + 1$, and let π be a finite projective plane of order q with point set $P = \{p_1, ..., p_n\}$ and line set $L = \{l_1, ..., l_n\}$. A bipartite graph G with partitions (P, L) is said to be the *incidence point-line graph of the projective plane* π if for all $i, j \in \{1, ..., n\}$, $\{p_i, l_j\}$ is an edge of G if and only if $p_i \in l_j$. See for example the paper of Lazebnik and Thomason [13] for a method to generate the point-line incidence graphs of projective planes.

Point-line graphs of projective planes are known to be amongst the hardest graphs for isomorphism testing. For the generation of the graphs, we have used the point-line incidence matrices, provided by Gordon Moorhouse [20], of the Desarguesian projective planes. This gives rise to a family a very hard graphs. There may be more than one projective plane for a given size. However, they differ in basic parameters, like the degree, so they are easy to differentiate. Hence, we have only considered the Desarguesian projective planes, so we will only perform positive tests.

The structure of these graphs is better understood with an example. Figure 4.2 shows the point-line graph of the Desarguesian projective plane of order 2. Although this graph is bipartite, it has been drawn taking one vertex, and placing the rest of the vertices according to their distance to this vertex. This way, it is clear that the diameter of the graph is 3. In fact, the diameter remains constant for all the graphs in the family.

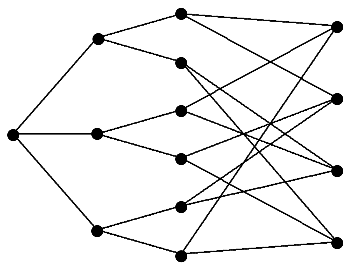

Figure 4.2: Point-line graph of the Desarguesian projective plane of order 2.

4.1.4 Strongly Regular Graphs

Strongly regular graphs (SRG) lie somewhere between highly structured and apparently random graphs. A *strongly regular graph* with parameters (n, k, λ, μ) is a regular graph of degree k on n vertices, such that each pair of adjacent vertices has λ common neighbors, and each pair of non-adjacent vertices has μ common neighbors.

In fact, though SRGs can be precisely characterized as a class of graphs, they can not be considered a proper family. Graphs of the same family should have more things in common. Strongly regular graphs can be further classified into several families, and even some of them might be impossible to fit into any family. Here, we will only take into consideration four families of strongly regular graphs: Paley graphs, triangular graphs, Latin Square graphs and lattice graphs.

Paley graphs

The strongly regular *Paley graph* $P(q)$ is a graph whose vertex set is the Finite Field of order q, $\mathbb{F}_q$, for q an odd prime power, $q \equiv 1 \pmod 4$, where two vertices are adjacent if and only if their difference is a nonzero square in $\mathbb{F}_q$. These SRGs have parameters $n = q$, $k = (q-1)/2$, $\lambda = (q-5)/4$, and $\mu = (q-1)/4$.

Paley graphs are not only vertex transitive, but also very regular with small automorphism groups. This makes it easy to compute their automorphism groups and canonical forms. However, since they do not have many automorphisms, they may be hard for direct backtracking algorithms.

Two subfamilies of Paley graphs have been distinguished: one contains the graphs generated for q prime, and the other contains the graphs generated for q a proper prime power. When q is prime, the corresponding graph has a smaller automorphism group than in the case q is a proper prime power. This is supposed to make computing the automorphism group, for the case where q is prime, faster.

For the case of q prime, an ad hoc program has been used to generate the graphs in our benchmark, while the graphs for the case of q a proper prime power have been generated with the aid of the GAP package [7] with Grape [26], which provide a very helpful tool to generate Finite Fields, and to operate with them. Then, random permutations have been generated for each graph size. Since there is only one Paley graph with certain parameters, only positive tests will be performed on this family of graphs.

Triangular Graphs

Let S_q be a set of cardinality $q \geq 5$, then the vertex set of the *triangular graph* $T(q)$ is the set of 2-element subsets of S_q, which contains $q(q-1)/2$ vertices. In $T(q)$ two vertices are adjacent if and only if they are not disjoint sets. The triangular graphs have parameters $n = q(q-1)/2$, $k = 2(q-2)$, $\lambda = q-2$, and $\mu = 4$.

These graphs are vertex transitive and have large automorphism groups. Therefore, for direct backtracking algorithms, isomorphism would be easier to test than non-isomorphism since it only needs to find the first isomorphism. However, for algorithms that compute the whole automorphism group of the graphs, triangular graphs should be harder than Paley graphs, though this may be mitigated by an efficient way to discover and make use of automorphisms.

There is only one triangular graph for each value of q, and no other SRG has the same parameters as a triangular graph, except for $q = 8$ (cf. [2]). Therefore, only positive tests will be considered for this family of graphs.

Latin Square Graphs

The family of Latin square graphs is generated from Latin squares. A *Latin square* of order n, $n \geq 2$, is an $n \times n$ matrix with n different symbols, where each symbol occurs once per row and column of the matrix.

Let $L_1 = (a_{ij})$ and $L_2 = (b_{ij})$ be two Latin squares with n symbols, $n \geq 2$. L_1 and L_2 are *orthogonal* if and only if every ordered pair of symbols occurs exactly once among the n^2 pairs (a_{ij}, b_{ij}), $i, j \in \{1, ..., n\}$. A set of Latin squares of order n where each pair of Latin squares are orthogonal is called a *set of mutually orthogonal Latin squares (MOLS).*

From each set of MOLS of order n, $n \geq 2$, a strongly regular graph can be generated in the following way: the vertices of the graph are the n^2 items of a Latin square of order n, and two vertices are adjacent if and only if the items are in the same row, in the same column, or they have the same symbol in one of the orthogonal Latin squares.

A *Latin square graph* is built using the construction above from a set of $g-2$ MOLS of order m, $m \geq g \geq 2$, and denoted $L_g(m)$. It is a strongly regular graph with parameters $n = m^2$, $k = g(m-1)$, $\lambda = m - 2 + (g-1)(g-2)$, and $\mu = g(g-1)$.

Since there are $m-1$ MOLS of order m, it is possible to generate $\binom{m-1}{g-2}$ combinations of Latin squares that yield $\binom{m-1}{g-2}$ Latin square graphs, some of which may be isomorphic. Remember that this holds for any g, $m \geq g \geq 2$. For a fixed g, the generated graphs have the same parameters and are potentially non-isomorphic, what allows the generation of negative tests. These graphs have a large automorphism group but at the same time, they are not so regular as the Paley or triangular graphs, what makes them harder examples of strongly regular graphs. The existence of non-isomorphic Latin square graphs with the same parameters suggests difficulty. In fact, for a long time, they have been considered hard instances for graph isomorphism (cf. [27]).

For our benchmark, we have included different permutations of Latin square graphs $L_3(5)$, $L_4(7)$, $L_5(9)$, $L_6(11)$, $L_7(13)$, $L_9(17)$, $L_{10}(19)$, $L_{12}(23)$, $L_{13}(25)$, $L_{14}(27)$, $L_{15}(29)$, and $L_{16}(31)$. Other combinations of m and g were also possible, and also different values of g for a fixed m could have been considered. However, we believe that the graphs included are significant enough for our purpose.

The generation of these graphs has been performed with the aid of GAP [7] and GUAVA [4], that has a function that generates the sets of MOLS required to build the graphs. For each set of parameters g and m, several graphs have been generated. Then, they were tested for isomorphism in order to discard redundant instances. Permutations of the resulting graphs were generated in order to obtain one hundred pairs of each size. Thus we can study the behavior of the algorithms with different permutations of the same graph, and with different graphs with the same parameters.

Lattice Graphs

A lattice graph is a graph whose vertices are the elements of an $m \times m$ square and two vertices are adjacent if and only if they are in the same row or in the same column. This graph may be seen as a Latin square graph $L_2(m)$ –for $g = 2$, there are no MOLS to consider–, and then, its parameters will be $n = m^2$, $k = 2(n-1)$, $\lambda = n-2$, and $\mu = 2$.

Lattice graphs are determined by their parameters. Except for $n = 4$, they are the unique strongly regular graphs with these parameters. This, along with their high regularity suggests that they may be quite simpler than proper Latin square graphs for direct backtracking algorithms, while their large automorphism group can make them still hard for algorithms that compute canonical labelings. Since no other SRG has the same parameters as a given lattice graph, only positive isomorphism tests will be performed for this family of graphs.

Steiner Triple Systems

These are the line graphs $srg(v(v-1)/6, 3(v-3)/2, (v+3)/2, 9)$ of Steiner triple systems which have $(v(v+1)-2)/18$ orbits.

4.1.5 Component-Based Graphs

Unions of Tripartite Graphs

This is a family of graphs built from small graph pieces. We have designed two small tripartite graphs that are very similar, yet non isomorphic. Their directed versions are shown in Figure 4.3. Their undirected versions are straightforward.

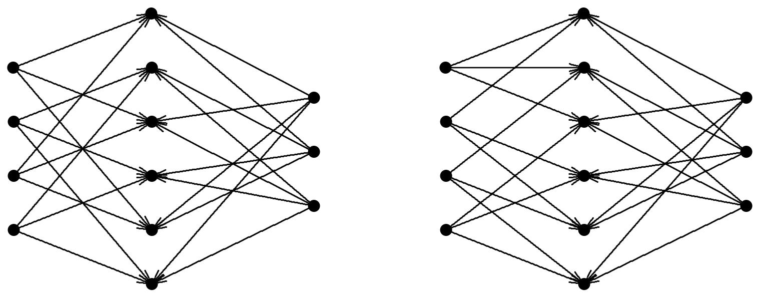

Figure 4.3: Tripartite graphs.

Our original idea was to generate graphs that were disjoint unions of several copies of these graphs. Since vf2 cannot deal with disconnected graphs, we considered computing the inverses of the graphs, and using these connected graphs for the tests. However, we decided to modify the generation process, so that instead of the disjoint union, we would connect each vertex in a connected component to every other vertex in the graph. This also produces a connected graph.

We have generated graphs with a different but similar number of copies of each component. Therefore, the graphs will be very similar. They have the same number of vertices, the same number of arcs, and the same sequence of vertex degrees. We expect these graphs to be hard, both for direct backtracking algorithms, and for those which compute canonical forms of the graphs since they have many automorphisms, but these come from structurally different components.

Directed and undirected versions of the graphs have been generated, and pairs of isomorphic and non isomorphic graphs will be used for the tests. Here, we expect the negative tests to be especially hard for direct backtracking algorithms.

Unions of Strongly Regular Graphs

Unions of strongly regular graphs with the same parameters are known to be good candidates to force nauty to exponential time. They are also likely to have the same effect on direct backtracking algorithms. Hence, we should test our algorithm with this kind of graphs and see if it suffered from the same disease. Since any set of strongly regular graphs would do the job, we just chose some $(29, 14, 6, 7)$ strongly regular graphs that are small enough to allow building graphs of the required sizes and there are enough to build sufficiently large graphs for the tests. These graphs were provided by Sven Reichard [23].

The unions of these strongly regular graphs have been accomplished in the same way as in the case above. Also, several permutations of different graphs have been generated for each size, so that both positive and negative tests may be performed. Again, for direct backtracking algorithms, the negative cases are expected to be much harder than the positive ones.

Cubic Hypo-Hamiltonian clique-connected

This family is built using as basic components two non-isomorphic cubic Hypo-Hamiltonian graphs with 22 vertices. 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.

4.1.6 Graphs based on Miyazaki's Construction

Another family of graphs included in our benchmark is the Miyazaki's Fürer gadgets. The graphs of this class have been generated with a program provided by Takunari Miyazaki, who showed in [19] that nauty needed exponential time to compute canonical forms for these graphs. In fact, since we do not consider colored graphs, and we do not force certain orderings of the vertices that could make the graphs harder, we will not be able to force nauty to require exponential time with all the instances. Our graphs can be assimilated to what Miyazaki calls his type-C family (with vertices randomly ordered). However, we will include also a directed version of these graphs, which are likely to force nauty to require exponential time.

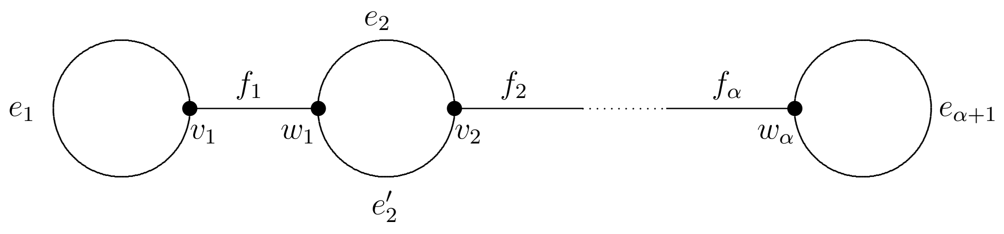

Figure 4.4: The graph Y_α.

The graphs are built in the following way: first, consider the undirected multigraph Y_α shown in Figure 4.4 with vertex set $V(Y_\alpha) = \{v_1, ..., v_\alpha, w_1, ..., w_\alpha\}$ and edge set $E(Y_\alpha) = \{E_1 \cup E_2 \cup E_3\}$ where:

(i) $E_1 = \{e_1, e_{\alpha+1} : e_1 = (v_1, v_1), e_{\alpha+1} = (w_\alpha, w_\alpha)\}$,
(ii) $E_2 = \{e_i, e'_i : e_i = e'_i = (w_{i-1}, v_i), 2 \leq i \leq \alpha\}, and$
(iii) $E_3 = \{f_i : f_i = (v_i, w_i), 1 \leq i \leq \alpha\}$.

Each node in Y_α has an incident cycle (one or two e-edges) and an incident bridge (edge f). Then, applying FÃ$\frac{1}{4}$rer's construction (cf. [34]), we obtain a new (simple, not multi) graph, in which each vertex in the multigraph Y_α is substituted by a FÃ$\frac{1}{4}$rer gadget. Thus, we obtain a 3-regular graph. Figure 4.5 shows the resulting graph, and its directed version.

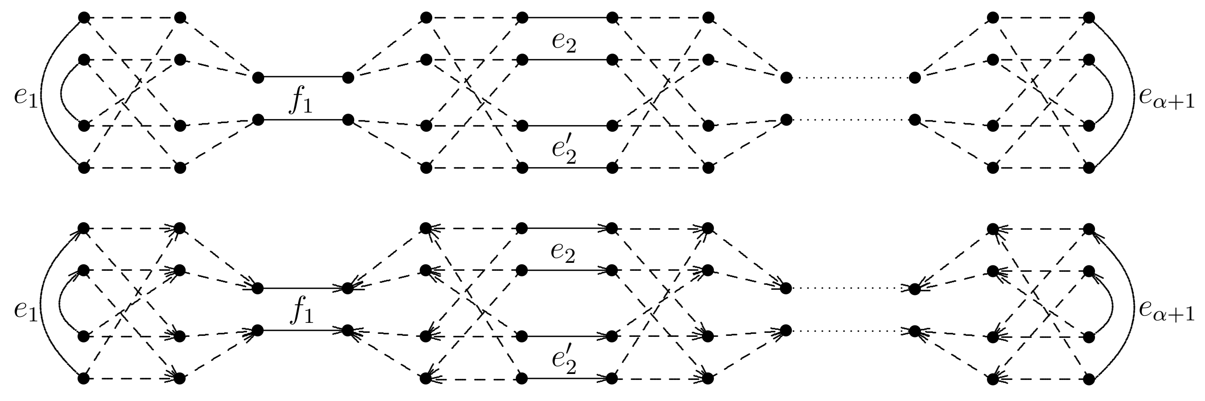

Figure 4.5: Miyazaki's graphs.

These are bounded valence graphs and, therefore, their canonical forms can be computed in polynomial time using the method of [16], what yields a polynomial time isomorphism test. However, as shown by Miyazaki, nauty may require exponential time to compute their canonical forms.

In our benchmark, different permutations of the graphs are used for each graph size. To provide for non-isomorphism tests, we generate graphs where one random bridge is changed for a switch. This yields, graphs that are very similar to the original ones, but not isomorphic. Finding this subtle difference should be hard for direct backtracking algorithms, but they are also hard for nauty (cf. [19]). Examples of such graphs with twenty vertices are shown in Figure 4.6.

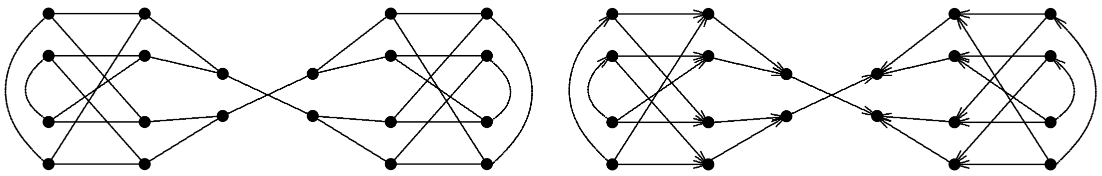
Figure 4.6: Miyazaki's switch graphs.

4.2 Example: Automorphism Group Computation

In this section we will show how the automorphism group computation of our algorithm works, for the graph of Figure 4.7. This is an undirected instance of a simple union of two non-isomorphic tripartite graphs (Figure 4.3).

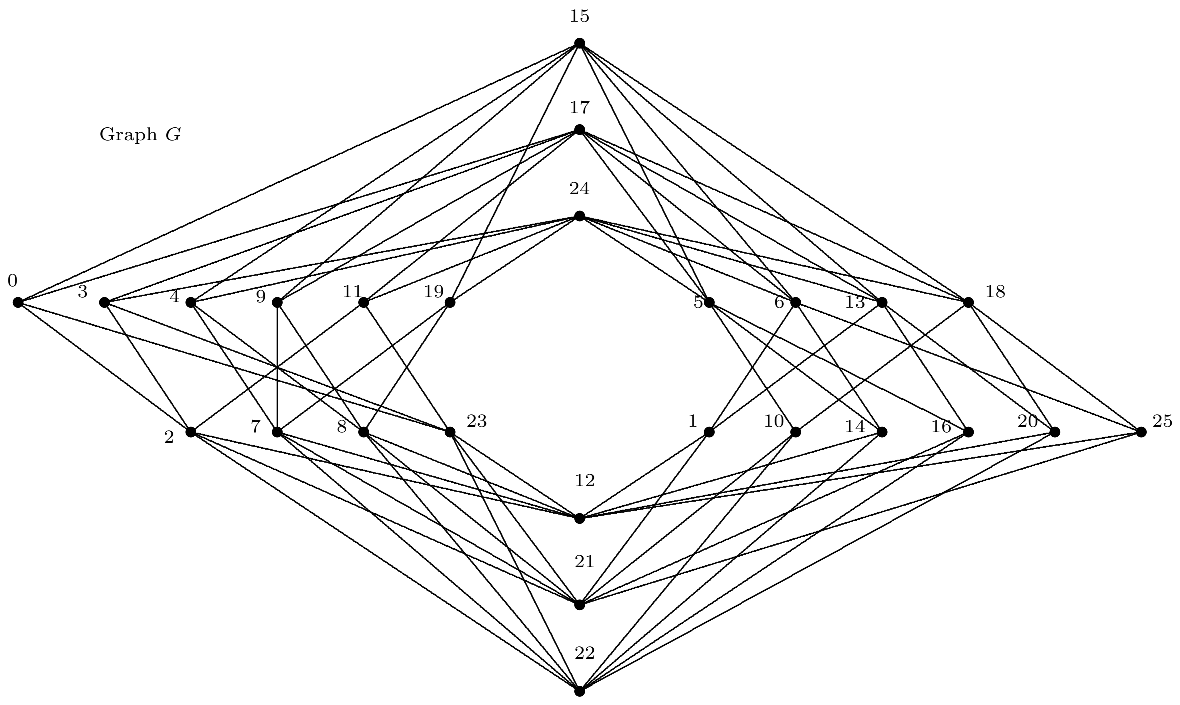

Figure 4.7: Sample graph for automorphism group computation.

First of all, the algorithm generates a sequence of partitions for the graph G. It starts computing the degree partition $\mathcal{D}$, and we get:

Graph G: $\mathcal{D} = (D_1, D_2, D_3)$

$D_1 = \{12, 15, 17, 21, 22, 24\}$	with $Deg(D_1, G) = (8, 0, 0)$
$D_2 = \{2, 5, 6, 7, 8, 13, 18, 23\}$	with $Deg(D_2, G) = (6, 0, 0)$
$D_3 = \{0, 1, 3, 4, 9, 10, 11, 14, 16, 19, 20, 25\}$	with $Deg(D_3, G) = (4, 0, 0)$

Once the degree partitions have been obtained a sequence of partitions will be generated for graph G, applying algorithm *GenerateSequenceOfPartitions*.

4.2.1 Generation of a Sequence of Partitions

Initially, algorithm *GenerateSequenceOfPartitions* sets $\mathcal{S}^0 = \mathcal{D} = (S_1^0, S_2^0, S_3^0) = (\{12, 15, 17, 21, 22, 24\}, \{2, 5, 6, 7, 8, 13, 18, 23\}, \{0, 1, 3, 4, 9, 10, 11, 14, 16, 19, 20, 25\})$, and marks the tree cells as valid. Next, the three cells are tried for a set refinements in order 1, 2 and 3 (according to algorithm *IndexBestValidPivot*), but none of them succeeds in splitting any cell and so that, they are marked as non valid. Then, each cell *type* according to algorithm *IndexBestIndividualizedCell* is attempt to be individualized (recall that only the first vertex of each cell is considered). The result of these individualization is performed by algorithm *NextEquitablePartition* and the results are as follows (parameter n is set to 0 initially):

- Partition $\mathcal{T}$ as result using cell 1 (vertex 12, level 8 reached):

  $$\begin{array}{ll}
  T_1^8 = \{15, 17, 24\} & \text{with } ADeg(T_1^8, W, G) = (6, 0, 0) \\
  T_2^8 = \{2, 7, 8, 23\} & \text{with } ADeg(T_2^8, W, G) = (3, 0, 0) \\
  T_3^8 = \{13, 18\} & \text{with } ADeg(T_3^8, W, G) = (5, 0, 0) \\
  T_4^8 = \{14\} & \text{with } ADeg(T_4^8, W, G) = (0, 0, 0) \\
  T_5^8 = \{1, 25\} & \text{with } ADeg(T_5^8, W, G) = (1, 0, 0) \\
  T_6^8 = \{10, 16\} & \text{with } ADeg(T_6^8, W, G) = (1, 0, 0) \\
  T_7^8 = \{0, 3, 4, 9, 11, 19\} & \text{with } ADeg(T_7^8, W, G) = (4, 0, 0)
  \end{array}$$

  Since the result partition $\mathcal{T}$ is not last partition, and it is not subpartition of $\mathcal{S}^0$, the normal criteria is applied to continue the attempting with next cell. The sum of the number of cells of partition $\mathcal{T}$ and the discarded vertices $(7+6)$ is greater than $n = 0$, so that b (best cell) is set to 1 and n is set to $7 + 6 = 13$.

- Partition $\mathcal{T}$ as result using cell 2 (vertex 2, level 8 reached):

  $$\begin{array}{ll}
  T_1^8 = \{12, 21, 22\} & \text{with } ADeg(T_1^8, W, G) = (6, 0, 0) \\
  T_2^8 = \{5, 6, 13, 18\} & \text{with } ADeg(T_2^8, W, G) = (3, 0, 0) \\
  T_3^8 = \{7, 8\} & \text{with } ADeg(T_3^8, W, G) = (5, 0, 0) \\
  T_4^8 = \{4, 19\} & \text{with } ADeg(T_4^8, W, G) = (2, 0, 0) \\
  T_5^8 = \{1, 10, 14, 16, 20, 25\} & \text{with } ADeg(T_5^8, W, G) = (4, 0, 0)
  \end{array}$$

  Since the result partition $\mathcal{T}$ is not last partition, and it is not subpartition of $\mathcal{S}^0$, the normal criteria is applied to continue the attempting with next cell. The sum of the number of cells of partition $\mathcal{T}$ and the discarded vertices $(5 + 9)$ is greater than $n = 13$, so that b (best cell) is set to 2 and n is set to $5 + 9 = 14$.

- Partition $\mathcal{T}$ as result using cell 3 (vertex 0, level 8 reached):

  $$\begin{array}{ll}
  T_1^8 = \{12, 21, 22\} & \text{with } ADeg(T_1^8, W, G) = (8, 0, 0) \\
  T_2^8 = \{2, 23\} & \text{with } ADeg(T_2^8, W, G) = (5, 0, 0) \\
  T_3^8 = \{5, 6, 13, 18\} & \text{with } ADeg(T_3^8, W, G) = (5, 0, 0) \\
  T_4^8 = \{7, 8\} & \text{with } ADeg(T_4^8, W, G) = (5, 0, 0) \\
  T_5^8 = \{3, 11\} & \text{with } ADeg(T_5^8, W, G) = (2, 0, 0) \\
  T_6^8 = \{4, 19\} & \text{with } ADeg(T_6^8, W, G) = (2, 0, 0) \\
  T_7^8 = \{1, 10, 14, 16, 20, 25\} & \text{with } ADeg(T_7^8, W, G) = (4, 0, 0)
  \end{array}$$

  Since the result partition $\mathcal{T}$ is not last partition, and it is not subpartition of $\mathcal{S}^0$, the normal criteria is applied to continue the attempting with next cell. The sum of the number of cells of partition $\mathcal{T}$ and the discarded vertices $(7+5)$ is not greater than $n = 14$, so that b is not modified.

Cell S_2^0 is chosen as the pivot set ($P^0 = 2$), setting $R^0 = \mathit{BACKTRACK}$ and a vertex refinement is performed, obtaining a new partition $\mathcal{S}^1 = (S_1^1, S_2^1, S_3^1, S_4^1, S_5^1)$, where $V^1 = \{0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25\}$, and:

$S_1^1 = \{12,21,22\}$	with $\mathit{ADeg}(S_1^1, \{2\}, G) = (1,0,0)$	and $\mathit{Valid}(S_1^1) = \mathit{TRUE}$
$S_2^1 = \{15,17,24\}$	with $\mathit{ADeg}(S_2^1, \{2\}, G) = (0,0,0)$	and $\mathit{Valid}(S_2^1) = \mathit{TRUE}$
$S_3^1 = \{5,6,7,8,13,18,23\}$	with $\mathit{ADeg}(S_3^1, \{2\}, G) = (0,0,0)$	and $\mathit{Valid}(S_3^1) = \mathit{TRUE}$
$S_4^1 = \{0,3,11\}$	with $\mathit{ADeg}(S_4^1, \{2\}, G) = (1,0,0)$	and $\mathit{Valid}(S_4^1) = \mathit{TRUE}$
$S_5^1 = \{1,4,9,10,14,16,19,20,25\}$	with $\mathit{ADeg}(S_5^1, \{2\}, G) = (0,0,0)$	and $\mathit{Valid}(S_5^1) = \mathit{TRUE}$

All cells of $\mathcal{S}^1$ are valid since they are new. Therefore, they must be tried for a set refinement. Algorithm *IndexBestValidPivot* selects them in increasing order size, preferring a greater available degree for those of equal size. Hence, it chooses S_2^1 ($P^1 = 2$) as the pivot set. Applying a set refinement to $\mathcal{S}^1$, we obtain a new partition $\mathcal{S}^2 = (S_1^2, S_2^2, S_3^2, S_4^2, S_5^2, S_6^2, S_7^2)$, where $V^2 = \{0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25\}$, $R^1 = \mathit{SET}$, and:

$S_1^2 = \{12,21,22\}$	with $\mathit{ADeg}(S_1^2, \{15,17,24\}, G) = (0,0,0)$	and $\mathit{Valid}(S_1^2) = \mathit{TRUE}$
$S_2^2 = \{15,17,24\}$	with $\mathit{ADeg}(S_2^2, \{15,17,24\}, G) = (0,0,0)$	and $\mathit{Valid}(S_2^2) = \mathit{FALSE}$
$S_3^2 = \{5,6,13,18\}$	with $\mathit{ADeg}(S_3^2, \{15,17,24\}, G) = (3,0,0)$	and $\mathit{Valid}(S_3^2) = \mathit{TRUE}$
$S_4^2 = \{7,8,23\}$	with $\mathit{ADeg}(S_4^2, \{15,17,24\}, G) = (0,0,0)$	and $\mathit{Valid}(S_4^2) = \mathit{TRUE}$
$S_5^2 = \{0,3,11\}$	with $\mathit{ADeg}(S_5^2, \{15,17,24\}, G) = (2,0,0)$	and $\mathit{Valid}(S_5^2) = \mathit{TRUE}$
$S_6^2 = \{4,9,19\}$	with $\mathit{ADeg}(S_6^2, \{15,17,24\}, G) = (2,0,0)$	and $\mathit{Valid}(S_6^2) = \mathit{TRUE}$
$S_7^2 = \{1,10,14,16,20,25\}$	with $\mathit{ADeg}(S_7^2, \{15,17,24\}, G) = (0,0,0)$	and $\mathit{Valid}(S_7^2) = \mathit{TRUE}$

Again, since there are valid cells but no singleton ones, algorithm *IndexBestValidPivot* chooses cell S_1^2 and S_4^2 without success for a set refinement. Finally cell S_6^2 is chosen as a pivot set ($P^2 = 6$), and this yields a new partition $\mathcal{S}^3 = (S_1^3, S_2^3, S_3^3, S_4^3, S_5^3, S_6^3, S_7^3, S_8^3, S_9^3, S_{10}^3)$, where $V^3 = \{0,1,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25\}$, $R^2 = \mathit{SET}$, and:

$S_1^3 = \{12,21,22\}$	with $\mathit{ADeg}(S_1^3, \{4,9,19\}, G) = (0,0,0)$	and $\mathit{Valid}(S_1^3) = \mathit{FALSE}$
$S_2^3 = \{15\}$	with $\mathit{ADeg}(S_2^3, \{4,9,19\}, G) = (3,0,0)$	and $\mathit{Valid}(S_2^3) = \mathit{TRUE}$
$S_3^3 = \{24\}$	with $\mathit{ADeg}(S_3^3, \{4,9,19\}, G) = (2,0,0)$	and $\mathit{Valid}(S_3^3) = \mathit{TRUE}$
$S_4^3 = \{17\}$	with $\mathit{ADeg}(S_4^3, \{4,9,19\}, G) = (1,0,0)$	and $\mathit{Valid}(S_4^3) = \mathit{TRUE}$
$S_5^3 = \{5,6,13,18\}$	with $\mathit{ADeg}(S_5^3, \{4,9,19\}, G) = (0,0,0)$	and $\mathit{Valid}(S_5^3) = \mathit{TRUE}$
$S_6^3 = \{7,8\}$	with $\mathit{ADeg}(S_6^3, \{4,9,19\}, G) = (3,0,0)$	and $\mathit{Valid}(S_6^3) = \mathit{TRUE}$
$S_7^3 = \{23\}$	with $\mathit{ADeg}(S_7^3, \{4,9,19\}, G) = (0,0,0)$	and $\mathit{Valid}(S_7^3) = \mathit{TRUE}$
$S_8^3 = \{0,3,11\}$	with $\mathit{ADeg}(S_8^3, \{4,9,19\}, G) = (3,0,0)$	and $\mathit{Valid}(S_8^3) = \mathit{TRUE}$
$S_9^3 = \{4,9,19\}$	with $\mathit{ADeg}(S_9^3, \{4,9,19\}, G) = (0,0,0)$	and $\mathit{Valid}(S_9^3) = \mathit{FALSE}$
$S_{10}^3 = \{1,10,14,16,20,25\}$	with $\mathit{ADeg}(S_{10}^3, \{4,9,19\}, G) = (0,0,0)$	and $\mathit{Valid}(S_{10}^3) = \mathit{TRUE}$

Having a valid singleton cell S_2^3, it will be selected as the pivot set ($P^3 = 2$) for a vertex refinement, what yields partition $\mathcal{S}^4 = (S_1^4, S_2^4, S_3^4, S_4^4, S_5^4, S_6^4, S_7^4, S_8^4, S_9^4, S_{10}^4)$, where $V^4 = \{0,1,3,4,5,6,7,8,9,10,11,12,13,14,16,17,18,19,20,21,22,23,24,25\}$, $R^3 = \mathit{VERTEX}$, and:

$S^4_1 = \{12, 21, 22\}$	with $ADeg(S^4_1, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_1) = FALSE$
$S^4_2 = \{24\}$	with $ADeg(S^4_2, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_2) = TRUE$
$S^4_3 = \{17\}$	with $ADeg(S^4_3, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_3) = TRUE$
$S^4_4 = \{5, 6, 13, 18\}$	with $ADeg(S^4_4, \{15\}, G) = (1, 0, 0)$	and $Valid(S^4_4) = TRUE$
$S^4_5 = \{7, 8\}$	with $ADeg(S^4_5, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_5) = TRUE$
$S^4_6 = \{23\}$	with $ADeg(S^4_6, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_6) = TRUE$
$S^4_7 = \{0\}$	with $ADeg(S^4_7, \{15\}, G) = (1, 0, 0)$	and $Valid(S^4_7) = TRUE$
$S^4_8 = \{3, 11\}$	with $ADeg(S^4_8, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_8) = TRUE$
$S^4_9 = \{4, 9, 19\}$	with $ADeg(S^4_9, \{15\}, G) = (1, 0, 0)$	and $Valid(S^4_9) = FALSE$
$S^4_{10} = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S^4_{10}, \{15\}, G) = (0, 0, 0)$	and $Valid(S^4_{10}) = TRUE$

Again, having a valid singleton cell S^4_2, it will be selected as the pivot set ($P^4 = 2$) for a vertex refinement, what yields partition $\mathcal{S}^5 = (S^5_1, S^5_2, S^5_3, S^5_4, S^5_5, S^5_6, S^5_7, S^5_8, S^5_9, S^5_{10})$, where $V^5 = \{0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 25\}$, $R^4 = VERTEX$, and:

$S^5_1 = \{12, 21, 22\}$	with $ADeg(S^5_1, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_1) = FALSE$
$S^5_2 = \{17\}$	with $ADeg(S^5_2, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_2) = TRUE$
$S^5_3 = \{5, 6, 13, 18\}$	with $ADeg(S^5_3, \{24\}, G) = (1, 0, 0)$	and $Valid(S^5_3) = TRUE$
$S^5_4 = \{7, 8\}$	with $ADeg(S^5_4, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_4) = TRUE$
$S^5_5 = \{23\}$	with $ADeg(S^5_5, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_5) = TRUE$
$S^5_6 = \{0\}$	with $ADeg(S^5_6, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_6) = TRUE$
$S^5_7 = \{3, 11\}$	with $ADeg(S^5_7, \{24\}, G) = (1, 0, 0)$	and $Valid(S^5_7) = TRUE$
$S^5_8 = \{4, 19\}$	with $ADeg(S^5_8, \{24\}, G) = (1, 0, 0)$	and $Valid(S^5_8) = TRUE$
$S^5_9 = \{9\}$	with $ADeg(S^5_9, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_9) = TRUE$
$S^5_{10} = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S^5_{10}, \{24\}, G) = (0, 0, 0)$	and $Valid(S^5_{10}) = TRUE$

The same process it is done while having a valid singleton cell S^5_2, it will be selected as the pivot set ($P^5 = 2$) for a vertex refinement, what yields partition $\mathcal{S}^6 = (S^6_1, S^6_2, S^6_3, S^6_4, S^6_5, S^6_6, S^6_7, S^6_8, S^6_9)$, where $V^6 = \{0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 23, 25\}$, $R^5 = VERTEX$, and:

$S^6_1 = \{12, 21, 22\}$	with $ADeg(S^6_1, \{17\}, G) = (0, 0, 0)$	and $Valid(S^6_1) = FALSE$
$S^6_2 = \{5, 6, 13, 18\}$	with $ADeg(S^6_2, \{17\}, G) = (1, 0, 0)$	and $Valid(S^6_2) = TRUE$
$S^6_3 = \{7, 8\}$	with $ADeg(S^6_3, \{17\}, G) = (0, 0, 0)$	and $Valid(S^6_3) = TRUE$
$S^6_4 = \{23\}$	with $ADeg(S^6_4, \{17\}, G) = (0, 0, 0)$	and $Valid(S^6_4) = TRUE$
$S^6_5 = \{0\}$	with $ADeg(S^6_5, \{17\}, G) = (1, 0, 0)$	and $Valid(S^6_5) = TRUE$
$S^6_6 = \{3, 11\}$	with $ADeg(S^6_6, \{17\}, G) = (1, 0, 0)$	and $Valid(S^6_6) = TRUE$
$S^6_7 = \{4, 19\}$	with $ADeg(S^6_7, \{17\}, G) = (0, 0, 0)$	and $Valid(S^6_7) = TRUE$
$S^6_8 = \{9\}$	with $ADeg(S^6_8, \{17\}, G) = (1, 0, 0)$	and $Valid(S^6_8) = TRUE$
$S^6_9 = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S^6_9, \{17\}, G) = (0, 0, 0)$	and $Valid(S^6_9) = TRUE$

The cell S^6_4 is selected as the pivot set ($P^6 = 4$) for a vertex refinement, what yields partition $\mathcal{S}^7 = (S^7_1, S^7_2, S^7_3, S^7_4, S^7_5, S^7_6, S^7_7, S^7_8)$, where $V^7 = \{0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 25\}$, $R^6 = VERTEX$, and:

$S_1^7 = \{12, 21, 22\}$	with $ADeg(S_1^7, \{23\}, G) = (1, 0, 0)$	and $Valid(S_1^7) = FALSE$
$S_2^7 = \{5, 6, 13, 18\}$	with $ADeg(S_2^7, \{23\}, G) = (0, 0, 0)$	and $Valid(S_2^7) = TRUE$
$S_3^7 = \{7, 8\}$	with $ADeg(S_3^7, \{23\}, G) = (0, 0, 0)$	and $Valid(S_3^7) = TRUE$
$S_4^7 = \{0\}$	with $ADeg(S_4^7, \{23\}, G) = (1, 0, 0)$	and $Valid(S_4^7) = FALSE$
$S_5^7 = \{3, 11\}$	with $ADeg(S_5^7, \{23\}, G) = (1, 0, 0)$	and $Valid(S_5^7) = FALSE$
$S_6^7 = \{4, 19\}$	with $ADeg(S_6^7, \{23\}, G) = (0, 0, 0)$	and $Valid(S_6^7) = TRUE$
$S_7^7 = \{9\}$	with $ADeg(S_7^7, \{23\}, G) = (0, 0, 0)$	and $Valid(S_7^7) = TRUE$
$S_8^7 = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S_8^7, \{23\}, G) = (0, 0, 0)$	and $Valid(S_8^7) = TRUE$

The cell S_7^7 is selected as the pivot set ($P^7 = 7$) for a vertex refinement, what yields partition $\mathcal{S}^8 = (S_1^8, S_2^8, S_3^8, S_4^8, S_5^8)$, where $V^8 = \{1, 4, 5, 6, 7, 8, 10, 12, 13, 14, 16, 18, 19, 20, 21, 21, 25\}$, $R^7 = VERTEX$, and:

$S_1^8 = \{12, 21, 22\}$	with $ADeg(S_1^8, \{9\}, G) = (0, 0, 0)$	and $Valid(S_1^8) = FALSE$
$S_2^8 = \{5, 6, 13, 18\}$	with $ADeg(S_2^8, \{9\}, G) = (0, 0, 0)$	and $Valid(S_2^8) = FALSE$
$S_3^8 = \{7, 8\}$	with $ADeg(S_3^8, \{9\}, G) = (1, 0, 0)$	and $Valid(S_3^8) = FALSE$
$S_4^8 = \{4, 19\}$	with $ADeg(S_6^8, \{9\}, G) = (0, 0, 0)$	and $Valid(S_6^8) = FALSE$
$S_5^8 = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S_8^8, \{9\}, G) = (0, 0, 0)$	and $Valid(S_8^8) = FALSE$

Note that not only the pivot vertex was discarded but also cells S_4^7 and S_5^7, because they do not have links. At this level, it is known, from the choosing of pivot cell at level 0, that level 8 is an equitable level (because it is not last level). So that, every cell is marked as non valid and the algorithm *IndexBestIndividualizedCell* selects the optimal cell for individualization as follows (parameter $n = 0$):

- Partition $\mathcal{T}$ as result using cell 3 (vertex 7, level 10 reached):

$T_1^{10} = \{12, 21, 22\}$	with $ADeg(T_1^{10}, W, G) = (4, 0, 0)$
$T_2^{10} = \{5, 6, 13, 18\}$	with $ADeg(T_2^{10}, W, G) = (3, 0, 0)$
$T_3^{10} = \{4, 19\}$	with $ADeg(T_3^{10}, W, G) = (0, 0, 0)$
$T_4^{10} = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(T_4^{10}, W, G) = (4, 0, 0)$

  Since the result partition $\mathcal{T}$ is subpartition of $\mathcal{S}^8$, we not attempt to individualize any other cell and this cell is chosen as pivot set.

The individualization is carried out with pivot set S_3^8 ($P^8 = 3$), $R^8 = BACKTRACK$ and the new partition $\mathcal{S}^9 = (S_1^9, S_2^9, S_3^9, S_4^9, S_5^9)$ is obtained, where $V^9 = \{1, 4, 5, 6, 8, 10, 12, 13, 14, 16, 18, 19, 20, 21, 22, 25\}$, and:

$S_1^9 = \{12, 21, 22\}$	with $ADeg(S_1^9, \{7\}, G) = (1, 0, 0)$	and $Valid(S_1^9) = FALSE$
$S_2^9 = \{5, 6, 13, 18\}$	with $ADeg(S_2^9, \{7\}, G) = (0, 0, 0)$	and $Valid(S_2^9) = FALSE$
$S_3^9 = \{8\}$	with $ADeg(S_3^9, \{7\}, G) = (0, 0, 0)$	and $Valid(S_3^9) = TRUE$
$S_4^9 = \{4, 19\}$	with $ADeg(S_4^9, \{7\}, G) = (1, 0, 0)$	and $Valid(S_4^9) = FALSE$
$S_5^9 = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(S_5^9, \{7\}, G) = (0, 0, 0)$	and $Valid(S_5^9) = FALSE$

The only singleton cell S_3^9 is chosen as the pivot set ($P^9 = 3$) for a vertex refinement, what yields partition $\mathcal{S}^{10} = (S_1^{10}, S_2^{10}, S_3^{10}, S_4^{10}, S_5^{10}, S_6^{10}, S_7^{10}, S_8^{10})$, where $V^{10} = \{1, 4, 5, 6, 10, 12, 13, 14, 16, 18, 19, 20, 21, 22, 25\}$, $R^9 = VERTEX$, and:

$S_1^{10} = \{12, 21, 22\}$ with $ADeg(S_1^{10}, \{8\}, G) = (1, 0, 0)$ and $Valid(S_1^{10}) = FALSE$
$S_2^{10} = \{5, 6, 13, 18\}$ with $ADeg(S_2^{10}, \{8\}, G) = (0, 0, 0)$ and $Valid(S_2^{10}) = FALSE$
$S_3^{10} = \{4, 19\}$ with $ADeg(S_3^{10}, \{8\}, G) = (1, 0, 0)$ and $Valid(S_3^{10}) = FALSE$
$S_4^{10} = \{1, 10, 14, 16, 20, 25\}$ with $ADeg(S_4^{10}, \{8\}, G) = (0, 0, 0)$ and $Valid(S_4^{10}) = FALSE$

At this level, it is known, from the choosing of pivot cell at level 8, that level 10 is an equitable level (because it is not last level). So that, every cell is marked as non valid and the algorithm *IndexBestIndividualizedCell* selects the optimal cell for individualization as follows (parameter $n = 0$):

- Partition $\mathcal{T}$ as result using cell 1 (vertex 12, level 17 reached):

  $T_1^{17} = \{13, 18\}$ with $ADeg(T_1^{17}, W, G) = (4, 0, 0)$
  $T_2^{17} = \{14\}$ with $ADeg(T_2^{17}, W, G) = (3, 0, 0)$
  $T_3^{17} = \{1, 25\}$ with $ADeg(T_3^{17}, W, G) = (0, 0, 0)$
  $T_4^{17} = \{10, 16\}$ with $ADeg(T_4^{17}, W, G) = (4, 0, 0)$

  Since the result partition $\mathcal{T}$ is not last partition, and it is not subpartition of $\mathcal{S}^{10}$, the normal criteria is applied to continue the attempting with next cell. The sum of the number of cells of partition $\mathcal{T}$ and the discarded vertices $(4 + 19)$ is greater than $n = 0$, so that b (best cell) is set to 1 and n is set to $4 + 19 = 23$.

- Partition $\mathcal{T}$ as result using cell 2 (vertex 5, level 17 reached):

  $T_1^{17} = \{13, 18\}$ with $ADeg(T_1^{17}, W, G) = (4, 0, 0)$
  $T_2^{17} = \{14\}$ with $ADeg(T_2^{17}, W, G) = (3, 0, 0)$
  $T_3^{17} = \{10, 16\}$ with $ADeg(T_3^{17}, W, G) = (0, 0, 0)$
  $T_4^{17} = \{1, 25\}$ with $ADeg(T_4^{17}, W, G) = (4, 0, 0)$

  Since the result partition $\mathcal{T}$ is not last partition, and it is not subpartition of $\mathcal{S}^{10}$, the normal criteria is applied to continue the attempting with next cell. The sum of the number of cells of partition $\mathcal{T}$ and the discarded vertices $(4 + 19)$ is equal than $n = 23$, so that b (best cell) size is compared with cell 2 size and best cell b is less than cell 2 and nothing changes.

- Partition $\mathcal{T}$ as result using cell 4 (vertex 1, level 18 reached):

  $T_1^{17} = \{18\}$ with $ADeg(T_1^{17}, W, G) = (4, 0, 0)$
  $T_2^{17} = \{16\}$ with $ADeg(T_2^{17}, W, G) = (3, 0, 0)$
  $T_3^{17} = \{10\}$ with $ADeg(T_3^{17}, W, G) = (0, 0, 0)$

  Since the result partition $\mathcal{T}$ is last partition, we not attempt to individualize any other cell and this cell is chosen as pivot set.

Then, the cell S_4^{10} is chosen for individualization ($P^{10} = 4$), and a vertex refinement is carried out setting $R^{10} = BACKTRACK$. Since we know the algorithm will reached last level partition without any other backtrack level, we summarize the generation of a sequence of partitions up to the end as follows:

- $\mathcal{S}^{11} = (\{12, 21\}, \{22\}, \{6, 13\}, \{5, 18\}, \{10, 14, 16, 20, 25\})$, $R^{11} = VERTEX$, and $P^{11} = 2$.
- $\mathcal{S}^{12} = (\{12, 21\}, \{6, 13\}, \{5, 18\}, \{10, 14, 16, 20\}, \{25\})$, $R^{12} = VERTEX$, and $P^{12} = 5$.

- $\mathcal{S}^{13} = (\{12, 21\}, \{6\}, \{13\}, \{18\}, \{5\}, \{10, 14, 16, 20\})$, $R^{13} = \mathit{VERTEX}$, and $P^{13} = 5$.
- $\mathcal{S}^{14} = (\{12, 21\}, \{6\}, \{13\}, \{18\}, \{10, 14, 16\}, \{20\})$, $R^{14} = \mathit{VERTEX}$, and $P^{14} = 6$.
- $\mathcal{S}^{15} = (\{12\}, \{21\}, \{6\}, \{13\}, \{18\}, \{10, 14, 16\})$, $R^{15} = \mathit{VERTEX}$, and $P^{15} = 2$.
- $\mathcal{S}^{16} = (\{12\}, \{6\}, \{13\}, \{18\}, \{10, 16\}, \{14\})$, $R^{16} = \mathit{VERTEX}$, and $P^{16} = 6$.
- $\mathcal{S}^{17} = (\{12\}, \{6\}, \{13\}, \{18\}, \{10, 16\})$, $R^{17} = \mathit{VERTEX}$, and $P^{17} = 3$.
- $\mathcal{S}^{18} = (\{18\}, \{16\}, \{10\})$.

This partition has only singleton cells. Therefore, the algorithm stops, inducing the following order on the vertices of the graph, which will be used as base generator γ^1 within the set of generators: $2, 15, 24, 17, 23, 9, 0, 3, 11, 7, 8, 1, 4, 19, 22, 25, 5, 20, 21, 14, 13, 12, 6, 18, 16, 10$.

4.2.2 Search for Automorphisms

The search for automorphisms is performed by algorithm *FindAutomorphisms*, which computes the backtrack levels and limit search levels first of all. Let $t = 18$ be the total number of levels, we only take into account those levels that are equitable ($R^l = \mathit{BACKTRACK}$), since the other values are not relevant for our purpose.

From the algorithm *ComputeBacktrackLevels* we get:

$B^0 = -1$	since it is the first backtrack level and $\mathcal{S}^0$ it is not subpartition of any other level.
$B^8 = 0$	since $\mathcal{S}^8$ is not subpartition of $\mathcal{S}^0$.
$B^{10} = -1$	since $\mathcal{S}^{10}$ is subpartition of both $\mathcal{S}^8$ and $\mathcal{S}^0$.

From the algorithm *ComputeLimitLevels* we get:

$L^0 = 10$	since $\mathcal{S}^{10}$ is subpartition of $\mathcal{S}^0$.
$L^8 = 10$	since $\mathcal{S}^{10}$ is subpartition of $\mathcal{S}^8$.
$L^{10} = 18$	since $\mathcal{S}^{10}$ is the last backtrack level.

Next, cells with no links are computed by algorithm *ComputeCellsWithNoLinks*, and it finds out cells with no links and cell size grater than 1 at levels 7 and 10:

γ^2: $S^7_5 = \{3, 11\}$: $2, 15, 24, 17, 23, 9, 0, 11, 3, 7, 8, 1, 4, 19, 22, 25, 5, 20, 21, 14, 13, 12, 6, 18, 16, 10$.
γ^3: $S^{10}_3 = \{4, 19\}$: $2, 15, 24, 17, 23, 9, 0, 3, 11, 7, 8, 1, 19, 4, 22, 25, 5, 20, 21, 14, 13, 12, 6, 18, 16, 10$.

Then, the search tree will be traversed from the backtrack points in ascending order. The only equivalence information that it is had, from $\Gamma = \{\gamma^1, \gamma^2, \gamma^3\}$, is: $(3, 11), (4, 19)$. Any other vertex of the graph is in a singleton orbit. Since $R^{10} = \mathit{BACKTRACK}$, the vertices in the pivot set $S^{10}_4 = \{1, 10, 14, 16, 20, 25\}$ other than 1 (the pivot vertex used in the original sequence of partitions) are tested for equivalence. However, as the pivot cell is not affected by the equivalences, the orbits of the vertices in the pivot cell are marked as valid. Algorithm *CheckAutomorphisms* performs the search for automorphisms at this level 10, setting $\Delta = \{\{\gamma^1, \gamma^2, \gamma^3\}\}$.

Vertex 10 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^{11} = (T_1^{11}, T_2^{11}, T_3^{11}, T_4^{11}, T_5^{11})$ is yielded, which is compatible with $\mathcal{S}^{11}$ and:

$T_1^{11} = \{21, 22\}$	with $ADeg(T_1^{11}, \{10\}, G) = (1, 0, 0)$
$T_2^{11} = \{12\}$	with $ADeg(T_2^{11}, \{10\}, G) = (0, 0, 0)$
$T_3^{11} = \{5, 18\}$	with $ADeg(T_3^{11}, \{10\}, G) = (1, 0, 0)$
$T_4^{11} = \{6, 13\}$	with $ADeg(T_4^{11}, \{10\}, G) = (0, 0, 0)$
$T_5^{11} = \{1, 14, 16, 20, 25\}$	with $ADeg(T_5^{11}, \{10\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^{12} = (\{21, 22\}, \{5, 18\}, \{6, 13\}, \{1, 14, 20, 25\}, \{16\})$.
- $\mathcal{T}^{13} = (\{21, 22\}, \{5\}, \{18\}, \{13\}, \{6\}, \{1, 14, 20, 25\})$.
- $\mathcal{T}^{14} = (\{21, 22\}, \{5\}, \{18\}, \{13\}, \{1, 14, 25\}, \{20\}$.
- $\mathcal{T}^{15} = (\{22\}, \{21\}, \{5\}, \{18\}, \{13\}, \{1, 14, 25\})$.
- $\mathcal{T}^{16} = (\{22\}, \{5\}, \{18\}, \{13\}, \{1, 25\}, \{14\})$.
- $\mathcal{T}^{17} = (\{22\}, \{5\}, \{18\}, \{13\}, \{1, 25\})$.
- $\mathcal{T}^{18} = (\{13\}, \{25\}, \{1\})$.

The compatibility between partitions is complete, and so that, the induced order by the set of partitions $\top$ $\{2, 15, 24, 17, 23, 9, 0, 3, 11, 7, 8, 10, 4, 19, 12, 16, 6, 20, 21, 14, 18, 22, 5, 13, 25, 1\}$ is added to the set of generators as γ^4, and algorithm returns with value $l' = 18$ and the next vertex is considered. Now, the equivalences of the pivot cell remain as: $(1, 10), (16, 25)$, from the set of generators $\Gamma = \{\gamma^1, ..., \gamma^4\}$.

Vertex 14 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^{11} = (T_1^{11}, T_2^{11}, T_3^{11}, T_4^{11}, T_5^{11})$ is yielded, which is compatible with $\mathcal{S}^{11}$ and:

$T_1^{11} = \{12, 22\}$	with $ADeg(T_1^{11}, \{14\}, G) = (1, 0, 0)$
$T_2^{11} = \{21\}$	with $ADeg(T_2^{11}, \{14\}, G) = (0, 0, 0)$
$T_3^{11} = \{5, 6\}$	with $ADeg(T_3^{11}, \{14\}, G) = (1, 0, 0)$
$T_4^{11} = \{13, 18\}$	with $ADeg(T_4^{11}, \{14\}, G) = (0, 0, 0)$
$T_5^{11} = \{1, 10, 16, 20, 25\}$	with $ADeg(T_5^{11}, \{14\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^{12} = (\{12, 22\}, \{5, 6\}, \{13, 18\}, \{1, 10, 16, 25\}, \{20\})$.

Partition $\mathcal{T}^{12}$ is not compatible with $\mathcal{S}^{12}$, since $ADeg(S_1^{12}, S_5^{12}, G) \neq ADeg(T_1^{12}, T_5^{12}, G)$, and this and the other differences between partitions (if any more) are used to generate the failure recording F (hash value). The function returns until level 10.

Then, next vertex 16 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^{11} = (T_1^{11}, T_2^{11}, T_3^{11}, T_4^{11}, T_5^{11})$ is yielded, which is compatible with $\mathcal{S}^{11}$ and:

$T_1^{11} = \{21, 22\}$ with $ADeg(T_1^{11}, \{16\}, G) = (1, 0, 0)$
$T_2^{11} = \{12\}$ with $ADeg(T_2^{11}, \{16\}, G) = (0, 0, 0)$
$T_3^{11} = \{5, 13\}$ with $ADeg(T_3^{11}, \{16\}, G) = (1, 0, 0)$
$T_4^{11} = \{6, 18\}$ with $ADeg(T_4^{11}, \{16\}, G) = (0, 0, 0)$
$T_5^{11} = \{1, 10, 14, 20, 25\}$ with $ADeg(T_5^{11}, \{16\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^{12} = (\{21, 22\}, \{5, 13\}, \{6, 18\}, \{1, 14, 20, 25\}, \{10\})$.
- $\mathcal{T}^{13} = (\{21, 22\}, \{5\}, \{13\}, \{18\}, \{6\}, \{1, 14, 20, 25\})$.
- $\mathcal{T}^{14} = (\{21, 22\}, \{5\}, \{13\}, \{18\}, \{1, 14, 25\}, \{20\}$.
- $\mathcal{T}^{15} = (\{22\}, \{21\}, \{5\}, \{13\}, \{18\}, \{1, 14, 25\})$.
- $\mathcal{T}^{16} = (\{22\}, \{5\}, \{13\}, \{18\}, \{1, 25\}, \{14\})$.
- $\mathcal{T}^{17} = (\{22\}, \{5\}, \{13\}, \{18\}, \{1, 25\})$.
- $\mathcal{T}^{18} = (\{18\}, \{1\}, \{25\})$.

The compatibility between partitions is complete, and so that, the induced order by the set of partitions $\mathsf{T}\ 2, 15, 24, 17, 23, 9, 0, 3, 11, 7, 8, 16, 4, 19, 12, 10, 6, 20, 21, 14, 13, 22, 5, 18, 1, 25$ is added to the set of generators as γ^5, and algorithm returns with value $l' = 18$ and the next vertex is considered. Now, the equivalences of the pivot cell remain as: $(1, 10, 16, 25)$, from the set of generators $\Gamma = \{\gamma^1, ..., \gamma^5\}$.

Now, vertex 20 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^{11} = (T_1^{11}, T_2^{11}, T_3^{11}, T_4^{11}, T_5^{11})$ is yielded, which is compatible with $\mathcal{S}^{11}$ and:

$T_1^{11} = \{12, 22\}$ with $ADeg(T_1^{11}, \{20\}, G) = (1, 0, 0)$
$T_2^{11} = \{21\}$ with $ADeg(T_2^{11}, \{20\}, G) = (0, 0, 0)$
$T_3^{11} = \{13, 18\}$ with $ADeg(T_3^{11}, \{20\}, G) = (1, 0, 0)$
$T_4^{11} = \{5, 6\}$ with $ADeg(T_4^{11}, \{20\}, G) = (0, 0, 0)$
$T_5^{11} = \{1, 10, 14, 16, 25\}$ with $ADeg(T_5^{11}, \{20\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^{12} = (\{12, 22\}, \{13, 18\}, \{5, 6\}, \{1, 10, 16, 25\}, \{14\})$.

Partition $\mathcal{T}^{12}$ is not compatible with $\mathcal{S}^{12}$, since $ADeg(S_1^{12}, S_5^{12}, G) \neq ADeg(T_1^{12}, T_5^{12}, G)$, and this and the other differences between partitions (if any more) are used to generate the failure recording F (hash value). The function returns until level 10.

Vertex 25 is already known to be equivalent to vertex 1, and no more vertices remain. R^{10} is not set, because not all vertices in S_4^{10} are equivalent. A next call to algorithm *CheckAutomorphisms* is done in order to find out automorphisms at level 8.

Since $R^8 = BACKTRACK$, the vertices in the pivot set $S_3^8 = \{7, 8\}$ other than 8 (the pivot vertex used in the original sequence of partitions) are tested for equivalence. However, as the pivot cell is not affected by the equivalences, the orbits of the vertices in the pivot cell are marked as valid. It is set $\Delta = \{\{\gamma^1, ..., \gamma^5\}\}$.

Vertex 8 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^9 = (T_1^9, T_2^9, T_3^9, T_4^9, T_5^9)$ is yielded, which is compatible with $\mathcal{S}^9$ and:

$T_1^9 = \{12, 21, 22\}$	with $ADeg(T_1^9, \{8\}, G) = (1, 0, 0)$
$T_2^9 = \{5, 6, 13, 18\}$	with $ADeg(T_2^9, \{8\}, G) = (0, 0, 0)$
$T_3^9 = \{7\}$	with $ADeg(T_3^9, \{8\}, G) = (0, 0, 0)$
$T_4^9 = \{4, 19\}$	with $ADeg(T_4^9, \{8\}, G) = (1, 0, 0)$
$T_5^9 = \{1, 10, 14, 16, 20, 25\}$	with $ADeg(T_5^9, \{8\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^{10} = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{4, 19\}, \{1, 10, 14, 16, 20, 25\})$.

Since $\mathcal{T}^{10} = \mathcal{S}^{10}$ (equal partitions), algorithm stop searching for a compatible sequence of partitions, because it is trivial to complete it. Thus, the induced order by the set of partitions T $2, 15, 24, 17, 23, 9, 0, 3, 11, 8, 7, 1, 4, 19, 22, 25, 5, 20, 21, 14, 13, 12, 6, 18, 16, 10$ is added to the set of generators as γ^6, and algorithm returns with value $l' = 10$ and no more vertices remain in the pivot cell. Then, algorithm sets $R^8 = VERTEX$ because all vertices in the pivot set are equivalent. Observe that, if $\mathcal{T}^{10}$ and $\mathcal{S}^{10}$ would have not been equal partitions, the search limit level $L^8 = 10$, so that an automorphism would have been found, if the partitions were compatible, without continue the search of a compatible sequence of partitions up to the end. A next call to algorithm *CheckAutomorphisms* is done in order to find out automorphisms at level 0.

Since $R^0 = BACKTRACK$, the vertices in the pivot set $S_2^0 = \{2, 5, 6, 7, 8, 13, 18, 23\}$ other than 2 (the pivot vertex used in the original sequence of partitions) are tested for equivalence. Now, the equivalences of the pivot cell are: $(5, 6), (7, 8), (13, 18)$, from the set of generators $\Gamma = \{\gamma^1, ..., \gamma^6\}$. It is set $\Delta = \{\{\gamma^1, ..., \gamma^6\}\}$.

Vertex 5 is used for generate an alternative sequence of partitions, what as first step it is individualized and partition $\mathcal{T}^1 = (T_1^1, T_2^1, T_3^1, T_4^1, T_5^1)$ is yielded, which is compatible with $\mathcal{S}^1$ and:

$T_1^1 = \{15, 17, 24\}$	with $ADeg(T_1^1, \{5\}, G) = (1, 0, 0)$
$T_2^1 = \{12, 21, 22\}$	with $ADeg(T_2^1, \{5\}, G) = (0, 0, 0)$
$T_3^1 = \{2, 6, 7, 8, 13, 18, 23\}$	with $ADeg(T_3^1, \{5\}, G) = (0, 0, 0)$
$T_4^1 = \{10, 14, 16\}$	with $ADeg(T_4^1, \{5\}, G) = (1, 0, 0)$
$T_5^1 = \{0, 1, 3, 4, 9, 11, 19, 20, 25\}$	with $ADeg(T_5^1, \{5\}, G) = (0, 0, 0)$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^2 = (\{15, 17, 24\}, \{12, 21, 22\}, \{2, 7, 8, 23\}, \{6, 13, 18\}, \{10, 14, 16\}, \{1, 20, 25\},$ $\{0, 3, 4, 9, 11, 19\})$

Clearly, vertex 5 does not generate a sequence of partitions compatible with the original one, since $\mathcal{T}^2$ is not compatible with $\mathcal{S}^2$ (number of cells). Then, this and the other differences between partitions (if any more) are used to generate the failure recording F (hash value). The function returns until level 0.

Vertex 6 is equivalent to vertex 5, which has been already tried.

Next vertex 7 is attempt to generate an alternative sequence of partitions. It is individualized and partition $\mathcal{T}^1 = (T_1^1, T_2^1, T_3^1, T_4^1, T_5^1)$ is yielded, which is compatible with $\mathcal{S}^1$ and:

$$\begin{array}{ll}
T_1^1 = \{12, 21, 22\} & \text{with } ADeg(T_1^1, \{7\}, G) = (1, 0, 0) \\
T_2^1 = \{15, 17, 24\} & \text{with } ADeg(T_2^1, \{7\}, G) = (0, 0, 0) \\
T_3^1 = \{2, 5, 6, 8, 13, 18, 23\} & \text{with } ADeg(T_3^1, \{7\}, G) = (0, 0, 0) \\
T_4^1 = \{4, 9, 19\} & \text{with } ADeg(T_4^1, \{7\}, G) = (1, 0, 0) \\
T_5^1 = \{0, 1, 3, 10, 11, 14, 16, 20, 25\} & \text{with } ADeg(T_5^1, \{7\}, G) = (0, 0, 0)
\end{array}$$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^2 = (\{12, 21, 22\}, \{15, 17, 24\}, \{5, 6, 13, 18\}, \{2, 8, 23\}, \{4, 9, 19\}, \{0, 3, 11\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^3 = (\{12, 21, 22\}, \{17\}, \{24\}, \{15\}, \{5, 6, 13, 18\}, \{2, 23\}, \{8\}, \{4, 9, 19\}, \{0, 3, 11\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^4 = (\{12, 21, 22\}, \{24\}, \{15\}, \{5, 6, 13, 18\}, \{2, 23\}, \{8\}, \{9\}, \{4, 19\}, \{0, 3, 11\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^5 = (\{12, 21, 22\}, \{15\}, \{5, 6, 13, 18\}, \{2, 23\}, \{8\}, \{9\}, \{4, 19\}, \{3, 11\}, \{0\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^6 = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{2, 23\}, \{8\}, \{9\}, \{4, 19\}, \{3, 11\}, \{0\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^7 = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{2, 23\}, \{9\}, \{4, 19\}, \{3, 11\}, \{0\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^8 = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{2, 23\}, \{3, 11\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^9 = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{23\}, \{3, 11\}, \{1, 10, 14, 16, 20, 25\})$
- $\mathcal{T}^{10} = (\{12, 21, 22\}, \{5, 6, 13, 18\}, \{3, 11\}, \{1, 10, 14, 16, 20, 25\})$

Since $\mathcal{T}^{10} = \mathcal{S}^{10}$ (equal partitions, because cell S_3^{10} has no links), algorithm stop searching for a compatible sequence of partitions, because it is trivial to complete it. Thus, the induced order by the set of partitions $\mathsf{T}\ 7, 17, 24, 15, 8, 0, 9, 4, 19, 2, 23, 1, 3, 11, 22, 25, 5, 20, 21, 14, 13, 12, 6, 18, 16, 10$ is added to the set of generators as γ^7, and algorithm returns with value $l' = 10$ and the algorithm goes on the searching. Observe that, if $\mathcal{T}^{10}$ and $\mathcal{S}^{10}$ would have not been equal partitions, the search limit level $L^8 = 10$, so that an automorphism would have been found, if the partitions were compatible, without continue the search of a compatible sequence of partitions up to the end.

The equivalence among vertices are: $(2, 8, 7, 23), (5, 6), (13, 18)$, from the set of generators $\Gamma = \{\gamma^1, ..., \gamma^7\}$. Next vertex 13 is individualized and attempt to generate an alternative sequence of partitions. The partition $\mathcal{T}^1 = (T_1^1, T_2^1, T_3^1, T_4^1, T_5^1)$ is yielded, which is compatible with $\mathcal{S}^1$ and:

$$\begin{array}{ll}
T_1^1 = \{15, 17, 24\} & \text{with } ADeg(T_1^1, \{13\}, G) = (1, 0, 0) \\
T_2^1 = \{12, 21, 22\} & \text{with } ADeg(T_2^1, \{13\}, G) = (0, 0, 0) \\
T_3^1 = \{2, 5, 6, 7, 8, 18, 23\} & \text{with } ADeg(T_3^1, \{13\}, G) = (0, 0, 0) \\
T_4^1 = \{1, 16, 20\} & \text{with } ADeg(T_4^1, \{13\}, G) = (1, 0, 0) \\
T_5^1 = \{0, 3, 4, 9, 10, 11, 14, 16, 25\} & \text{with } ADeg(T_5^1, \{13\}, G) = (0, 0, 0)
\end{array}$$

The followings compatible sequence of partitions, generated by algorithm *SubtreeCompatible* are summarized as follows:

- $\mathcal{T}^2 = (\{15, 17, 24\}, \{12, 21, 22\}, \{2, 7, 8, 23\}, \{5, 6, 18\}, \{1, 16, 20\}, \{10, 14, 25\}, \{0, 3, 4, 9, 11, 19\})$

Clearly, vertex 5 does not generate a sequence of partitions compatible with the original one, since $\mathcal{T}^2$ is not compatible with $\mathcal{S}^2$ (number of cells). Then, this and the other differences between partitions (if any more) are used to generate the failure recording F (hash value). The function returns until level 0.

Remaining vertices of pivot cell at level 0 are not tried, because belongs to non valid orbits (they are equivalent to already tried vertices).

Thus, the automorphism group of the graph is computed (the set of generators Γ) from which we know the orbit partitions:

$$(0, 9)\ (16, 25, 10, 1)\ (8, 7, 2, 23)\ (4, 19, 3, 11)\ (6, 5)\ (12, 22)\ (18, 13)\ (14)\ (17, 15)\ (20)\ (21)\ (24)$$

and the size of the automorphism group as follows:

- Calculate the product of factorials of the cell with no links sizes (greater than 1) which were discarded. These cells were $|S_5^7| = 2$ and $|S_3^{10}| = 2$, and we obtain: $2! * 2! = 2^2$.
- Calculate the product of number of equivalent vertices to the pivot vertex at each initial *BACKTRACK* level. These values are: for level 10 $= 2^2$, for level 8 $= 2$, and for level 0 $= 2^2$, and the number obtained is: $2^2 * 2 * 2^2 = 2^5$.
- The automorphism group size is the product of these calculations: $|Aut(G)| = 2^2 * 2^5 = 2^7$.

4.3 Example: Isomorphism Test

In this section we will show how our algorithm computes isomorphism test between two graphs (Figure 4.8). These sample graphs are isomorphic, and we will observe the algorithm behaviour.

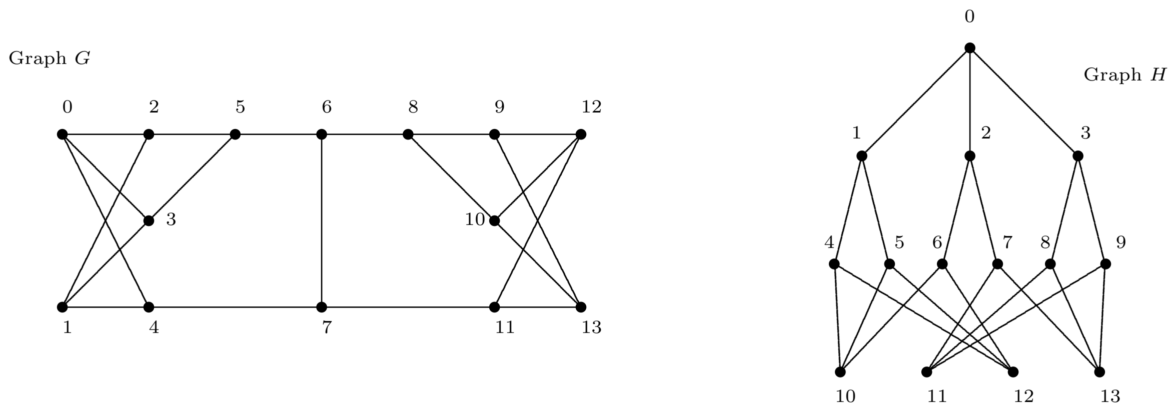

Figure 4.8: Sample graphs for isomorphism testing.

Since the algorithm generate a sequence of partitions for each graph, and then it does a search for automorphism (computing the automorphism group), these steps have been detailed in previous section, so then, we will ommit them, but summarize the results.

For graph G, the sequence of partitions is as follows:

- $\mathcal{S}^0 = (\{0,1,2,3,4,5,6,7,8,9,10,11,12,13\})$, $R^0 = BACKTRACK$, and $P^0 = 1$.
- $\mathcal{S}^1 = (\{2,3,4\},\{1,5,6,7,8,9,10,11,12,13\})$, $R^1 = SET$, and $P^1 = 1$.
- $\mathcal{S}^2 = (\{2,3,4\},\{1\},\{5\},\{7\},\{6,8,9,10,11,12,13\})$, $R^2 = VERTEX$, and $P^2 = 2$.
- $\mathcal{S}^3 = (\{2,3,4\},\{5\},\{7\},\{6,8,9,10,11,12,13\})$, $R^3 = VERTEX$, and $P^3 = 2$.
- $\mathcal{S}^4 = (\{2,3\},\{4\},\{7\},\{6\},\{8,9,10,11,12,13\})$, $R^4 = VERTEX$, and $P^4 = 3$.
- $\mathcal{S}^5 = (\{4\},\{6\},\{11\},\{8,9,10,12,13\})$, $R^5 = VERTEX$, and $P^5 = 3$.
- $\mathcal{S}^6 = (\{6\},\{12,13\},\{8,9,10\})$, $R^6 = VERTEX$, and $P^6 = 1$.
- $\mathcal{S}^7 = (\{12,13\},\{8\},\{9,10\})$, $R^7 = VERTEX$, and $P^7 = 2$.
- $\mathcal{S}^8 = (\{12,13\},\{9,10\})$, $R^8 = VERTEX$, and $P^8 = 1$.
- $\mathcal{S}^9 = (\{13\},\{9,10\})$, $R^9 = VERTEX$, and $P^9 = 1$.
- $\mathcal{S}^{10} = (\{9,10\})$.

Note that, the partition at level 8 was a backtrack level which was changed to $R^8 = VERTEX$ because all vertices of $S^8_{P^8}$ was found to be equivalent.

And the automorphism group of graph G as follows:

- Set of generators:
  $\gamma^1 = 0,1,5,7,2,3,11,4,6,8,12,13,9,10$
  $\gamma^2 = 0,1,5,7,2,3,11,4,6,8,12,13,10,9$
  $\gamma^3 = 0,1,5,7,3,2,11,4,6,8,12,13,9,10$
  $\gamma^4 = 0,1,5,7,2,3,11,4,6,8,13,12,9,10$
  $\gamma^5 = 1,0,5,7,2,3,11,4,6,8,12,13,9,10$
  $\gamma^6 = 2,3,4,6,0,1,8,5,7,11,9,10,12,13$
  $\gamma^7 = 9,10,11,6,12,13,5,8,7,4,2,3,0,1$
- Orbits:
  $(0,1,2,3,9,10,13,12),(4,5,8,11),(6,7)$
- $|Aut(G)| = 2^2 * 2^4 = 2^6$

We obtain the extended sequence of partitions $\mathsf{E}_G = (\mathsf{Q}_G, \mathsf{B}_G, \mathsf{L}_G, \mathsf{F}_G, \Gamma_G)$, where $\mathsf{Q}_G = (\mathsf{S},\mathsf{R},\mathsf{P})$, and $\Gamma = \{\gamma^1, ..., \gamma^7\}$.

Since the sequence of partitions for graph G has remaining backtrack levels, a complete sequence of partitions will be generated for grah H, and its sequence of partitions is as follows:

- $\mathcal{S}^0 = (\{0,1,2,3,4,5,6,7,8,9,10,11,12,13\})$, $R^0 = BACKTRACK$, and $P^0 = 1$.
- $\mathcal{S}^1 = (\{1,2,3\},\{4,5,6,7,8,9,10,11,12,13\})$, $R^1 = SET$, and $P^1 = 1$.
- $\mathcal{S}^2 = (\{1,2,3\},\{4,5,6,7,8,9\},\{10,11,12,13\})$, $R^2 = VERTEX$, and $P^2 = 3$.
- $\mathcal{S}^3 = (\{1,2,3\},\{4,5,6\},\{7,8,9\},\{11,12,13\})$, $R^3 = VERTEX$, and $P^3 = 7$.
- $\mathcal{S}^4 = (\{3\},\{2\},\{1\},\{4,5,6\},\{7,8,9\},\{11,13\},\{12\})$, $R^4 = VERTEX$, and $P^4 = 7$.
- $\mathcal{S}^5 = (\{3\},\{2\},\{1\},\{4,5,6\},\{7,8,9\},\{11,13\})$, $R^5 = VERTEX$, and $P^5 = 1$.

- $\mathcal{S}^6 = (\{2\}, \{1\}, \{4,5,6\}, \{8,9\}, \{7\}, \{11,13\})$, $R^6 = \mathit{VERTEX}$, and $P^6 = 5$.
- $\mathcal{S}^7 = (\{2\}, \{1\}, \{4,5,6\}, \{8,9\}, \{11,13\})$, $R^7 = \mathit{VERTEX}$, and $P^7 = 2$.
- $\mathcal{S}^8 = (\{2\}, \{4,5\}, \{6\}, \{8,9\}, \{11,13\})$, $R^8 = \mathit{VERTEX}$, and $P^8 = 1$.
- $\mathcal{S}^9 = (\{6\}, \{8,9\}, \{11,13\})$, $R^9 = \mathit{VERTEX}$, and $P^9 = 2$.
- $\mathcal{S}^{10} = (\{9\}, \{11,13\})$, $R^{10} = \mathit{VERTEX}$, and $P^{10} = 1$.
- $\mathcal{S}^{11} = (\{11,13\})$.

Note that, the partitions at levels 2 and 9 were both backtrack level which were changed to $R^2 = R^9 = \mathit{VERTEX}$ because all vertices of $S^2_{P^2}$ and $S^9_{P^9}$ were found to be equivalent respectively.

And the automorphism group of graph H as follows:

- Set of generators:
  $\gamma^1 = 0, 10, 12, 3, 7, 1, 2, 4, 5, 8, 6, 9, 11, 13$
  $\gamma^2 = 0, 10, 12, 3, 7, 1, 2, 4, 5, 8, 6, 9, 13, 11$
  $\gamma^3 = 0, 10, 12, 3, 7, 1, 2, 5, 4, 8, 6, 9, 11, 13$
  $\gamma^4 = 0, 10, 12, 3, 7, 1, 2, 4, 5, 9, 6, 8, 11, 13$
  $\gamma^5 = 0, 11, 13, 1, 6, 3, 2, 8, 9, 4, 7, 5, 10, 12$
  $\gamma^6 = 2, 4, 5, 7, 3, 6, 0, 10, 12, 11, 1, 13, 8, 9$
- Orbits:
  $(0,2), (1,3,6,7), (4,5,8,9,10,11,12,13)$
- $|\mathit{Aut}(H)| = 2^2 * 2^4 = 2^6$

We obtain the extended sequence of partitions $\mathsf{E}_H = (\mathsf{Q}_H, \mathsf{B}_H, \mathsf{L}_H, \mathsf{F}_H, \Gamma_H)$, where $\mathsf{Q}_H = (\mathsf{S}, \mathsf{R}, \mathsf{P})$, and $\Gamma = \{\gamma^1, ..., \gamma^6\}$.

Since $\mathit{Aut}(G) = \mathit{Aut}(H)$ and $\mathit{Orbits}(\mathsf{E}_G) = \mathit{Orbits}(\mathsf{E}_H)$, the algorithm will attempt to find a sequence of partitions compatible with the sequence of partitions of the other graph.

4.3.1 Match Graphs G and H

Being that both graph has the same number of backtrack levels (1), any of the sequences of partitions may be the target one. The sequence of partitions $\mathit{SeqPart}(\mathsf{E}_G)$ is chosen as the target (line 20 of Algorithm *AreIsomorphic*). Then, Algorithm *Match* will try to find a sequence of partitions for graph H compatible with this. It it is possible to find one such sequence, it returns a value ≥ 0 (i.e., the level where the compatible was found).

Algorithm *Match* is called with parameters 0 for the starting level, G for the graph whose sequence of partitions is $\mathit{SeqPart}(\mathsf{Q}_G)$, and H for the graph whose initial partition is $\mathcal{D}_H = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\})$ and whose known orbits are $\mathit{Orbits}(\mathsf{Q}_H) = \{\{0, 2\},$ $\{1, 3, 6, 7\}$, $\{4, 5, 8, 9, 10, 11, 12, 13\}\}$.

Since $R^0 = \mathit{BACKTRACK}$ and it is the first level ($B^0 = -1$), the partial orbits computed for graph H are the total orbits. Then, the compatibility between the orbits of both graph is complete, and only those orbits of graph H, that are the same size as the orbit which pivot vertex of the sequence of partitions for graph G belongs, are considered. Pivot vertex used at level 0 in the sequence of partitions of graph G was vertex 0, which $|\mathit{OrbitsOf}(0, \Gamma_G)| = 8$.

Thus, only one orbit of graph H will be considered, and consequently, an only one vertex will be attempted to obtain a sequence of partitions for graph H, compatible with the target.

First, vertex 4 is chosen, and a new partition $\mathcal{T}' = (T'_1, T'_2)$ is generated refining the initial partition $\mathcal{T} = \mathcal{D}_H$ by vertex using 4 as the pivot vertex, where:

$$\begin{array}{ll}
T'_1 = \{1, 10, 12\} & \text{with } \mathit{ADeg}(T'_1, W', G) = (1, 0, 0) \\
T'_2 = \{0, 2, 3, 5, 6, 7, 8, 9, 11, 13\} & \text{with } \mathit{ADeg}(T'_2, W', G) = (0, 0, 0)
\end{array}$$

This new partition is compatible with the target partition $\mathcal{S}^1$ (recall that this will always be the case, since both graphs are regular of degree three, and have the same number of vertices), so a recursive call is made to process the next partition in the sequence.

Since $R^1 = \mathit{SET}$, a set refinement is performed using cell $\{1, 10, 12\}$ as the pivot set (recall that $P^1 = 1$). This yields a partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where:

$$\begin{array}{ll}
T'_1 = \{1, 10, 12\} & \text{with } \mathit{ADeg}(T'_1, \{1, 10, 12\}, H) = (0, 0, 0) \\
T'_2 = \{5\} & \text{with } \mathit{ADeg}(T'_2, \{1, 10, 12\}, H) = (3, 0, 0) \\
T'_3 = \{6\} & \text{with } \mathit{ADeg}(T'_3, \{1, 10, 12\}, H) = (2, 0, 0) \\
T'_4 = \{0\} & \text{with } \mathit{ADeg}(T'_4, \{1, 10, 12\}, H) = (1, 0, 0) \\
T'_5 = \{2, 3, 7, 8, 9, 11, 13\} & \text{with } \mathit{ADeg}(T'_2, \{1, 10, 12\}, H) = (0, 0, 0)
\end{array}$$

This partition is compatible with the target $\mathcal{S}^2$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^2 = 2$ and $R^2 = \mathit{VERTEX}$, pivot cell $\{5\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4)$, where:

$$\begin{array}{ll}
T'_1 = \{1, 10, 12\} & \text{with } \mathit{ADeg}(T'_1, \{5\}, H) = (0, 0, 0) \\
T'_2 = \{6\} & \text{with } \mathit{ADeg}(T'_2, \{5\}, H) = (2, 0, 0) \\
T'_3 = \{0\} & \text{with } \mathit{ADeg}(T'_3, \{5\}, H) = (1, 0, 0) \\
T'_4 = \{2, 3, 7, 8, 9, 11, 13\} & \text{with } \mathit{ADeg}(T'_4, \{5\}, H) = (0, 0, 0)
\end{array}$$

This partition is compatible with the target $\mathcal{S}^3$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^3 = 2$ and $R^3 = \mathit{VERTEX}$, pivot cell $\{6\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where:

$$\begin{array}{ll}
T'_1 = \{10, 12\} & \text{with } \mathit{ADeg}(T'_1, \{6\}, H) = (1, 0, 0) \\
T'_2 = \{1\} & \text{with } \mathit{ADeg}(T'_2, \{6\}, H) = (0, 0, 0) \\
T'_3 = \{0\} & \text{with } \mathit{ADeg}(T'_3, \{6\}, H) = (0, 0, 0) \\
T'_4 = \{2\} & \text{with } \mathit{ADeg}(T'_4, \{6\}, H) = (1, 0, 0) \\
T'_5 = \{3, 7, 8, 9, 11, 13\} & \text{with } \mathit{ADeg}(T'_5, \{6\}, H) = (0, 0, 0)
\end{array}$$

This partition is compatible with the target $\mathcal{S}^4$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^4 = 3$ and $R^4 = \mathit{VERTEX}$, pivot cell $\{0\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4)$, where:

$$\begin{array}{ll}
T'_1 = \{1\} & \text{with } \mathit{ADeg}(T'_1, \{0\}, H) = (1, 0, 0) \\
T'_2 = \{2\} & \text{with } \mathit{ADeg}(T'_2, \{0\}, H) = (1, 0, 0) \\
T'_3 = \{3\} & \text{with } \mathit{ADeg}(T'_3, \{0\}, H) = (1, 0, 0) \\
T'_4 = \{7, 8, 9, 11, 13\} & \text{with } \mathit{ADeg}(T'_4, \{0\}, H) = (0, 0, 0)
\end{array}$$

This partition is compatible with the target $\mathcal{S}^5$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^5 = 3$ and $R^5 = \mathit{VERTEX}$, pivot cell $\{3\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where:

$$\begin{array}{ll} T'_1 = \{2\} & \text{with } ADeg(T'_1, \{3\}, H) = (0,0,0) \\ T'_2 = \{8,9\} & \text{with } ADeg(T'_2, \{3\}, H) = (1,0,0) \\ T'_3 = \{7,11,13\} & \text{with } ADeg(T'_3, \{3\}, H) = (0,0,0) \end{array}$$

This partition is compatible with the target $\mathcal{S}^6$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^6 = 1$ and $R^6 = \mathit{VERTEX}$, pivot cell $\{2\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where:

$$\begin{array}{ll} T'_1 = \{8,9\} & \text{with } ADeg(T'_1, \{2\}, H) = (0,0,0) \\ T'_2 = \{7\} & \text{with } ADeg(T'_2, \{2\}, H) = (1,0,0) \\ T'_3 = \{11,13\} & \text{with } ADeg(T'_3, \{2\}, H) = (0,0,0) \end{array}$$

This partition is compatible with the target $\mathcal{S}^7$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^7 = 2$ and $R^7 = \mathit{VERTEX}$, pivot cell $\{7\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2)$, where:

$$\begin{array}{ll} T'_1 = \{8,9\} & \text{with } ADeg(T'_1, \{7\}, H) = (0,0,0) \\ T'_2 = \{11,13\} & \text{with } ADeg(T'_2, \{7\}, H) = (1,0,0) \end{array}$$

This partition is also compatible with the target $\mathcal{S}^8$. Hence, a recursive call is made to process the next partition. Recall that R^8 was changed from *BACKTRACK* to *VERTEX* during the search for automorphisms. Then, a vertex refinement is performed, using any vertex in $T'_1 = \{8,9\}$, since $P^8 = 1$. Let us lexicographically choose vertex 8 as the pivot. Thus we get a partition $\mathcal{T}' = (T'_1, T'_2)$, where:

$$\begin{array}{ll} T'_1 = \{9\} & \text{with } ADeg(T'_1, \{8\}, H) = (0,0,0) \\ T'_2 = \{11,13\} & \text{with } ADeg(T'_2, \{8\}, H) = (1,0,0) \end{array}$$

This partition is compatible with the target $\mathcal{S}^9$ and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^9 = 1$ and $R^9 = \mathit{VERTEX}$, pivot cell $\{9\}$ is used for a vertex refinement, what yields a new partition $\mathcal{T}' = (T'_1)$, where:

$$T'_1 = \{11,13\} \quad \text{with } ADeg(T'_1, \{9\}, H) = (1,0,0)$$

This last partition is compatible with the target $\mathcal{S}^{10}$, and as consequently, the complete alternative sequence of partitions for graph H results to be compatible with this for graph G. Thus, an isomorphism of both graphs was found, which correspondence derives from the induced order of both compatible sequences of partitions as follows:

Graph G	0	1	5	7	2	3	11	4	6	8	12	13	9	10
Graph H	4	5	6	0	11	12	3	1	2	7	8	9	11	13

4.4 Experimental Results

In this section we compare the performance of conauto-2.0 against other algorithms for graph isomorphism testing and automorphism group computation. The experiments have been carried out in an Intel i7 Q 720 @ 1.6GHz with 8GiB of RAM under Ubuntu 10.04. All programs have been compiled with gcc 4.4.3 with their respective default configuration, but modified to perform isomorphism testing or automorphism group computation, depending on the experiment.

4.4.1 Graph Isomorphism Test Performance

We will compare the graph canonical labeling performance of bliss (both 0.35 and 0.72 versions) and nauty, against our algorithm, which not compute any canonical labeling for graphs. This experiment is intended to see how the size of the graphs affects the running time of isomorphism testing programs. Each point shown in the plots corresponds to the average running time of 100 executions with different instances of the corresponding graph. The CPU time limit of each execution has been set to 10,000 seconds for this experiment. Once an algorithm exceeds the time limit for any graph size instance, this point is not considered, and the execution stops.

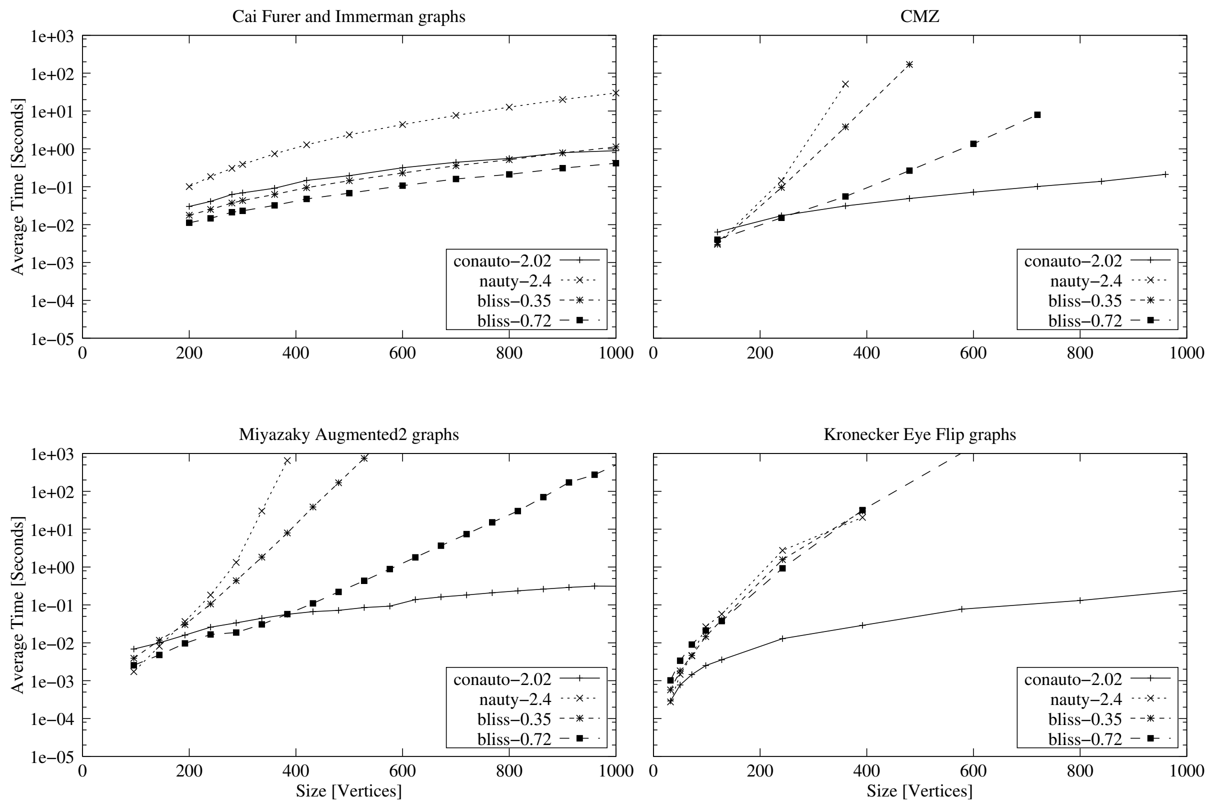

Figure 4.9: Performance for GI-testing with CFI, CMZ, MZ2, and KEF graph families.

Cai Fürer and Immerman (CFI), CMZ series, Miyazaki Augmented2 (MZ2), and Kronecker Eye Flip (KEF) graphs, are graph families from bliss benchmark [8], and we can see their performance in Figure 4.9. The results for CFI graphs are similar for all the algorithms, being bliss-0.72 the fastest, although conauto-2.0 seems to be closer as the graph sizes grows. It is also observable that nauty is more than one order of magnitude slower. The results for

CMZ and Miyazaki Augmented2 graphs are almost the same, where nauty and bliss-0.35 have exponential behaviour and bliss-0.72 also seems to be exponential, while conauto-2.0 has clearly no exponential behaviour in spite of not being the fastest for smaller graph sizes. And for KEF graphs performance, conauto-2.0 is the fastest algorithm, while the rest are unable to finish, before the time limit, for the largest graph instances.

Desarguesian Projective Planes of order 2 (PG2), and Affine Geometries graphs (AG2), are graph families derived from projective planes and projective geometries respectively, and we can see their performance in Figure 4.10. The results are quite similar, where nauty has exponential behaviour in both cases, while the rest have similar results, being bliss-0.35 the fastest. It is observable from [22] that the previous version of conauto was even slower than nauty, which is unable to finish for more than 200 vertices graph instances, while now its performance is closer to the fastest algorithm.

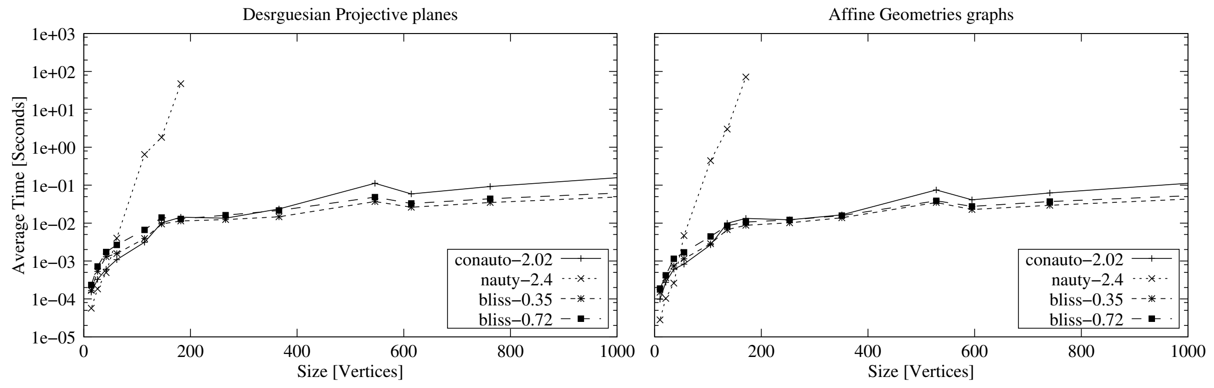

Figure 4.10: Performance for GI-testing with PG2 and AG2 graph families.

We classify the Hadamard Matrices from bliss benchmark for their number of orbits. In Figure 4.11 we have considered those which have 1 (H1O) and 2 (H2O) orbits respectively for performance. The results for Hadamard Matrices of 1 orbit are similar for all the algorithms, being conauto-2.0 the fastest, and also it seems to be more stable when a time peak appears in the figure. Moreover, conauto-2.0 is almost one order of magnitude faster than the next closer algorithm. On the other hand, the results for Hadamard Matrices of 2 orbits seems to be exponential for all the algorithms, but conauto-2.0, because the rest are unable to finish before the time limit for the largest graph instances. Furthermore, conauto-2.0 is always one or more orders of magnitude faster than the rest.

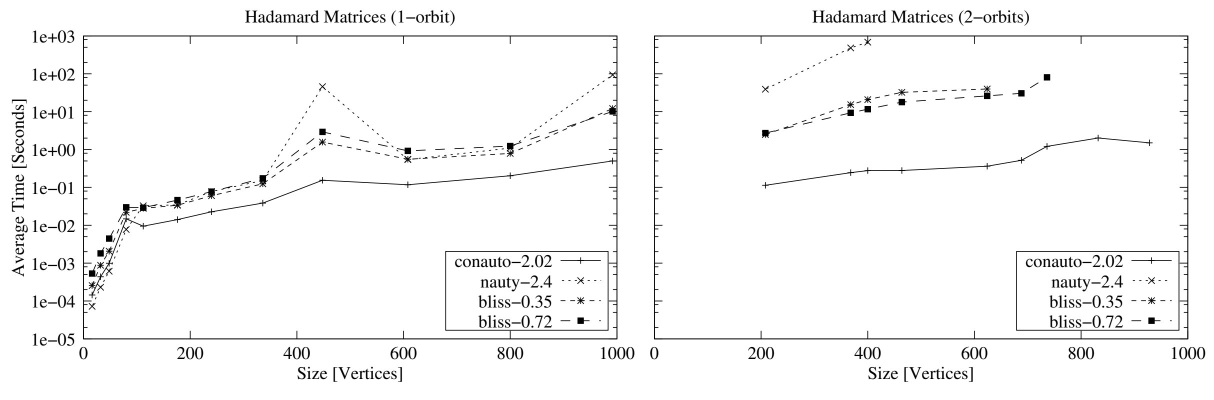

Figure 4.11: Performance for GI-testing with H1O and H2O graph families.

The Strongly Regular graph are considered very hard for graph isomorphism testing. However, they are dealt within considerable time with the current algorithms. So that, the performance for these graph families are similar for the algorithms considered, and the results does not contribute relevant information. However, the Steiner Triple System (STS) and Latin Square (LST) graph families are a little different performance, and they are shown in Figure 4.12, where conauto-2.0 is the fastest. All the algorithm have not exponential behaviour, and all of them have similar results. Moreover, conauto-2.0 is almost one order of magnitude faster than the closer algorithm, bliss-0.72.

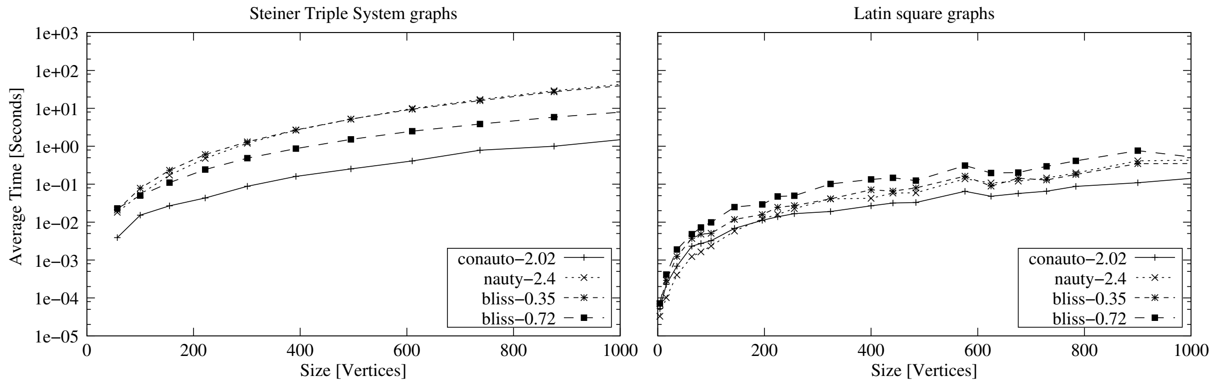

Figure 4.12: Performance for GI-testing with Strongly Regular graphs (STS and LST).

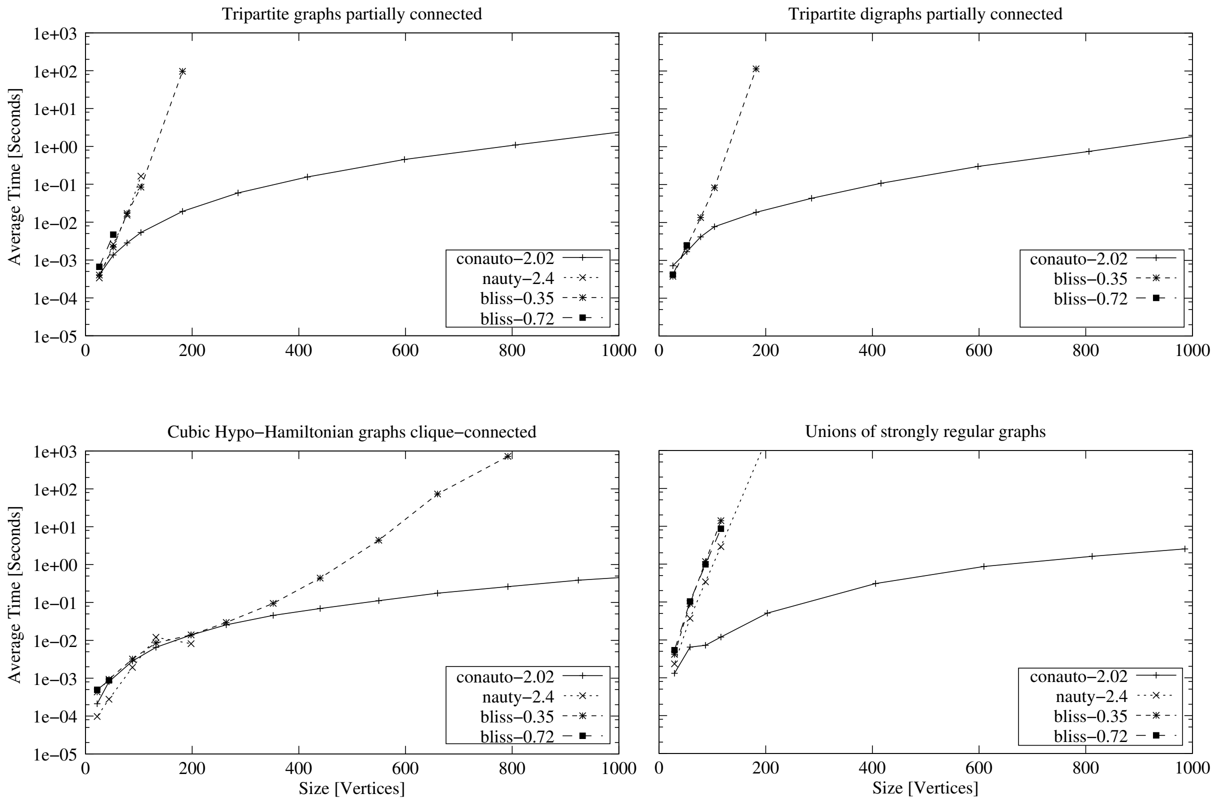

Figure 4.13: Performance for GI-testing with TNN, TND, CHH, and USR graph families.

We also show the performance for component-based graphs. For these graph families, we have chosen Tripartite digraphs partially connected (TND) and its undirected version (TNN), Cubic

Hypo-Hamiltonian clique-connected (CHH), and Unions of Strongly Regular graphs (USR). The results are shown in Figure 4.13. The results for the component-base graphs are spectacular. Conauto-2.0 is the only which is able to finish the whole experiments within the time limit, while the rest have exponential time or seems to have exponential behaviour (in case of bliss-0.72, it has a run-time internal error for this kind of graphs). It is observable that nauty is not able to finish with the smaller TND graph (26 vertices) within the time limit.

Finally we show the results for Complete (COM) and Paley Tournament (TOU) graph families in Figure 4.14. All the algorithms seems not to have exponential behaviour, being conauto-2.0 the fastest. However, conauto-2.0 is one order of magnitude faster than the closer algorithm (nauty) for COM graphs. And for TOU graphs, nauty and conauto-2.0 are in the same order of magnitude and both of them are one order of magnitude faster than bliss-0.72 and two orders of magnitude faster than bliss-0.35.

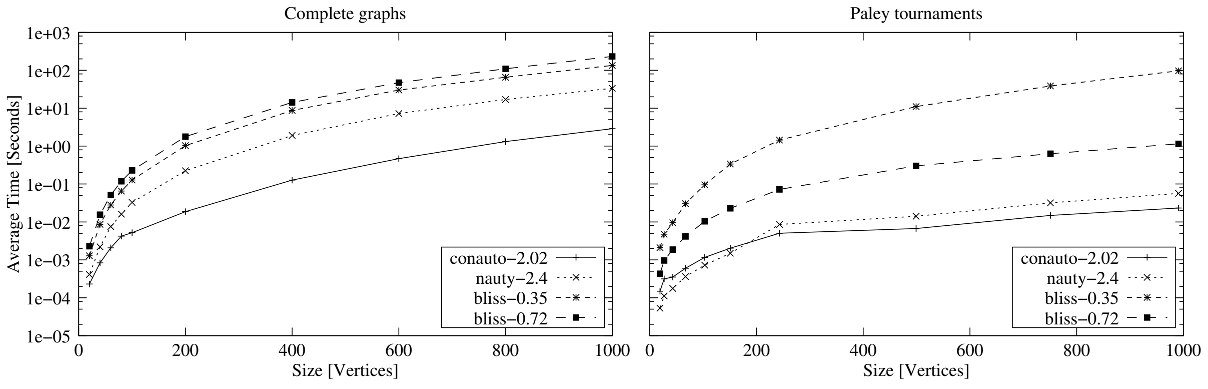

Figure 4.14: Performance for GI-testing with COM and TOU graph families.

4.4.2 Automorphism Group Computation Performance

We will compare the automorphism group computation of bliss (both 0.35 and 0.72 versions), Traces, saucy, and nauty, against our algorithm's. This experiment is intended to see how the size of the graphs affects the running time of automorphism group computation of these programs. Each point shown in the plots corresponds to the average running time of 100 executions with different instances of the corresponding graph. The CPU time limit of each execution has been set to 1,000 seconds for this experiment. Once an algorithm exceeds the time limit for any graph size instance, this point is not considered, and the execution stops.

The automorphism group computation performance for PG2 and AG2 graph families are shown in Figure 4.15. All the algorithms have quite similar behaviour (non exponential) except nauty and saucy. While nauty are clearly exponential time for both experiments, saucy just have exponential behaviour for AG2 graph family and not for PG2 graph family. Conauto-2.0 is not the fastest but it keeps a regular behaviour.

For component-based graph families, CHH is considered, whose results are shown in Figure 4.16, next to its normalized standard deviation. Traces, nauty and saucy have exponential behaviour, where nauty and saucy have additionally disparity in results. Although bliss-0.72 seems to be the fastest with the smaller graph sizes, the difference is quite little w.r.t. bliss-0.35 and conauto-2.0, and its behaviour changes for larger graphs, being slower than bliss-0.35 and conauto-2.0. This fact is also reflected in its normalized standard deviation. The figure shows that conauto-2.0

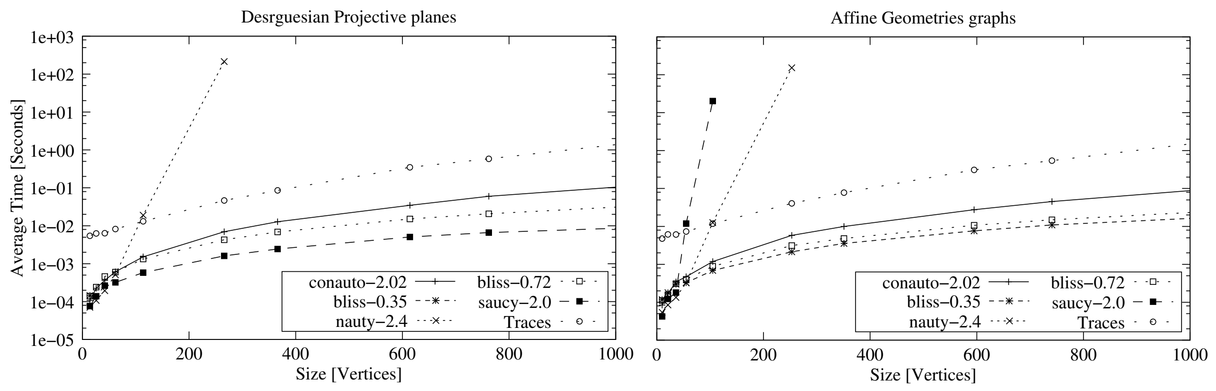

Figure 4.15: Performance for Aut. Group Computation with PG2 and AG2 graph families.

and bliss-0.35 are closer as the size grows, and it seems that conauto-2.0 may be faster for larger graphs.

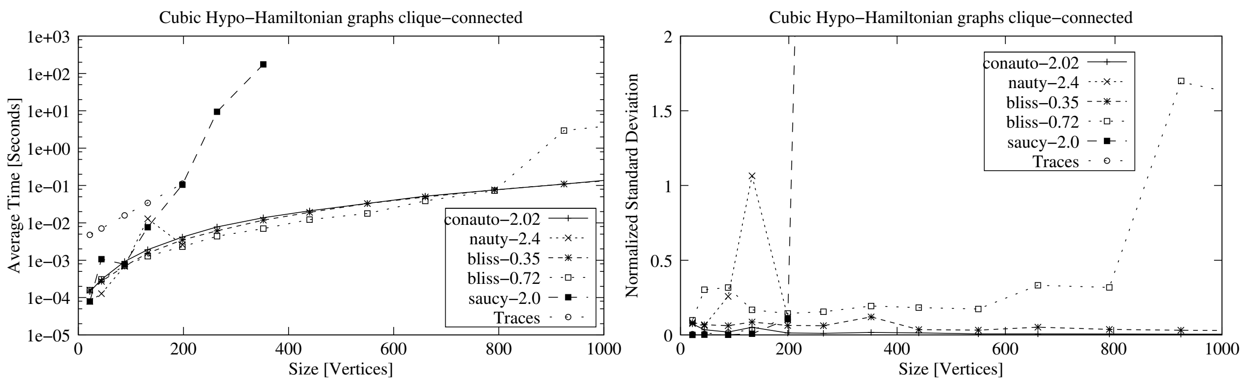

Figure 4.16: Performance for Aut. Group Computation with CHH graph family and its normalized standard deviation.

Other component-based graph families are TNN and USR, whose results are shown in Figure 4.17 alongside their normalized standard deviation. Clearly, conauto-2.0 is the fastest algorithm in order to deal with this kind of graphs, and hardly have time variation for different graph instances. For TNN graph families, saucy, nauty and bliss-0.72 have exponential behaviour. Although Traces has not seem to be exponential time, it is three orders of magnitude slower than conauto-2.0 and bliss-0.35. For USR, only saucy has a clear exponential behaviour. The rest of the algorithms have similar behaviour, but conauto-2.0 is one order of magnitude faster than the closer. And it is observable, that nauty has a huge deviation of results, in spite of not being exponential time.

For KEF graph family, results are shown in Figure 4.18, next to its normalized standard deviation. All algorithms have similar behaviour, which is not exponential, but Traces is the slower and the more irregular. This last fact, is reflected in the normalized standard deviation, where Traces have a huge time variation for different graph instances.

For regular graphs, LST has been considered, and its results, alongside its normalized standard deviation is shown in Figure 4.19. Although all algorithms seem to have the same behaviour, there is a huge variation of results, depending on the graph instance. It is observable in the normalized standard deviation figure, where nauty and conauto-2.0 are the fastest algorithms,

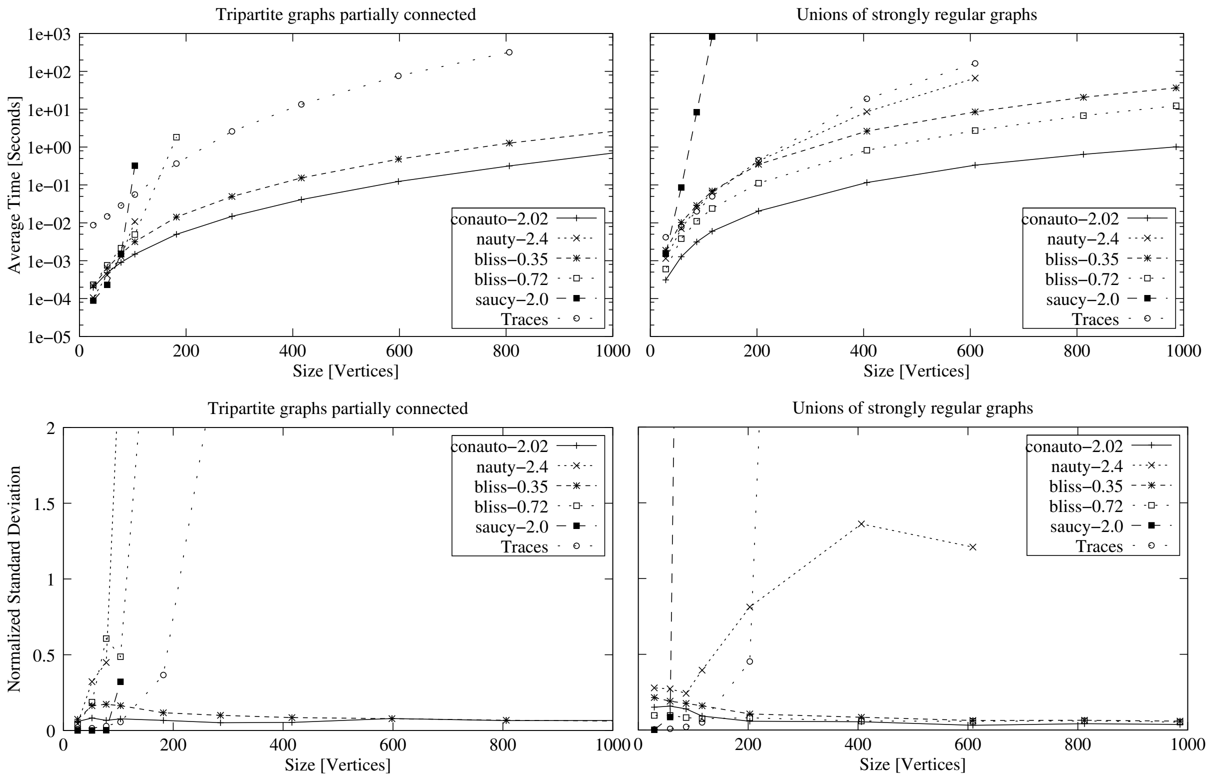

Figure 4.17: Performance for Aut. Group Computation with TNN and USR graph families and their normalized standard deviation.

but nauty behaviour tends to be more irregular.

For CMZ graph family, results are shown in Figure 4.20, next to its normalized standard deviation. Saucy is the fastest algorithm, and it is one order of magnitude faster than the closer, which is conauto-2.0. Moreover, this both algorithms have the more regular behaviour, what is observable in the normalized standard deviation figure.

HAD graph family results, of 1 and 2 orbits respectively, are shown in Figure 4.21, alongside its normalized standard deviation. In spite of the fact that conauto-2.0 is the fastest algorithm in both cases, all algorithms present an irregular behaviour depending on the graph instance. This fact is reflected in the normalized standard deviation figure.

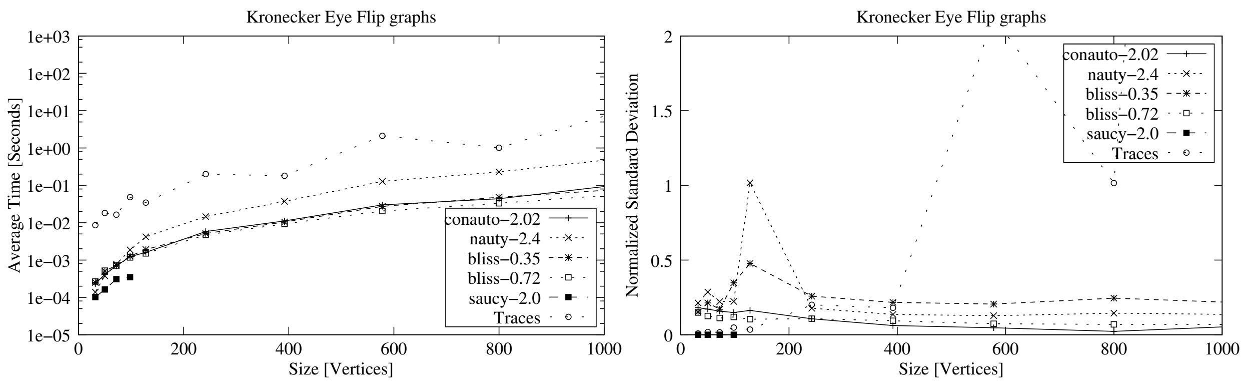

Figure 4.18: Performance for Aut. Group Computation with KEF graph family and its normalized standard deviation.

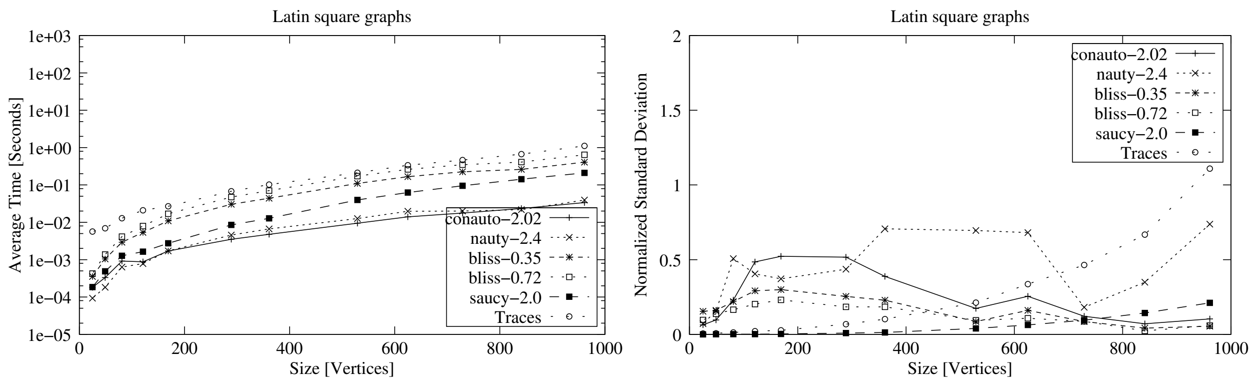

Figure 4.19: Performance for Aut. Group Computation with LST graph family and its normalized standard deviation.

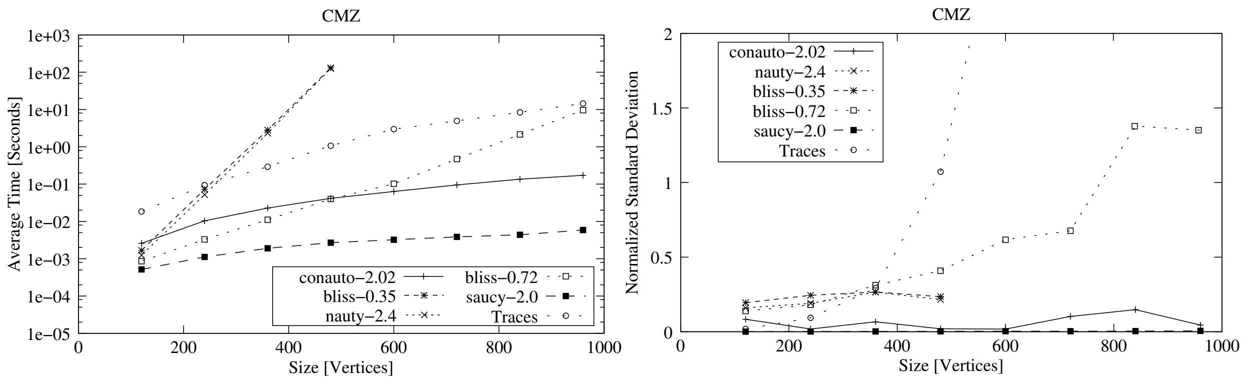

Figure 4.20: Performance for Aut. Group Computation with CMZ graph family and its normalized standard deviation.

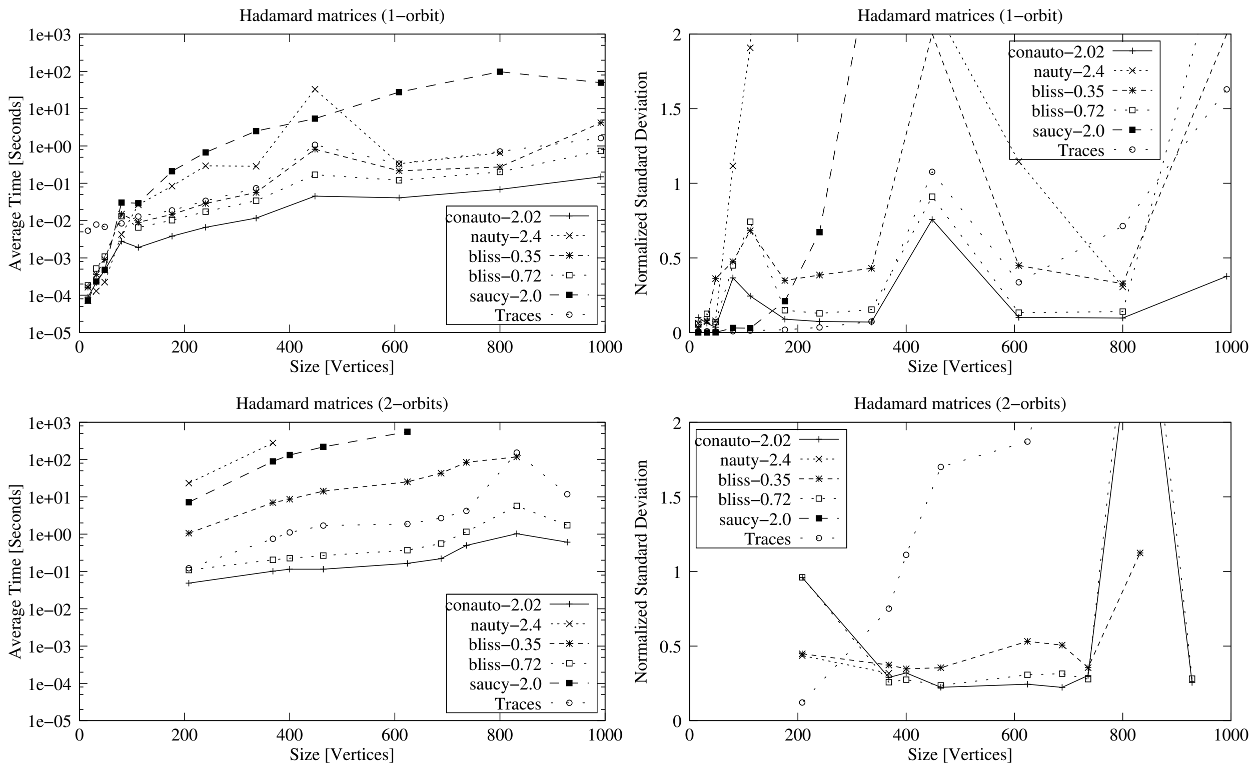

Figure 4.21: Performance for Aut. Group Computation with H1O and H2O graph family and their normalized standard deviation.

Chapter 5

Conclusions and Future Work

In this chapter, we summarize our conclusions and propose some extensions to our algorithm that may help improving its performance.

5.1 Conclusions

The main contributions of this work are the two theorems based on the concept of sub-partition. Although they are especially suited for graphs built from components which are uniformly connected, they have a much general field of application, since they do not relay on the detection of components. They allow early automorphism detection, and backjumping.

We have improved Algorithm *conauto* for graph isomorphism testing and added the capability of computing the automorphism group computation of a graph. This new algorithm *conauto-2.0* is not only faster than its predecessor for graph isomorphism testing, but also faster than the other worldwide-known algorithms based on canonical labeling, namely nauty, bliss, and Traces, for several graph families. Besides, when it is not the fastest, its behaviour seems to be close to the best algorithm for each case. Furthermore, the algorithm does not seem to be sensitive to permutations of the same graph. However, there are some graphs for which our algorithm is outperformed by Traces (e.g. Non-Desarguesian Projective Planes).

If we focus on the automorphism group computation, the results are similar, among all algorithms considered, for most families of graphs. In the particular case of graph families based on components, bliss-0.72 and conauto-2.0 usually outperform the rest of algorithms. However, bliss-0.72 seems to be much more sensitive to permutations. Different labelings of the same graph yield running times that may differ in several orders of magnitude for both automorphism group computation and canonical labeling. That is not the case of conauto-2.0, which has a much more regular behavior.

We have overcome the drawback of *conauto* for the point line graphs of Desarguesian projective planes with the new pivot sell selector algorithm for individualization. This cell selector also helps generating sub-partitions, what allows the application of Theorem 2.2 during the search for automorphisms.

To sum up, we have carried out almost all extensions proposed in [22], and other additional extensions. As a result, we have developed a faster algorithm, which can be used for isomorphism testing and for automorphism group computation.

5.2 Future Work

Canonical Labeling should be the next step in the development of this algorithm. In order to deal with the problem of canonical labeling, an idea proposed by Tener in [28] might help. He suggests that the canonical labeling should be easier to find having knowledge of the automorphism group in advance. Then, when vertex individualization is necessary to refine a partition, the pivot cell selector algorithm could choose the cell with the smallest number of orbits of the corresponding colored graph. Since we represent the automorphism group as a set of generators, an efficient way to compute orbits of subgroups will be necessary for such a pivot cell selector.

Junttila and Kaski [9], and López-Presa [22] independently suggested the idea of conflict propagation. Recording conflicts (non automorphisms) may help pruning bad paths in the search tree when a conflict does not match any of the recorded (valid) conflicts at some backtrack point. However, this is not all that can be done. When a conflict is found during the search for automorphisms, the location of the original pivot vertex may be recorded, so when some conflict is found later, only the vertices in the corresponding valid positions need to be tested. If all those vertices have already been tested unsuccessfully, then this path is a dead end. In other words, during the search for automorphisms, the unsuccessfully explored paths of the search tree bring us closer to either the successful path or the discarding of the current path. However, for proper detection of the pivot vertex location, an efficient automorphism group management is needed.

As it could be deduced, this improvements above, and almost any other sophisticated improvement that anybody may suggest, is conditioned to an efficient automorphism group management. The use of the Schreier-Sims algorithm for representing the automorphism group [25], by means of Leon's implementation [14], was adopted by Traces in [21]. However, this implementation has some limitations (such as the number of generators that it can handle). A different variant of the Schreier-Sims algorithm is being developed by Tener (called 'fssw': the fastest Schreier-Sims in the west) but not finished yet. However, our idea is not to use the Schreier-Sims representation, but another alternative what would be more compact. The Schreier-Sims representation of the automorphism group may need $O(n^3)$ space (for graphs on n vertices), what means using Gigabytes for graphs of one thousand vertices. Only $O(n)$ space should be necessary, since no more than $n-1$ generators are needed to represent the automorphism group of a graph.

Finally, improvements at implementation level may be done for efficiency in the performance of the algorithm, such as parallelization of the code for either computing the automorphism group of both graphs simultaneously or parallel searching for automorphisms at some level.

Bibliography

[1] Magdy S. Abadir and Jack Ferguson. An improved layout verification algorithm (LAVA). In *EURO-DAC '90: Proceedings of the conference on European design automation*, pages 391–395, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[2] Peter J. Cameron. Strongly regular graphs. In L.W. Beineke and R.J. Wilson, editors, *Topics in Algebraic Graph Theory*, pages 203–221. Cambridge University Press, 2004.

[3] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Graph matching applications in pattern recognition and image processing. In *IEEE International Conference on Image Processing*, volume 2, pages 21–24, Barcelona, Spain, September 2003. IEEE Computer Society Press.

[4] Jasper Cramwinckel, Erik Roijackers, Reinald Baart, Eric Minkes, Lea Ruscio, and David Joyner. GAP package GUAVA. Division of Engineering and Weapons at the U.S. Naval Academy, 2005. http://cadigweb.ew.usna.edu/∼wdj/gap/GUAVA/.

[5] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004.

[6] Jean-Loup Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial–time for molecular graphs. *Journal of chemical information and computer science*, 38:432–444, 1998.

[7] The GAP Group. GAP - Groups, Algorithms, Programming - a system for computational discrete algebra. Centre for Interdisciplinary Research in Computational Algebra, University of St. Andrews, Mathematical Institute, 2005. http://www-gap.mcs.st-and.ac.uk/.

[8] Tommi 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.

[9] Tommi Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19754-3_16.

[10] Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX*. SIAM, 2007.

[11] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2010.

[12] Donald L. Kreher and Douglas R. Stinson. *Combinatorial algorithms: generation, enumeration and search.* The CRC Press Series on Discrete Mathematics and its Applications. CRC Press LLC, Boca Raton, Florida, 1998.

[13] Felix Lazebnik and Andrew Thomason. Orthomorphisms and the construction of projective planes. *Mathematics of Computation*, 73(247):1547–1557, 2004.

[14] Jeffrey S. Leon. Partition backtrack programs. available at. ftp.math.uic.edu/pub/leon/partn.

[15] José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In Jan Vahrenhold, editor, *SEA*, volume 5526 of *Lecture Notes in Computer Science*, pages 221–232. Springer, 2009.

[16] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[17] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[18] Brendan D. McKay. The nauty page. Computer Science Department, Australian National University, 2010. http://cs.anu.edu.au/∼bdm/nauty/.

[19] Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 239–256. American Mathematical Society, Providence, Rhode Island, USA, 1997.

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

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

[22] José Luis López Presa. *Efficient Algorithms for Graph Isomorphism Testing.* Doctoral thesis, Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad Rey Juan Carlos, Madrid, Spain, March 2009. Available at http://www.diatel.upm.es/jllopez/tesis/thesis.pdf.

[23] Sven Reichard. Personal communication, April 2002.

[24] Massimo De Santo, Pasquale Foggia, Carlo Sansone, and Mario Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, 2003.

[25] Charles C. Sims. Computation with permutation groups. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC '71, pages 23–28, New York, NY, USA, 1971. ACM.

[26] Leonard H. Soicher. GRaph Algorithms using PErmutation groups – GRAPE 4.2 package for GAP 4.4. School of Mathematical Sciences, Queen Mary, University of London, 2005. http://www.maths.qmul.ac.uk/∼leonard/grape/.

[27] Daniel A. Spielman. Faster isomorphism testing of strongly regular graphs. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 576–584, New York, NY, USA, 1996. ACM Press.

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

[29] Greg Tener and Narsingh Deo. Attacks on hard instances of graph isomorphism. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 64:203–226, 2008.

[30] Gottfried Tinhofer and Mikhail Klin. Algebraic combinatorics in mathematical chemistry. Methods and algorithms III. Graph invariants and stabilization methods. Technical Report TUM-M9902, Technische Universität München, March 1999. http://www-lit.ma.tum.de/veroeff/quel/990.05005.pdf.

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[32] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1):59–68, 2003.

[33] Boris Weisfeiler, editor. *On construction and identification of graphs.* Number 558 in Lecture Notes in Mathematics. Springer, 1976.

[34] Jin yi Cai, Martin Frer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.