

UNIVERSIDAD REY JUAN CARLOS

ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN



Efficient Algorithms for
Graph Isomorphism Testing

Doctoral Thesis

José Luis López Presa

Licenciado en Informática

Madrid, 2009

©José Luis López Presa
2009
All Rights Reserved

Thesis submitted to the Departamento de Sistemas Telemáticos y
Computación in partial fulfillment of the requirements for the degree of

Doctor por la Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos
Madrid, Spain

DOCTORAL THESIS

Efficient Algorithms for Graph Isomorphism Testing

Author:

José Luis López Presa
Licenciado en Informática

Director:

Antonio Fernández Anta
Ph.D. in Computer Science

Madrid, Spain, 2009

Autorizo la defensa de la tesis doctoral *Efficient Algorithms for Graph Isomorphism Testing* cuyo autor es José Luis López Presa.

El director de la tesis

Antonio Fernández Anta

Móstoles, Madrid, 29 de diciembre de 2008

Acknowledgements

I would like to thank Prof. Antonio Fernández, who, prior to becoming my dissertation director, has been a friend of mine for many years. Thank you for your constant support and patience. This dissertation is the result of our intense joint work of several years.

I would also like to thank Dr. Jesús García López de la Calle for his help with some mathematical concepts, and Prof. Peter Cameron for his help with the generation of some families of strongly regular graphs for the benchmark.

I dedicate this dissertation to my family, my wife Janeth, and my son Andrés.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	3
1.3	Organization of the Dissertation	3
2	Related Work	5
2.1	Invariants	6
2.2	Direct Backtracking Algorithms	7
2.3	Canonical Labelings	7
3	Our Benchmark	9
3.1	Random Graphs	9
3.2	Regular Meshes	10
3.3	Miyazaki's Fürer Gadgets	10
3.4	Strongly Regular Graphs	12
3.4.1	Paley graphs	12
3.4.2	Triangular Graphs	12
3.4.3	Latin Square Graphs	13
3.4.4	Lattice Graphs	14
3.5	Unions of Tripartite Graphs	14
3.6	Unions of Strongly Regular Graphs	15
3.7	Point-Line graphs of Desarguesian Projective Planes	15
3.8	Graph Encoding	16
4	Definitions and Notation	17
4.1	Basic Definitions	17
4.2	Specific Notation and Definitions for the Algorithms	18
5	Basic Algorithm	23
5.1	Algorithm <i>sinauto</i>	24
5.1.1	Main Algorithm	24
5.1.2	Generation of a Sequence of Partitions	26
5.1.3	Search for a Sequence of Partitions Compatible with the Target	27
5.2	Example	28
5.2.1	Generation of the Sequence of Partitions for Graph G	29
5.2.2	Generation of the Sequence of Partitions for Graph H	30
5.2.3	Finding a Sequence of Partitions Compatible with the Target	31
5.3	Correctness of the Algorithm	34

5.4	Performance Evaluation	37
6	Using Automorphisms	43
6.1	Theoretical Background	43
6.2	Algorithm <i>conauto-v0</i>	46
6.2.1	Main Algorithm	46
6.2.2	Search for Automorphisms	47
6.2.3	Search for a Sequence of Partitions Compatible with the Target	49
6.3	Example	52
6.3.1	Search for Automorphisms in graph G	52
6.3.2	Search for Automorphisms in Graph H	55
6.3.3	Match Graphs G and H	59
6.4	Proof of correctness	61
6.5	Performance Evaluation	62
7	Detecting Components	67
7.1	Proving Theorem 7.1	68
7.2	Algorithm <i>conauto-v1</i>	73
7.2.1	Main Algorithm	73
7.2.2	Search for a Sequence of Partitions Compatible with the Target	73
7.3	Performance Evaluation	76
8	Further Automorphism Detection	79
8.1	Algorithm <i>conauto-v2</i>	80
8.1.1	Main Algorithm	80
8.1.2	Generation of a Sequence of Partitions	80
8.1.3	Extended Look for Automorphisms	80
8.1.4	Computing Orbits Applicable	81
8.1.5	Finding a Sequence of Partitions Compatible with the Target	85
8.2	Performance Evaluation	85
9	Complexity Analysis	91
9.1	Space Complexity	91
9.2	Time Complexity	92
9.2.1	Generation of a Sequence of Partitions	92
9.2.2	Finding a Match	93
9.2.3	Time Complexity for Undirected Graphs	93
10	Conclusions and Further Improvements	95
10.1	Conclusions	95
10.2	Future extensions	95
	Bibliography	99
	Resumen en Castellano	105

List of Figures

3.1	2D-meshes.	10
3.2	The graph Y_α	11
3.3	Miyazaki's graphs.	11
3.4	Miyazaki's switch graphs.	11
3.5	Tripartite graphs.	14
3.6	Point-line graph of the Desarguesian projective plane of order 2.	15
5.1	Sample graphs for isomorphism testing.	28
5.2	Equivalence of the final partitions.	33
5.3	Performance of sinauto with isomorphic randomly connected graphs.	38
5.4	Performance of sinauto with isomorphic 2D-meshes.	38
5.5	Performance of sinauto with isomorphic Miyazaki's Fürer gadgets.	39
5.6	Performance of sinauto with non-isomorphic Miyazaki's Fürer gadgets.	39
5.7	Performance of sinauto with Paley graphs.	39
5.8	Performance of sinauto with triangular and lattice graphs.	40
5.9	Performance of sinauto with Latin square graphs.	40
5.10	Performance of sinauto with isomorphic unions of tripartite graphs.	41
5.11	Performance of sinauto with non-isomorphic unions of tripartite graphs.	41
5.12	Performance of sinauto with unions of strongly regular graphs.	42
5.13	Performance of sinauto with point-line graphs of Desarguesian projective planes.	42
6.1	Adjacencies in G_{V^5} and G_{W^5}	52
6.2	Adjacencies in G_{V^5} and G_{W^5}	55
6.3	Adjacencies in H_{V^5} and H_{W^5}	56
6.4	Adjacencies in H_{V^5} and H_{W^5}	59
6.5	Performance of conauto-v0 with isomorphic randomly connected graphs.	63
6.6	Performance of conauto-v0 with isomorphic 2D-meshes.	63
6.7	Performance of conauto-v0 with isomorphic Miyazaki's Fürer gadgets.	64
6.8	Performance of conauto-v0 with non-isomorphic Miyazaki's Fürer gadgets.	64
6.9	Performance of conauto-v0 with Paley graphs.	64
6.10	Performance of conauto-v0 with triangular and lattice graphs.	65
6.11	Performance of conauto-v0 with Latin square graphs.	65
6.12	Performance of conauto-v0 with isomorphic unions of tripartite graphs.	65
6.13	Performance of conauto-v0 with non-isomorphic unions of tripartite graphs.	66
6.14	Performance of conauto-v0 with unions of strongly regular graphs.	66
6.15	Performance of conauto-v0 with point-line graphs of Desarguesian projective planes.	66
7.1	Partition of S_i^k into subsets $A_i, B_i, C_i,$ and D_i for all $i \in \{1, \dots, r\}$	69
7.2	Partition of A_i into subsets $A_i^c,$ and A_i^n	70

7.3	Adjacencies between E_i and E_j , and between E'_i and E'_j	71
7.4	Adjacencies between S_i^l and S_j^l , and between T_i^l and T_j^l	72
7.5	Performance of conauto-v1 with isomorphic unions of tripartite graphs.	76
7.6	Performance of conauto-v1 with non-isomorphic unions of tripartite graphs.	76
7.7	Performance of conauto-v1 with unions of strongly regular graphs.	77
8.1	Performance of conauto-v2 with isomorphic randomly connected graphs.	86
8.2	Performance of conauto-v2 with isomorphic 2D-meshes.	86
8.3	Performance of conauto-v2 with isomorphic Miyazaki's Fürer gadgets.	86
8.4	Performance of conauto-v2 with non-isomorphic Miyazaki's Fürer gadgets.	87
8.5	Performance of conauto-v2 with Paley graphs.	87
8.6	Performance of conauto-v2 with triangular and lattice graphs.	87
8.7	Performance of conauto-v2 with Latin square graphs.	88
8.8	Performance of conauto-v2 with isomorphic unions of tripartite graphs.	88
8.9	Performance of conauto-v2 with non-isomorphic unions of tripartite graphs.	88
8.10	Performance of conauto-v2 with unions of strongly regular graphs.	89
8.11	Performance of conauto-v2 with point-line graphs of Desarguesian projective planes.	89

List of Algorithms

1	Test whether G and H are isomorphic (<i>sinauto</i>).	24
2	Generate a sequence of partitions for a graph G	25
3	Find the best set $S_i^l \in \mathcal{S}^l$ to be used as a pivot.	26
4	Find a sequence of partitions compatible with the target (<i>sinauto</i>).	27
5	Test whether G and H are isomorphic (<i>conauto-v0</i>).	47
6	Look for automorphisms.	48
7	Apply Lemma 6.1.	49
8	Apply Lemma 6.2.	49
9	Try to generate a compatible sequence of partitions without backtracking.	50
10	Find a sequence of partitions compatible with the target (<i>conauto-v0</i>).	51
11	Test whether G and H are isomorphic (<i>conauto-v1</i>).	74
12	Find a sequence of partitions compatible with the target (<i>conauto-v1</i>).	75
13	Test whether G and H are isomorphic (<i>conauto-v2</i>).	81
14	Generate a sequence of partitions for a graph G (<i>conauto-v2</i>).	82
15	Look for more automorphisms.	83
16	Compute the orbit partition applicable at this point.	83
17	Find a sequence of partitions compatible with the target (<i>conauto-v2</i>).	84

Chapter 1

Introduction

1.1 Motivation

The Graph Isomorphism problem (GI) tests whether two given graphs are isomorphic or not. In other words, it asks whether there is a one-to-one mapping between the vertices of the graphs, preserving the arcs. This problem has been studied for decades by mathematicians, chemists and computer scientists, and is considered interesting from both the theoretical and the practical point of view, since it has applications in many fields, ranging from pattern recognition and computer vision [14, 32, 2, 51] to information retrieval [9], data mining [71], VLSI layout validation [24, 23, 58, 1, 59], or chemistry [43, 25, 68]. In this last field, for instance, a unique identifier (canonical numbering) for every chemical compound is needed. Examples of unique identifier generators are MOLGEN-CID [12] and SMILES [72]. The latter has been recently shown [57] to be unable to give a unique identifier for certain pairs of isomorphic graphs representing the same chemical component. GI and canonization are problems that go together.

In 1977, Reed and Corneil wrote an interesting survey on the GI problem, called “The Graph Isomorphism Disease” [60], to review the state of the art at that time. Later, in 1996, Fortin [28] updated the survey, focusing on practical isomorphism algorithms and hard graphs for isomorphism testing. Recently, Goldberg has summarized the most significant results in graph isomorphism in a brief, yet very comprehensive, survey [31].

While the Graph Isomorphism problem is clearly in NP, it has not been possible thus far to prove it to be in P nor NP-complete. Although there is no known polynomial-time algorithm for the general graph isomorphism problem, there is strong evidence that it is not NP-complete. Complexity theory results show that if GI were NP-complete, then the polynomial time hierarchy would collapse to its second level ($\Sigma_2^P = \Pi_2^P = \text{AM}$) [11, 64], and while no NP-complete problem has been shown to be polynomial-time equivalent for its decision and counting versions, GI has been [46]. Consequently, it has been conjectured that GI might lie strictly between P and NP-complete, as proposed by Karp [36].

Many attempts have been made to find upper complexity bounds for GI. Schöning, who introduced the notion of lowness in complexity theory, showed in [64] that GI is low for the class Σ_2^P . Subsequently, in [38] it was shown that it was also low for PP and C=P. Furthermore, it was shown that if GI were NP-complete, then the polynomial-time hierarchy would be low for the classes PP and C=P, obtaining more evidence that GI is not NP-complete. Later, in [4] it was shown that Graph Isomorphism was, in fact, in SPP (“Stoic PP”). That question had been left

open in [38].

For some restricted classes of graphs, GI is known to be solvable in polynomial time. A simple labeling algorithm can determine whether two trees are isomorphic in linear time [3]. In [35], a linear time isomorphism test is shown for planar graphs. Combinatorial methods yielded an algorithm for graphs of bounded genus that works in time $n^{O(g)}$ for genus $g \geq 1$ [26]. Group-theoretic methods led to polynomial time algorithms for graphs with bounded eigenvalue multiplicity [7] (an alternative algorithm for this class of graphs that uses only combinatorics and group theory was proposed by Fürer [29]). With considerably deeper use of group theory, Luks [44] obtained also a polynomial time algorithm for graphs of bounded valence (degree). More specific classes of graphs like geometric circulants have also been shown recently to be testable for isomorphism in polynomial time [56]. However, most of these algorithms, although polynomial in time, are not designed to be implemented or they are not of practical use for their hidden complexity.

Problems tend to arise when dealing with graphs that have very few automorphisms, but a high degree of regularity (cf. [39]), like certain families of strongly regular graphs (SRGs). Examples of SRGs are Latin square graphs and Steiner graphs. Miller [52] showed that the isomorphism of Latin square and Steiner triple system graphs is decidable in time $O(n^{\log n + O(1)})$. Subsequently, Spielman [67] obtained a time bound of $n^{O(n^{1/3} \log n)}$ for the general case of strongly regular graphs. Projective planes are amongst the hardest cases for graph isomorphism testing. Miller [52] showed that their isomorphism can be decided in $O(n^{\log \log n + O(1)})$ steps. Babai and Luks [8] generalized this result to λ -planes (for $\lambda = 1$ these are the projective planes), with bounded λ .

A major contribution in trying to establish the time complexity for the general case of graph isomorphism, due to Luks and Zemlyachenko, is their $\exp \sqrt{cn \log n}$ time bound (cf. [8, 75]). In [8] Babai and Luks describe an algorithm that compute canonical forms of general graphs in $\exp(n^{1/2 + o(1)})$, which is the best time bound up to date.

Although the isomorphism problem may be hard for some specific classes of graphs, a naive algorithm has been presented in [5] that is able to canonically label most graphs on n vertices in average linear time. In fact, it is shown there that the probability that a graph on n vertices can be canonically labeled with that algorithm is greater than $1 - \sqrt[3]{1/n}$ (for sufficiently large n). Subsequently, Babai and Kučera [6] improved this result obtaining a linear time canonical labeling algorithm with only $\exp(-cn \log n / \log \log n)$ probability of failure. Adding a depth-first search, they also derived a canonical labeling algorithm for all graphs with linear average time complexity. Although these results do not yield practical algorithms, they show how easy it is to test the isomorphism of random graphs. Recently, in [22], a linear time graph isomorphism algorithm is presented that works with even higher probability. All this has encouraged many researchers to look for efficient practical graph isomorphism algorithms.

Up to now, practical graph isomorphism algorithms have an exponential upper bound time complexity, although many of them work fine for many graph families. Among these, the most powerful currently available is Brendan McKay's *nauty* package [50]. Despite its impressive performance in most cases, there are some families of graphs that force it to run in exponential time [54]. The purpose of this thesis is to devise an algorithm that is comparable with *nauty* in the good cases and, at the same time, overcomes at least some of the deficiencies found in *nauty*.

1.2 Objectives

The aim of this dissertation is to propose a new algorithm that can be used to test (directed) graphs for (exact) isomorphism. The algorithm must be applicable to the general case, and must be *complete*, i.e. give a yes or no answer in every case. Starting with the fundamentals of the algorithm, we will develop incremental versions, showing the improvements attained with each new version. We must ensure that, for the families of graphs for which a version has good performance, the new version will run as fast as the previous one. Specifically, we will show that certain families of graphs that are hard for some version become easy or manageable with the new version.

We are going to impose additional restrictions on our algorithm. In particular, the memory space required to store the data structures needed to perform the tests must be at most $O(n^2)$ for graphs on n vertices. This restriction is natural for any practical algorithm since, otherwise, if the algorithm needed more memory, then it would only be useful for small graphs (hundreds of vertices). Then, our algorithm must work for graphs with thousands of vertices.

To test our algorithm, and as an additional contribution, we will construct a benchmark for testing GI practical algorithms. Then, 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 very hard graphs for all the GI algorithms we know of, the point-line graphs of Desarguesian Projective Planes. 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 evaluate our algorithms and compare them with other algorithms (mainly nauty) 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 [49] and vf2 [16] (an improved version of vf [15, 17]) as algorithms of reference. The choice of nauty is obvious, since it is the referent for practical graph isomorphism algorithms. In turn, vf2 is chosen because it uses a completely different approach from that of nauty. It is more of a traditional direct backtracking approach with what seem to be good heuristics to prune the search tree (at least for some classes of graphs).

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.

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows. In the next chapter, we compile previous related work conducted by other researchers, that is, we present previous graph isomorphism algorithms that use different approaches to the problem, showing their advantages and disadvantages.

In Chapter 3, 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.

Chapter 4 contains the formal definitions and the notation to be used in the rest of this document.

Chapter 5 describes the algorithm we start from. We put forward the main idea of our algorithms, which consists in analyzing the first graph and building a data structure that represents the structure of the graph, and then trying to build a new data structure from the second graph, which matches the first one. As it will be shown, this process yields an isomorphism between the two graphs. We also prove the correctness of the algorithm, and perform some tests with different families of graphs, showing the results obtained with this first algorithm.

In Chapters 6, 7, and 8 we show how, adding some more capabilities to our basic algorithm improves its performance for certain families of graphs, having a low impact on other families. Like in Chapter 5, we prove the correctness of the algorithms proposed, and compare the performance of the new versions with the other algorithms.

In Chapter 9, we study space and time complexity of our algorithm and show that it works in polynomial time with high probability.

Finally, in Chapter 10 we summarize the conclusions of the work described here, and how this work might be extended in the future.

Chapter 2

Related Work

In this chapter we present the different approaches to the problem of finding a “good in practice” algorithm to test graphs for isomorphism. We will not attempt to give a thorough description of each algorithm, but just a sketch of the main ideas on which they are based. Practical isomorphism algorithms may be roughly classified into two main categories. The first class uses a *direct* approach. They take the two graphs to be compared, and try to find an isomorphism between them directly with a classical depth-first backtrack algorithm, possibly using heuristics to prune the search tree. The second class uses a different approach. They take a single graph G and compute some function $C(G)$ which returns a *certificate* (canonical labeling) of the graph, such that for two graphs G and H , $C(G) = C(H)$ if and only if G and H are isomorphic. Once the certificates have been computed, comparing them is straightforward.

All the algorithms proposed thus far have an exponential worst case time complexity. Furthermore, as they get more sophisticated to handle difficult graphs, they become slower for the simple graphs. Therefore, some algorithms that give quick positive or negative answers for simple graphs, but can sometimes answer “I do not know” (*incomplete* algorithms), have been proposed in the literature. These algorithms have polynomial time complexity and may be very fast for easy graphs but do not solve the general problem. One such algorithm is RW [32], which is based on Markov Random Walks. It uses the steady state probability distribution of the Random Walk as a topological signature for the nodes. The problem arises when nodes get colliding signatures. Then, it is not able to give a positive or negative answer. The authors have proposed a modified version, RW2 [33], with a better matching rate, and yet, the same time complexity. They show that its matching rate is very good for random graphs, which is not surprising (remember the linear average time algorithm of Babai and Kučera [6]).

Other non-traditional approaches have been explored, like, for example, that of McGregor [47]. He proposes transforming the isomorphism problem into a constraint satisfaction problem, and then applying especially tuned constraint algorithms to solve the problem in the new domain. However, this approach does not seem intuitively inviting, since the constraint satisfaction problem is known to be NP-hard, and, as it has been previously mentioned, probably, graph isomorphism is not. As Fortin says, “a fundamental weakness of non-traditional approaches is that once the problem is transformed into another paradigm, it is no longer possible to apply group theory ideas to prune the search space. In some sense, the semantics of the data is being lost in the translation.” For other examples of non-traditional approaches see, for example, the references in [28, 39].

2.1 Invariants

A *graph invariant* is a function f such that, if applied to two isomorphic graphs G and H , then $f(G) = f(H)$ (the converse is not necessarily true). Therefore, an invariant imposes a necessary condition for isomorphism. If an invariant is both necessary and sufficient for isomorphism, it is said to be *complete*. A complete graph invariant is also called a *certificate*. A certificate for a tree can be computed in linear time, as shown in [3] and [40, Chapter 7]. However, there is no known complete graph invariant computable in polynomial time for the general case (otherwise, GI would be in P). Simple graph invariants are the number of vertices and the number of arcs of a graph. Other graph invariants are the determinant, the characteristic equation of its adjacency matrix, and the set of its roots (the *spectrum* of the graph), all of them computable in polynomial time, yet none of them complete.

A popular method of defining a certificate is to consider the adjacency matrix of the graph. Each possible permutation of the vertices of a graph on n vertices defines a different adjacency matrix. Concatenating the rows (or columns) of the adjacency matrix, we obtain a string that can be interpreted as an n^2 -bit number. Considering all the possible $n!$ permutations (i.e. all such numbers), it is possible to define a certificate of the graph as the smallest such number. This certificate induces an order on the vertices of the graph, which is called a *canonical labeling* of the graph. Unfortunately, this certificate is difficult to compute. In fact, this certificate will have as many leading zeroes as possible. Hence, the first k vertices in the canonical labeling are pairwise non-adjacent and k is as large as possible. Since these k vertices form a maximum clique in the complement of the graph, this certificate also solves the Maximum Clique problem, and this problem is known to be NP-complete. Since graph isomorphism is not likely to be NP-complete, computing this certificate may be harder than testing isomorphism by other means. What most isomorphism algorithms that use this invariant try to do is to generate only certain permutations, according to the structure of the graph, but not by any particular ordering of the vertices [48, 40, 39, 73]. This way, the certificate will not necessarily have as many leading zeroes as possible, but yet the ordering will be canonical and reproducible (i.e., the same) for any pair of isomorphic graphs. Unfortunately, there are cases where an exponential number of permutations are needed to compute these certificates.

A *vertex invariant* is a function f on a vertex, such that if there is an isomorphism I between G and H , for each $v \in G, v' \in H$, such that $I(v) = v'$, then $f(v) = f(v')$. The typical example of a vertex invariant is the degree of a vertex. If an isomorphism I maps v to v' , both vertices must have the same degree. Yet the converse is not necessarily true. This vertex invariant can be extended to a graph invariant: if G and H are isomorphic, then they must have the same number of vertices for each possible degree. Moreover, any isomorphism between them will map vertices with some degree to vertices with the same degree. This is useful, for example, when computing a canonical labeling of a graph. A previous classification of the vertices according to their degree will reduce the number of permutations to be considered to permutations among vertices with the same degree. Vertex classification is used by almost all graph isomorphism algorithms to prune the search space [48, 40, 39, 73].

Many other vertex invariants have been proposed in the literature. Some are applied directly by the algorithm, while others might be more useful if applied only under certain circumstances. Sophisticated invariants take long to compute, and most of the times, extending the search is faster than applying a sophisticated invariant. In this case, their application should be determined by the user. This is the case of nauty [49], which offers a number of invariants that can be applied at the user's request. Some of them are: twopaths (the number of vertices reachable along a path

of length two), distances (the number of vertices reachable at each distance), adjacencies (to mitigate the bad performance of nauty with digraphs), etc. A well known graph invariant is the characteristic polynomial (of the adjacency matrix) of the graph, but there are families of graphs that cannot be distinguished by their characteristic polynomial (i.e. they have the same spectra) [30]. However, eigenvalues and eigenvectors may help in distinguishing non-isomorphic graphs [43]. The distance matrix was used by Schmidt and Druffel [63] who proved that its use reduced considerably the amount of backtracking needed by their algorithm. Subsequently, Mittal [53] proposed a similar algorithm with better performance, but, since computing the distance matrix is quite costly, this idea has not been paid much more attention thereafter. King and Tzeng [37] proposed what they call the “probability propagation matrix”, whose computation may be parallelized to be used in multiprocessors. However, the use of invariants does not guarantee that the search space will be reduced (cf. [20]).

2.2 Direct Backtracking Algorithms

Perhaps the most natural way to tackle the graph isomorphism problem is to use direct backtracking. Usually, this class of algorithms classify the vertices (and maybe edges) according to some invariants to reduce the matching alternatives, but they rely mainly on a routine that explores the possible matchings of the vertices of one graph against the vertices of the other graph, backtracking if a branch does not reach a valid solution. To prune the search tree, they use feasibility heuristic functions that try to detect and discard unsuccessful branches as soon as possible. Examples of these class of algorithms are *vf* [17, 15] and *vf2* [16]. Also the previously mentioned algorithm by Schmidt and Druffel [63] fits in this class, along with other well known algorithms, like Ullman’s [69]. Other early examples may be found in [42] and [70].

Algorithms in this class are feasible for both graph and subgraph isomorphism (according to Ullman [69] it is due to the fact that they process both graphs at once), but tend to have high time complexity when the graphs being tested have many automorphisms (i.e. when they are highly symmetric). In this case, for positive tests, they benefit from the fact that when they find an isomorphism, they stop exploring the search space, while for negative tests, they have to go around the whole search tree, exploring branches that are automorphic to other branches that have been already discarded, considerably degrading the algorithm’s performance. The major drawback of these algorithms is that they do not detect automorphisms.

2.3 Canonical Labelings

In 1970, Corneil and Gotlieb [19] proposed an algorithm for graph isomorphism testing based on a conjecture that later Mathon disproved [45], giving examples of graphs that did not satisfy their conjecture. Thus their algorithm, which, if their conjecture held would be complete, was shown to be incomplete. However, they used an interesting partitioning and refining method, similar to that of Weisfeiler [73], which are the basis of most canonical labeling algorithms.

These algorithms do not compare one graph against the other directly, but they work separately on one graph first and on the other next, to generate canonical labelings of the graphs that may be compared directly. Detailed explanations on how to implement this class of algorithms may be found in [40, Chapter 7] and [39]. Currently, the most powerful canonical labeling program is

nauty [50], which is also currently the most widely used. While vf2 is faster than nauty in some cases [27], with hard graphs, nauty outperforms any other proposed isomorphism program.

The algorithm used by nauty (and most canonical labeling programs) is a backtracking algorithm that traverses a search tree looking for a canonical labeling, and, in the process, builds the automorphism group of the graph. Nauty starts with an initial vertex classification by their degree, that defines a partition of the vertices. From this partition, it performs successive refinements based on the adjacencies of the vertices of a cell of the partition with the vertices in all the cells of the partition. The basic refinement techniques used by nauty are described in [48] and are similar to the ones presented in [73]. When a partition is stable (i.e. it is not possible to refine it), a vertex individualization is done and the refinement process restarts. When the discrete partition (all the cells have size one) is reached, a leaf in the search tree is reached and a labeling and its associated candidate certificate is obtained. The nodes of the tree at which a vertex individualization has been done are the backtracking points that will be used to find other paths to a leaf, and therefore, obtain other labelings. If this new labeling induces the same candidate certificate, an automorphism is detected and memorized. This automorphism will be used to prune branches in the search tree. If the new labeling induces a smaller certificate, it is chosen as the new candidate certificate. If a new branch is at some point known to induce a bigger certificate, it is discarded. When all possible branches have been fully traversed or discarded, the certificate and the automorphism group have been computed.

This approach benefits from automorphisms to prune the search tree. Therefore, with highly symmetric graphs, this class of algorithms are very fast. Besides, they need the same time independently of whether the graphs being tested are isomorphic or not. However, they need to compute the whole automorphism group, what might be harder than other alternatives for isomorphism testing. This is particularly evident for a family of graphs defined in [74] and used by Miyazaki [54] to prove exponential lower time bounds for nauty. These graphs have bounded valence and, therefore, it should be possible to test them for isomorphism in polynomial time [44]. This family of graphs will be part of our benchmark.

Chapter 3

Our 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, Miyazaki's Fürer gadgets, different families of strongly regular graphs (lattice graphs, triangular graphs, Latin square graphs, and Paley graphs), unions of tripartite graphs, unions of strongly regular graphs, and Point-Line graphs of Desarguesian projective planes. Other families of graphs have been considered, but discarded since they did not discriminate among the algorithms, like cubic symmetric graphs or tournaments. Directed and undirected versions of the graphs will be considered when possible.

Since we intend to test our algorithm against *nauty* and *vf2*, and *vf2* can not handle disconnected graphs, all the graphs included in our benchmark will be connected. However, some families simulate disconnected graphs (graphs built from unions of small graphs). For each such family, the building process will be described in detail.

3.1 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 [62]. 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 [69]. This model fixes the probability η of an arc connecting two distinct vertices. For our benchmark, only the graphs in [62] 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 [62] 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 [62], it is shown how `vf2` is faster than `nauty` for these non-isomorphic random graphs.

3.2 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 included in the benchmark have also been taken from [62]. 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 3.1.(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` [62]. The subsequent version 2.2 of `nauty` performs much better, though it is still quite slower than `vf2`.

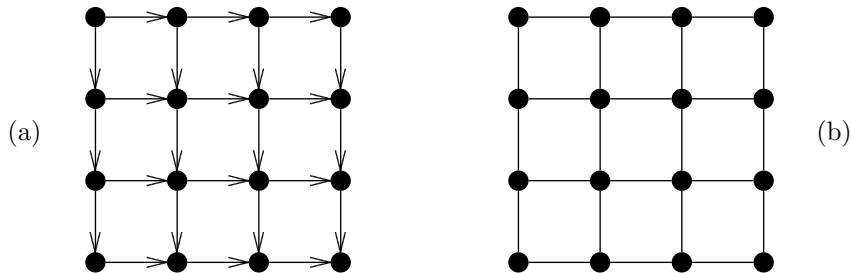


Figure 3.1: 2D-meshes.

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 3.1.(b).

3.3 Miyazaki's Fürer Gadgets

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 [54] 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.

The graphs are built in the following way: first, consider the undirected multigraph Y_α shown in Figure 3.2 with vertex set $V(Y_\alpha) = \{v_1, \dots, v_\alpha, w_1, \dots, w_\alpha\}$ 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) $E_3 = \{f_i : f_i = (v_i, w_i), 1 \leq i \leq \alpha\}$.

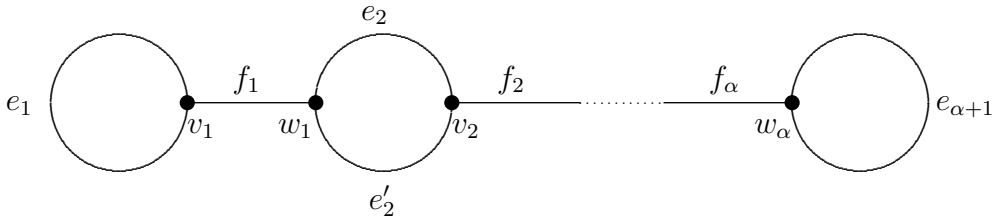


Figure 3.2: The graph Y_α .

Each node in Y_α has an incident cycle (one or two e -edges) and an incident bridge (edge f). Then, applying Fürer's construction (cf. [74]), we obtain a new (simple, not multi) graph, in which each vertex in the multigraph Y_α is substituted by a Fürer gadget. Thus, we obtain a 3-regular graph. Figure 3.3 shows the resulting graph, and its directed version.

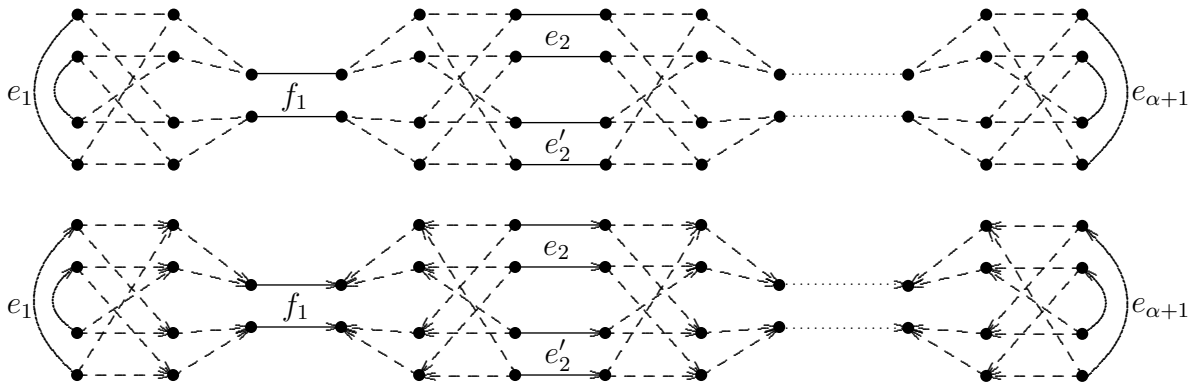


Figure 3.3: Miyazaki's graphs.

These are bounded valence graphs and, therefore, their canonical forms can be computed in polynomial time using the method of [44], 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. [54]). Examples of such graphs with twenty vertices are shown in Figure 3.4.

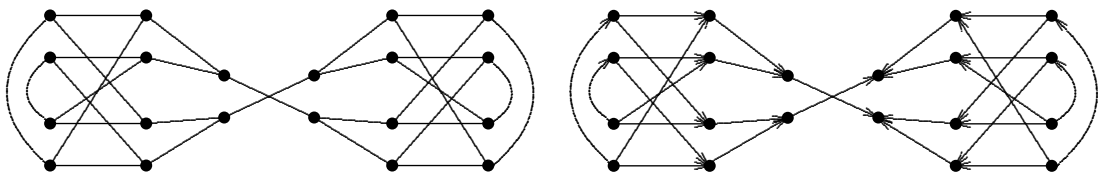


Figure 3.4: Miyazaki's switch graphs.

3.4 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.

3.4.1 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 [34] with Grape [66], 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.

3.4.2 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. [13]). Therefore, only positive tests will be considered for this family of graphs.

3.4.3 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}) , $i, j \in \{1, \dots, n\}$. 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. [67]).

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 [34] and GUAVA [21], 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.

3.4.4 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.

3.5 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 3.5. Their undirected versions are straightforward.

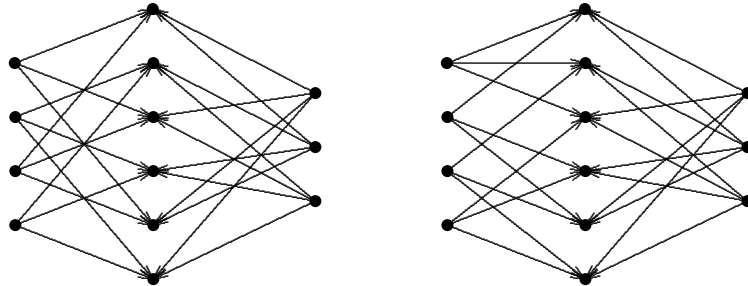


Figure 3.5: Tripartite graphs.

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.

3.6 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 [61].

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.

3.7 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 $P = \{p_1, \dots, p_n\}$ and line set $L = \{l_1, \dots, l_n\}$. 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 $i, j \in \{1, \dots, n\}$, $\{p_i, l_j\}$ is an edge of G if and only if $p_i \in l_j$. See for example the paper of Lazebnik and Thomason [41] 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 [55], 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 3.6 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.

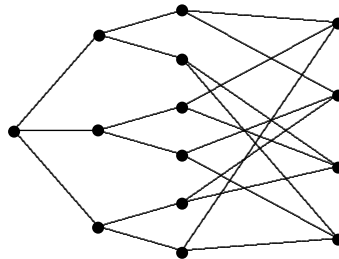


Figure 3.6: Point-line graph of the Desarguesian projective plane of order 2.

3.8 Graph Encoding

The graphs in our benchmark are stored each in one file in the same format used in the graph database in <http://amalfi.dis.unina.it/graph/> [62]. Each graph file is composed by a sequence of 16-bit words. The words are encoded in little-endian format. The first word is the number of vertices of the graph. Then, for each vertex, there is a word telling the number of arcs coming out of that vertex, followed by a sequence of words indicating the endpoints of these arcs. This encoding limits the number of vertices of a graph to tenths of thousands, but it is enough for our purpose; bigger graphs would need too much storage. Graph sizes will range from tenths of vertices to around one thousand vertices, and for each graph size, we will consider one hundred pairs of graphs.

The undirected version of the graphs is generated in the following way. For each existing arc between two vertices, the arc with the opposite direction is added if it did not already exist. This is the way undirected graphs are represented, according to the definition given in Chapter 4.

Chapter 4

Definitions and Notation

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

4.1 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 4.1 *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 $i \in \{1, 2, 3\}$.*

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.) 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 $i \in \{1, 2, 3\}$. Note that $ADeg(v, V_1, G) = Deg(v, G_{V_1 \cup \{v\}})$.

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

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

A permutation $\pi : V \rightarrow V$ 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 $\pi(v)$.

Definition 4.3 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 $(v, u) \in R_G \iff (\pi(v), \pi(u)) \in R_H$.

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.

4.2 Specific Notation and Definitions for the Algorithms

It will be necessary to introduce some specific notation to be used in the specification of our algorithms. Like other isomorphism testing algorithms, ours relies on vertex classification. This classification is performed by partitioning the vertex set using some vertex invariant, refining the successive partitions in an iterative process, and individualizing vertices when no refinement is possible and the vertices have not been completely classified yet. Let us start defining what a partition is, and the partition concatenation operation.

Definition 4.4 A partition of a set S is a sequence $\mathcal{S} = (S_1, \dots, S_r)$ of disjoint nonempty subsets of S such that $S = \bigcup_{i=1}^r S_i$. The sets S_i are called the cells of \mathcal{S} . The empty partition will be denoted by \emptyset .

Definition 4.5 Let $\mathcal{S} = (S_1, \dots, S_r)$ and $\mathcal{T} = (T_1, \dots, T_s)$ 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 $(S_1, \dots, S_r, T_1, \dots, T_s)$. 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. Let us formally define the degree partition of a graph.

Definition 4.6 Let $G = (V, R)$ be a graph. The degree partition of G , denoted $\text{DegreePartition}(G)$, is a partition $\mathcal{V} = (V_1, \dots, V_r)$ of V such that for all $i, j \in \{1, \dots, r\}$, $i < j$ implies $\text{Deg}(V_i, G) \succ \text{Deg}(V_j, G)$.

We will now introduce the concept of compatibility among partitions. It is rather intuitive to understand the compatibility of the degree partitions of two graphs. However, we will generalize the concept of compatibility to partitions in general.

Definition 4.7 Let $\mathcal{S} = (S_1, \dots, S_r)$ be a partition of the set of vertices of a graph $G = (V_G, R_G)$, and let $\mathcal{T} = (T_1, \dots, T_s)$ be a partition of the set of vertices of a graph $H = (V_H, R_H)$. \mathcal{S} and \mathcal{T} are said to be compatible under G and H respectively if $|\mathcal{S}| = |\mathcal{T}|$ (i.e. $r = s$), and for all $i \in \{1, \dots, r\}$, $|S_i| = |T_i|$ and $\text{Deg}(S_i, G) = \text{Deg}(T_i, H)$.

Partitions may be further refined by two means. 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.

Definition 4.8 Let $G = (V, R)$ be a graph, $v \in V$, $V_1 \subseteq V \setminus \{v\}$. The vertex partition of V_1 by v , denoted $\text{PartitionByVertex}(V_1, v, G)$, is a partition (S_1, \dots, S_r) of V_1 such that for all $i, j \in \{1, \dots, r\}$, $i < j$ implies $\text{ADeg}(S_i, \{x\}, G) > \text{ADeg}(S_j, \{x\}, G)$.

Definition 4.9 Let $G = (V, R)$ be a graph, and $\mathcal{S} = (S_1, \dots, S_r)$ a partition of V . Let $v \in S_x$ for some $x \in \{1, \dots, r\}$. The vertex refinement of \mathcal{S} by v , denoted $\text{VertexRefinement}(\mathcal{S}, v, G)$ is the partition $\mathcal{T} = \mathcal{T}_1 \circ \dots \circ \mathcal{T}_r$ such that for all $i \in \{1, \dots, r\}$, \mathcal{T}_i is the empty partition \emptyset if $\neg \text{HasLinks}(S_i, V, G)$ and $\text{PartitionByVertex}(S_i \setminus \{v\}, v, G)$ otherwise. S_x is the pivot set and v is the pivot vertex.

Definition 4.10 Let $G = (V, R)$ be a graph, and $V_1, V_2 \subseteq V$. The set partition of V_1 by V_2 , denoted $\text{PartitionBySet}(V_1, V_2, G)$, is a partition (S_1, \dots, S_r) of V_1 such that for all $i, j \in \{1, \dots, r\}$, $i < j$ implies $\text{ADeg}(S_i, V_2, G) \succ \text{ADeg}(S_j, V_2, G)$.

Definition 4.11 Let $G = (V, R)$ be a graph, and $\mathcal{S} = (S_1, \dots, S_r)$ a partition of V . Let $P = S_x$ for some $x \in \{1, \dots, r\}$ be a given pivot set. The set refinement of \mathcal{S} by P , denoted $\text{SetRefinement}(\mathcal{S}, P, G)$ is the partition $\mathcal{T} = \mathcal{T}_1 \circ \dots \circ \mathcal{T}_r$ such that for all $i \in \{1, \dots, r\}$, \mathcal{T}_i is the empty partition \emptyset if $\neg \text{HasLinks}(S_i, V, G)$ and $\text{PartitionBySet}(S_i, P, G)$ otherwise.

Once we have presented the possible partition refinements that may be applied to partitions, we can build sequences of partitions 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 (when a vertex refinement is performed with a pivot set with more than one vertex).

Definition 4.12 Let $G = (V, R)$ be a graph. A sequence of partitions for graph G is a tuple $(\mathcal{S}, \mathcal{R}, \mathcal{P})$, where $\mathcal{S} = (\mathcal{S}^0, \dots, \mathcal{S}^t)$, are the partitions themselves, $\mathcal{R} = (R^0, \dots, R^{t-1})$ indicate the type of refinement applied at each step, and $\mathcal{P} = (P^0, \dots, P^{t-1})$ choose the pivot set used for each refinement step, such that all the following statements hold:

1. For all $i \in \{0, \dots, t-1\}$, $R^i \in \{\text{VERTEX}, \text{SET}, \text{BACKTRACK}\}$, and $P^i \in \{1, \dots, |\mathcal{S}^i|\}$.

2. For all $i \in \{0, \dots, t-1\}$, let $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$, $V^i = \bigcup_{j=1}^{r_i} S_j^i$, $\mathcal{S}^{i+1} = (S_1^{i+1}, \dots, S_{r_{i+1}}^{i+1})$.
Then:
 - (a) For all $x \in \{1, \dots, r_{i+1}\}$, it exists $y \in \{1, \dots, r_i\}$, such that $S_x^{i+1} \subseteq S_y^i$ and $\text{HasLinks}(S_y^i, V^i, G)$.
 - (b) For all $x \in \{1, \dots, r_{i+1}-1\}$, $S_x^{i+1} \subseteq S_y^i$ implies $S_{x+1}^{i+1} \subseteq S_z^i$ where $y \leq z$.
 - (c) $R^i = \text{SET}$ implies $\mathcal{S}^{i+1} = \text{SetRefinement}(\mathcal{S}^i, S_{P^i}^i, G_{V^i})$.
 - (d) $R^i \neq \text{SET}$ implies $\mathcal{S}^{i+1} = \text{VertexRefinement}(\mathcal{S}^i, v, G_{V^i})$ for some $v \in S_{P^i}^i$.
3. Let $\mathcal{S}^t = (S_1^t, \dots, S_r^t)$, $V^t = \bigcup_{j=1}^r S_j^t$, then for all $x \in \{1, \dots, r\}$, $\text{NumLinks}(S_x^t, V^t, G) = 0$ or $|S_x^t| = 1$.

For convenience, for all $l \in \{1, \dots, t-1\}$, 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.

Note that, at each refinement step, from Statement 2b the relative order of the vertices in the old partition is preserved in the new one, and the vertices with no links in a partition, are discarded for the following one. This way, it is possible to define a (partial) order of the vertices of a graph, induced by a sequence of partitions, in the following way:

Definition 4.13 Let $\mathbf{Q} = (\mathbf{S}, \mathbf{R}, \mathbf{P})$ be a sequence of partitions for graph $G = (V, R)$ where $\mathbf{S} = (\mathcal{S}^0, \dots, \mathcal{S}^t)$, $\mathbf{R} = (R^0, \dots, R^{t-1})$, and $\mathbf{P} = (P^0, \dots, P^{t-1})$. For all $i \in \{0, \dots, t\}$, let $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$, and $V^i = \bigcup_{j=1}^{r_i} S_j^i$. The (partial) order induced by \mathbf{Q} on the set of vertices V is that which satisfies the following conditions:

1. For all $i \in \{0, \dots, t-1\}$, all the vertices in $V^i \setminus V^{i+1}$ precede all the vertices in V^{i+1} .
2. For all $i \in \{0, \dots, t-1\}$, $\mathcal{S}^{i+1} = \text{VertexRefinement}(\mathcal{S}^i, v, G_{V^i})$ implies that v precedes all the vertices in $V^i \setminus V^{i+1}$.
3. For all $x, y \in \{1, \dots, r\}$ such that $\text{NumLinks}(S_x^i, V^i, G_{V^i}) = 0$ and $\text{NumLinks}(S_y^i, V^i, G_{V^i}) = 0$, $x < y$ implies that all the vertices in S_x^i precede all the vertices in S_y^i .
4. Let $\mathcal{S}^t = (S_1^t, \dots, S_r^t)$, then for all $x, y \in \{1, \dots, r\}$, $x < y$ implies all the vertices in S_x^t precede all the vertices in S_y^t .

Note that there are pairs of vertices that are not ordered by this partial order (those that belong to the same cell, when they are discarded for having no remaining links). These vertices are therefore interchangeable since they are adjacent to exactly the same vertices in the same way. When referring to the order induced by a sequence of partitions \mathbf{Q} , we mean any total order that respects the (partial) order defined. Let $\leq_{\mathbf{Q}}$ be any such order. We will denote as $\omega_{\mathbf{Q}}(i)$ the i^{th} vertex with respect to $\leq_{\mathbf{Q}}$ (i.e. $|\{v : v \in V \wedge v \leq_{\mathbf{Q}} \omega_{\mathbf{Q}}(i)\}| = i - 1$).

Now, we will introduce the concept of compatibility among two sequences of partitions. Finding compatible sequences of partitions for two graphs leads to finding an isomorphism between them, as it will be proved in Section 5.3. Each sequence of partitions induces an order on the vertices of its corresponding graph. Mapping the i^{th} vertex in one order to the i^{th} vertex in the other order we get such isomorphism. This will be proved in Section 5.3 as well.

Definition 4.14 Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two graphs. Let $\mathbf{Q}_G = (\mathbf{S}_G, \mathbf{R}_G, \mathbf{P}_G)$, and $\mathbf{Q}_H = (\mathbf{S}_H, \mathbf{R}_H, \mathbf{P}_H)$ be two sequences of partitions for graphs G and H respectively. \mathbf{Q}_G and \mathbf{Q}_H are said to be compatible sequences of partitions if they satisfy all the following:

1. $|\mathbf{S}_G| = |\mathbf{S}_H| = t$, $|\mathbf{R}_G| = |\mathbf{R}_H| = t - 1$, $|\mathbf{P}_G| = |\mathbf{P}_H| = t - 1$.
2. Let $\mathbf{R}_G = (R_G^0, \dots, R_G^{t-1})$, and $\mathbf{R}_H = (R_H^0, \dots, R_H^{t-1})$, then for all $i \in \{0, \dots, t-1\}$, $R_G^i = R_H^i$.
3. Let $\mathbf{P}_G = (P_G^0, \dots, P_G^{t-1})$, and $\mathbf{P}_H = (P_H^0, \dots, P_H^{t-1})$, then for all $i \in \{0, \dots, t-1\}$, $P_G^i = P_H^i$.
4. Let $\mathbf{S}_G = (\mathcal{S}^0, \dots, \mathcal{S}^t)$, $\mathbf{S}_H = (\mathcal{T}^0, \dots, \mathcal{T}^t)$, then:
 - (a) For all $i \in \{0, \dots, t\}$, $|\mathcal{S}^i| = |\mathcal{T}^i|$.

- (b) For all $i \in \{0, \dots, t\}$, let $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$, $\mathcal{T}^i = (T_1^i, \dots, T_{r_i}^i)$, let $V_G^i = \bigcup_{j=1}^{r_i} S_j^i$, and $V_H^i = \bigcup_{j=1}^{r_i} T_j^i$, then \mathcal{S}^i and \mathcal{T}^i are compatible under $G_{V_G^i}$ and $H_{V_H^i}$ respectively.
- (c) Let $\mathcal{S}^t = (S_1^t, \dots, S_r^t)$, $\mathcal{T}^t = (T_1^t, \dots, T_r^t)$, then for all $x, y \in \{1, \dots, r\}$, $ADeg(S_x^t, S_y^t, G) = ADeg(T_x^t, T_y^t, H)$.

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 chapter), is the number of refinement steps where backtracking will be needed.

Definition 4.15 Let $\mathbf{Q} = (\mathcal{S}, \mathcal{R}, \mathcal{P})$ be a sequence of partitions, and let $\mathbf{R} = (R^0, \dots, R^{t-1})$. The amount of backtracking induced by \mathbf{Q} is $BacktrackAmount(\mathbf{Q}) = |\{i : i \in \{1, \dots, t-1\} \wedge R^i = \text{BACKTRACK}\}|$.

Chapter 5

Basic Algorithm

In this dissertation we are going to explore a new approach to graph isomorphism testing. As we said in Chapter 2, direct backtracking algorithms benefit from the fact that once they have found an isomorphism between the graphs, they can stop the search. We also mentioned that computing a canonical labeling implies going round every non isomorphic ordering of the vertices, looking for the one that best fits the canonicalization criterion. One way to prune this search is using a feasibility function to detect when a path leads to a worse solution than the best found yet, i.e. the value of the certificate induced by the new ordering is smaller than the one induced by the candidate certificate found so far. However, if the graph is very regular, it will not be until a long path has been followed, that this will be detected. Additionally, a canonical labeling algorithm, like nauty, learns from the discovered automorphisms, and prunes automorphic paths in the search tree. Our algorithm tries to capture the advantages of these approaches without suffering from their disadvantages, as follows:

- 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, and then, try to produce an equivalent ordering for the other graph, using the information obtained from the first graph to aid in the search.
- We use partition (vertex or set) refinement to classify the vertices so that the correspondence between the vertices of one graph and the vertices of the other graph is as much constrained as possible, to reduce the amount of backtracking necessary to try all feasible paths in the search space.
- 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 [73, 48, 1]). However, this can be quite costly. Therefore, we do things the other way round; i.e. we take 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 (though limited to $O(n^2)$ as required), 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 requirements of the algorithm.
- We look for automorphisms, but only those that can be found without backtracking.

(Otherwise, finding automorphisms would be equivalent to finding the canonical form of the graph.)

5.1 Algorithm *sinauto*

First, we will present a simple version of the algorithm, where automorphism detection is not performed, and then, we will add successive improvements that discover and make use of automorphisms, and apply other techniques to prune the search space.

5.1.1 Main Algorithm

Algorithm 1, *AreIsomorphic*, receives two graphs G and H as parameters and returns TRUE if both graphs are isomorphic, and FALSE if they are not.

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

```

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    $\mathcal{D}_G \leftarrow \text{DegreePartition}(G)$ 
6    $\mathcal{D}_H \leftarrow \text{DegreePartition}(H)$ 
7   if  $\mathcal{D}_G$  and  $\mathcal{D}_H$  are not compatible under  $G$  and  $H$  respectively then
8     return FALSE
9   else
10     $Q_G \leftarrow \text{GenerateSequenceOfPartitions}(G, \mathcal{D}_G)$ 
11     $Q_H \leftarrow \text{GenerateSequenceOfPartitions}(H, \mathcal{D}_H)$ 
12    if  $\text{BacktrackAmount}(Q_G) \leq \text{BacktrackAmount}(Q_H)$  then
13      return  $\text{Match}(0, G, H, Q_G, \mathcal{D}_H)$ 
14    else
15      return  $\text{Match}(0, H, G, Q_H, \mathcal{D}_G)$ 
16    end if
17  end if
18 end if

```

This algorithm tests first if both graphs have the same number of vertices and arcs. It is easy to see that this 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 *DegreePartition* 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 sequences of partitions Q_G and Q_H for graphs G and H respectively. The one that has less backtracking points 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 4, *Match*. If it succeeds, G and H are isomorphic. Otherwise, they are not.

Note that the partition with larger number of backtracking points (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 backtracking points). Besides, the sequences of partitions may be very different. Therefore, it is worth generating a sequence of partitions for each graph, and then choosing the one which will generate less backtracking points during the search process. This way, we also obtain an algorithm that takes the same time, independently of the order of its parameters.

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  $\mathcal{S}^l = (S_1^l, \dots, S_{r_l}^l)$ ,  $V^l = \bigcup_{j=1}^{r_l} S_j^l$ 
3  $\mathcal{S}^0 \leftarrow \mathcal{D}$ 
4 for each  $S_x^0 \in \mathcal{S}^0$  do
5    $Valid(S_x^0) \leftarrow (|S_x^0| > 1) \wedge HasLinks(S_x^0, V^0, G)$ 
6 end for
7  $l \leftarrow 0$ 
8 while  $\exists S_x^l \in \mathcal{S}^l : (|S_x^l| > 1) \wedge HasLinks(S_x^l, V^l, G)$  do
9    $P^l \leftarrow IndexBestPivot(\mathcal{S}^l, G)$ 
10  if  $|S_{P^l}^l| = 1$  then
11     $R^l \leftarrow VERTEX$ 
12     $v \leftarrow$  the only vertex in  $S_{P^l}^l$ 
13     $\mathcal{S}^{l+1} \leftarrow VertexRefinement(\mathcal{S}^l, v, G_{V^l})$ 
14  else
15     $success \leftarrow FALSE$ 
16    while  $Valid(S_{P^l}^l) \wedge \neg success$  do
17       $Valid(S_{P^l}^l) \leftarrow FALSE$ 
18       $R^l \leftarrow SET$ 
19       $\mathcal{S}^{l+1} \leftarrow SetRefinement(\mathcal{S}^l, S_{P^l}^l, G_{V^l})$ 
20       $success \leftarrow \exists S_x^{l+1}, S_{x+1}^{l+1} : S_x^{l+1}, S_{x+1}^{l+1} \subset S_y^l$  for some  $S_y^l \in \mathcal{S}^l$ 
21      if  $\neg success$  then
22         $P^l \leftarrow IndexBestPivot(\mathcal{S}^l, G)$ 
23      end if
24    end while
25    if  $\neg success$  then
26       $R^l \leftarrow BACKTRACK$ 
27       $v \leftarrow$  any vertex in  $S_{P^l}^l$ 
28       $\mathcal{S}^{l+1} \leftarrow VertexRefinement(\mathcal{S}^l, v, G_{V^l})$ 
29    end if
30  end if
31   $l \leftarrow l + 1$ 
32  for each  $S_x^l \in \mathcal{S}^l$  do
33    -- let  $S_x^l \subseteq S_y^{l-1}$ ,  $S_y^{l-1} \in \mathcal{S}^{l-1}$ 
34     $Valid(S_x^l) \leftarrow HasLinks(S_x^l, V^l, G) \wedge (Valid(S_y^{l-1}) \vee (|S_x^l| < |S_y^{l-1}|))$ 
35  end for
36 end while
37  $t \leftarrow l$ 
38  $S \leftarrow (S^0, \dots, S^t)$ ;  $R \leftarrow (R^0, \dots, R^{t-1})$ ;  $P \leftarrow (P^0, \dots, P^{t-1})$ 
39 return  $(S, R, P)$ 

```

5.1.2 Generation of a Sequence of Partitions

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.

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

IndexBestPivot(\mathcal{S}^l, G) : integer

```

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

```

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 a singleton cell with remaining links (as many as possible), if such a cell exists. This corresponds to Case 1 above.
- If there is not such a cell, it chooses one among the smallest valid cells (if there is one such cell). This corresponds to Case 2 above.
- If there are no valid cells, it chooses one among the smallest cells with remaining links

with the vertices in the partition. This corresponds to Case 3 above.

The pivot set 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 backtracking point, which is the algorithm's worst option. Finally, a smaller cell is preferred, since it will be faster to process than a bigger one.

5.1.3 Search for a Sequence of Partitions Compatible with the Target

In order to minimize the backtracking of Algorithm 4, *Match*, the target sequence of partitions will be that, among Q_G and Q_H , with less backtracking points. This does not guarantee that less backtracking will be needed to find a valid match (if it exists), but helps in practice.

Algorithm 4 Find a sequence of partitions compatible with the target (*sinauto*).

```

Match( $l, G, H, Q_G, T$ ) : boolean
1 -- let  $Q_G = (S, R, P)$ , let  $S = (S^0, \dots, S^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $P = (P^0, \dots, P^{t-1})$ 
2 -- let  $S^l = (S_1^l, \dots, S_{r_l}^l)$ ,  $V^l = \bigcup_{j=1}^{r_l} S_j^l$ , for all  $l \in \{0, \dots, t\}$ 
3 -- let  $T = (T_1, \dots, T_r)$ ,  $W = \bigcup_{j=1}^r T_j$ , since  $S^l$  and  $T$  are compatible,  $r = r_l$ 
4 if  $l = t$  then
5      $success \leftarrow \forall x, y \in \{1, \dots, r\}, ADeg(S_x^t, S_y^t, G) = ADeg(T_x, T_y, H)$ 
6 else
7      $X \leftarrow T_{P^l}$ 
8     if  $R^l = \text{BACKTRACK}$  then
9         repeat
10             $v \leftarrow$  any vertex in  $X$ 
11             $X \leftarrow X \setminus \{v\}$ 
12             $T' \leftarrow \text{VertexRefinement}(T, v, H_W)$ 
13            -- let  $T' = (T'_1, \dots, T'_{r'})$ ,  $W' = \bigcup_{j=1}^{r'} T'_j$ 
14            if  $S^{l+1}$  and  $T'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
15                 $success \leftarrow \text{Match}(l+1, G, H, Q_G, T')$ 
16            else
17                 $success \leftarrow \text{FALSE}$ 
18            end if
19            until  $X = \emptyset \vee success$ 
20        else
21            if  $R^l = \text{VERTEX}$  then
22                 $v \leftarrow$  the only vertex in  $X$ 
23                 $T' \leftarrow \text{VertexRefinement}(T, v, H_W)$ 
24            else (i.e.  $R^l = \text{SET}$ )
25                 $T' \leftarrow \text{SetRefinement}(T, X, H_W)$ 
26            end if
27            -- let  $T' = (T'_1, \dots, T'_{r'})$ ,  $W' = \bigcup_{j=1}^{r'} T'_j$ 
28            if  $S^{l+1}$  and  $T'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
29                 $success \leftarrow \text{Match}(l+1, G, H, Q_G, T')$ 
30            else
31                 $success \leftarrow \text{FALSE}$ 
32            end if
33        end if
34    end if
35    return  $success$ 

```

Algorithm *Match* is used to look for a sequence of partitions \mathcal{Q}_H for a graph H , compatible with the target \mathcal{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 with a partition \mathcal{T} that is compatible with \mathcal{S}^l , and then it generates a new partition \mathcal{T}' using the same refinement used to generate \mathcal{S}^{l+1} from \mathcal{S}^l , with the corresponding pivot set T_{Pl} . If partitions \mathcal{T}' and \mathcal{S}^{l+1} are compatible, then it makes a recursive call to process the new partition. Otherwise, it returns FALSE.

Backtracking is controlled by parameter l , which determines at which step in the process of generating the sequence of partitions for graph H the algorithm is. If at some step, the refinement P^l is VERTEX or SET, then the corresponding refinement is done for partition \mathcal{T} and a new partition \mathcal{T}' is generated. If this new partition is compatible with \mathcal{S}^{l+1} , then a recursive call is made to *Match*. If it succeeds, an isomorphism has been found. Otherwise, this is a dead-end path, and the algorithm backtracks (returns FALSE) until it reaches a node in the search tree with a feasible alternative.

A step with $R^l = \text{BACKTRACK}$ establishes a point where a choice was made in the generation of the target sequence of partitions. One of the vertices in the pivot set was chosen as the pivot vertex for a vertex refinement. That means that every vertex in the corresponding pivot set T_{Pl} must be tried, since the correspondence with the original pivot vertex might be found with any of the vertices in the pivot set T_{Pl} . This is achieved with the loop in lines 9-19. If one choice succeeds, the isomorphism has been found. Otherwise, other options are tried until no more vertices are available. Then, the algorithm returns FALSE, so that other options at a previous level should be tried. When every possible path has been tried unsuccessfully, the algorithm returns FALSE indicating that graphs G and H are not isomorphic.

5.2 Example

In this section we will show how the algorithm works, for the graphs of Figure 5.1. They are simple regular graphs which have some automorphisms but are not vertex-transitive. Since the graph is regular but not vertex-transitive, backtracking will be necessary, at least for the first refinement.

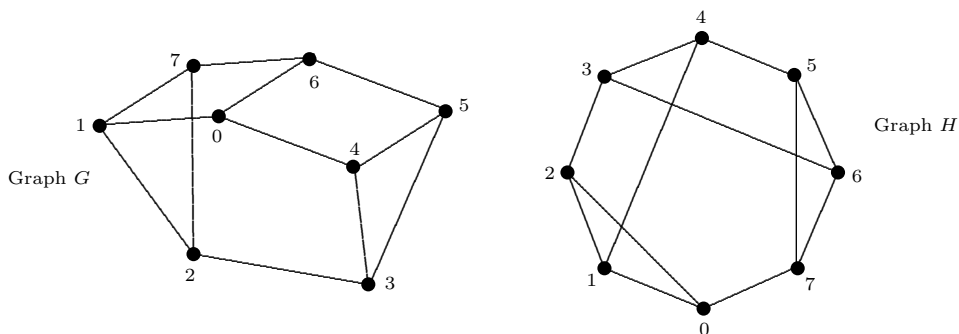


Figure 5.1: Sample graphs for isomorphism testing.

The algorithm starts comparing the number of vertices in graphs G and H . Both have 8 vertices and 12 edges. Hence, it computes the degree partitions \mathcal{D}_G and \mathcal{D}_H . Since all vertices in both graphs have degree $(3, 0, 0)$, we get:

For graph G : $\mathcal{D}_G = (D_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\})$ with $Deg(D_1, G) = (3, 0, 0)$
For graph H : $\mathcal{D}_H = (E_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\})$ with $Deg(E_1, H) = (3, 0, 0)$

Once the degree partitions have been verified to be compatible (both have one cell and $Deg(D_1, G) = Deg(E_1, H)$), a sequence of partitions will be generated for each graph, applying algorithm *GenerateSequenceOfPartitions*.

5.2.1 Generation of the Sequence of Partitions for Graph G

Initially, algorithm *GenerateSequenceOfPartitions* sets $\mathcal{S}^0 = \mathcal{D}_G = (\{0, 1, 2, 3, 4, 5, 6, 7\})$, and marks its only cell as not valid. Then it starts generating subsequent partitions. Since there is no singleton cell, nor a valid cell, R^0 is set to BACKTRACK and a vertex refinement is performed using a vertex in cell S_1^0 as the pivot vertex (hence, $P^0 = 1$). For simplicity, we will choose the first vertex in the cell, though any of them could be chosen randomly. Taking vertex 0 as the pivot vertex, we obtain $\mathcal{S}^1 = (S_1^1, S_2^1)$, where $V^1 = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} S_1^1 &= \{1, 4, 6\} & \text{with } ADeg(\{1, 4, 6\}, \{0\}, G) &= (1, 0, 0) & \text{and } Valid(S_1^1) &= \text{TRUE} \\ S_2^1 &= \{2, 3, 5, 7\} & \text{with } ADeg(\{2, 3, 5, 7\}, \{0\}, G) &= (0, 0, 0) & \text{and } Valid(S_2^1) &= \text{TRUE} \end{aligned}$$

Both cells in \mathcal{S}^1 are valid since they are new. Therefore, they must be tried for a set refinement. Algorithm *IndexBestPivot* selects them in increasing size order. Hence, it chooses S_1^1 ($P^1 = 1$) as the pivot set. Applying a set refinement to \mathcal{S}^1 , we obtain a new partition $\mathcal{S}^2 = (S_1^2, S_2^2, S_3^2)$, where $V^2 = \{1, 2, 3, 4, 5, 6, 7\}$, $R^1 = \text{SET}$, and:

$$\begin{aligned} S_1^2 &= \{1, 4, 6\} & \text{with } ADeg(\{1, 4, 6\}, \{1, 4, 6\}, G) &= (0, 0, 0) & \text{and } Valid(S_1^2) &= \text{FALSE} \\ S_2^2 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{1, 4, 6\}, G) &= (2, 0, 0) & \text{and } Valid(S_2^2) &= \text{TRUE} \\ S_3^2 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{1, 4, 6\}, G) &= (1, 0, 0) & \text{and } Valid(S_3^2) &= \text{TRUE} \end{aligned}$$

Again, since there are valid cells but no singleton ones, algorithm *IndexBestPivot* chooses S_2^2 ($P^2 = 2$) as a pivot set for a set refinement. This yields a new partition $\mathcal{S}^3 = (S_1^3, S_2^3, S_3^3, S_4^3)$, where $V^3 = \{1, 2, 3, 4, 5, 6, 7\}$, $R^2 = \text{SET}$, and:

$$\begin{aligned} S_1^3 &= \{6\} & \text{with } ADeg(\{6\}, \{5, 7\}, G) &= (2, 0, 0) & \text{and } Valid(S_1^3) &= \text{TRUE} \\ S_2^3 &= \{1, 4\} & \text{with } ADeg(\{1, 4\}, \{5, 7\}, G) &= (1, 0, 0) & \text{and } Valid(S_2^3) &= \text{TRUE} \\ S_3^3 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{5, 7\}, G) &= (0, 0, 0) & \text{and } Valid(S_3^3) &= \text{FALSE} \\ S_4^3 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{5, 7\}, G) &= (1, 0, 0) & \text{and } Valid(S_4^3) &= \text{TRUE} \end{aligned}$$

Having a valid singleton cell S_1^3 , it will be selected as the pivot set ($P^3 = 1$) for a vertex refinement, what yields partition $\mathcal{S}^4 = (S_1^4, S_2^4, S_3^4)$, where $V^4 = \{1, 2, 3, 4, 5, 7\}$, $R^3 = \text{VERTEX}$, and:

$$\begin{aligned} S_1^4 &= \{1, 4\} & \text{with } ADeg(\{1, 4\}, \{6\}, G) &= (0, 0, 0) & \text{and } Valid(S_1^4) &= \text{TRUE} \\ S_2^4 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{6\}, G) &= (1, 0, 0) & \text{and } Valid(S_2^4) &= \text{FALSE} \\ S_3^4 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{6\}, G) &= (0, 0, 0) & \text{and } Valid(S_3^4) &= \text{TRUE} \end{aligned}$$

This refinement has not been able to split any cell, but has discarded one cell (and one vertex), what reduces the complexity of the problem. This leaves two valid cells to be tried for a set refinement. However, none of them succeeds in splitting any cell. Hence, they are marked non valid, and a vertex refinement is performed with S_1^4 as the pivot set ($P^4 = 1$), setting

R^4 to BACKTRACK, and obtaining a new partition $\mathcal{S}^5 = (S_1^5, S_2^5, S_3^5, S_4^5, S_5^5)$, where $V^5 = \{2, 3, 4, 5, 7\}$, and:

$$\begin{aligned} S_1^5 &= \{4\} & \text{with } ADeg(\{4\}, \{1\}, G) &= (0, 0, 0) & \text{and } Valid(S_1^5) &= \text{TRUE} \\ S_2^5 &= \{7\} & \text{with } ADeg(\{7\}, \{1\}, G) &= (1, 0, 0) & \text{and } Valid(S_2^5) &= \text{TRUE} \\ S_3^5 &= \{5\} & \text{with } ADeg(\{5\}, \{1\}, G) &= (0, 0, 0) & \text{and } Valid(S_3^5) &= \text{TRUE} \\ S_4^5 &= \{2\} & \text{with } ADeg(\{2\}, \{1\}, G) &= (1, 0, 0) & \text{and } Valid(S_4^5) &= \text{TRUE} \\ S_5^5 &= \{3\} & \text{with } ADeg(\{3\}, \{1\}, G) &= (0, 0, 0) & \text{and } Valid(S_5^5) &= \text{TRUE} \end{aligned}$$

This partition has only singleton cells. Therefore, the algorithm stops, inducing the following order on the vertices of the graph: 0, 6, 1, 4, 7, 5, 2, 3. This order can be used to define a mapping between the vertices of the graphs, provided that an isomorphism is found.

5.2.2 Generation of the Sequence of Partitions for Graph H

The procedure is analogous to that for graph G . However, the sequence of partitions obtained for graph H may be completely different from the one generated for graph G . Algorithm *GenerateSequenceOfPartitions* starts setting $\mathcal{S}^0 = \mathcal{D}_H = (\{0, 1, 2, 3, 4, 5, 6, 7\})$, and marks its only cell as not valid. Then, since there is not a valid cell, it sets $R^0 = \text{BACKTRACK}$ and $P^0 = 1$, and performs a vertex refinement using vertex 0 as the pivot vertex, obtaining a partition $\mathcal{S}^1 = (S_1^1, S_2^1)$, where $V^1 = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} S_1^1 &= \{1, 2, 7\} & \text{with } ADeg(\{1, 2, 7\}, \{0\}, H) &= (1, 0, 0) & \text{and } Valid(S_1^1) &= \text{TRUE} \\ S_2^1 &= \{3, 4, 5, 6\} & \text{with } ADeg(\{3, 4, 5, 6\}, \{0\}, H) &= (0, 0, 0) & \text{and } Valid(S_2^1) &= \text{TRUE} \end{aligned}$$

Since both cells are valid, they must be tried for a set refinement. Algorithm *IndexBestPivot* sets $P^1 = 1$ (S_1^1 is the smallest cell), and a set refinement is performed ($R^1 = \text{SET}$), obtaining a partition $\mathcal{S}^2 = (S_1^2, S_2^2, S_3^2)$, where $V^2 = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} S_1^2 &= \{1, 2\} & \text{with } ADeg(\{1, 2\}, \{1, 2, 7\}, H) &= (1, 0, 0) & \text{and } Valid(S_1^2) &= \text{TRUE} \\ S_2^2 &= \{7\} & \text{with } ADeg(\{7\}, \{1, 2, 7\}, H) &= (0, 0, 0) & \text{and } Valid(S_2^2) &= \text{TRUE} \\ S_3^2 &= \{3, 4, 5, 6\} & \text{with } ADeg(\{3, 4, 5, 6\}, \{1, 2, 7\}, H) &= (1, 0, 0) & \text{and } Valid(S_3^2) &= \text{TRUE} \end{aligned}$$

Having a valid singleton cell S_2^2 , it is chosen as the pivot set for a vertex refinement. Hence, $P^2 = 2$, and $R^2 = \text{VERTEX}$. A new partition $\mathcal{S}^3 = (S_1^3, S_2^3, S_3^3)$ is generated, where $V^3 = \{1, 2, 3, 4, 5, 6\}$, and:

$$\begin{aligned} S_1^3 &= \{1, 2\} & \text{with } ADeg(\{1, 2\}, \{7\}, H) &= (0, 0, 0) & \text{and } Valid(S_1^3) &= \text{TRUE} \\ S_2^3 &= \{5, 6\} & \text{with } ADeg(\{5, 6\}, \{7\}, H) &= (1, 0, 0) & \text{and } Valid(S_2^3) &= \text{TRUE} \\ S_3^3 &= \{3, 4\} & \text{with } ADeg(\{3, 4\}, \{7\}, H) &= (0, 0, 0) & \text{and } Valid(S_3^3) &= \text{TRUE} \end{aligned}$$

Now, since there are valid cells, they must be tried for a set refinement. However, none of them succeeds in splitting at least one cell. Hence, they are all marked not valid, and a cell ($P^3 = 1$) is chosen as the pivot set for a vertex refinement, with pivot vertex 1, and setting $R^3 = \text{BACKTRACK}$. This yields a new partition $\mathcal{S}^4 = (S_1^4, S_2^4, S_3^4, S_4^4)$, where $V^4 = \{2, 3, 4, 5, 6\}$, and:

$$\begin{aligned}
S_1^4 = \{2\} & \quad \text{with } ADeg(\{2\}, \{1\}, H) = (1, 0, 0) & \quad \text{and } Valid(S_1^4) = \text{TRUE} \\
S_2^4 = \{5, 6\} & \quad \text{with } ADeg(\{5, 6\}, \{1\}, H) = (0, 0, 0) & \quad \text{and } Valid(S_2^4) = \text{FALSE} \\
S_3^4 = \{4\} & \quad \text{with } ADeg(\{4\}, \{1\}, H) = (1, 0, 0) & \quad \text{and } Valid(S_3^4) = \text{TRUE} \\
S_4^4 = \{3\} & \quad \text{with } ADeg(\{3\}, \{1\}, H) = (0, 0, 0) & \quad \text{and } Valid(S_4^4) = \text{TRUE}
\end{aligned}$$

Since there are singleton cells, algorithm *IndexBestPivot* selects the one, among them, with more remaining links. Vertex 2 is adjacent to vertex 3 (one remaining link) and vertex 4 is adjacent to vertices 3 and 5 (two remaining links), while vertex 3 is adjacent to vertices 2, 4, and 6. Hence, P^4 and R^4 are set to 4 and VERTEX respectively, and a vertex refinement is performed, obtaining a new partition $\mathcal{S}^5 = (S_1^5, S_2^5, S_3^5, S_4^5)$, where $V^5 = \{2, 4, 5, 6\}$, and:

$$\begin{aligned}
S_1^5 = \{2\} & \quad \text{with } ADeg(\{2\}, \{3\}, H) = (1, 0, 0) & \quad \text{and } Valid(S_1^5) = \text{FALSE} \\
S_2^5 = \{6\} & \quad \text{with } ADeg(\{6\}, \{3\}, H) = (1, 0, 0) & \quad \text{and } Valid(S_2^5) = \text{TRUE} \\
S_3^5 = \{5\} & \quad \text{with } ADeg(\{5\}, \{3\}, H) = (0, 0, 0) & \quad \text{and } Valid(S_3^5) = \text{TRUE} \\
S_4^5 = \{4\} & \quad \text{with } ADeg(\{4\}, \{3\}, H) = (1, 0, 0) & \quad \text{and } Valid(S_4^5) = \text{TRUE}
\end{aligned}$$

Thus we get a partition with only singleton cells, and hence, the algorithm stops. This sequence of partitions induces the following order on the vertices of graph H : 0, 7, 1, 3, 2, 6, 5, 4. It could be used subsequently to define a mapping between the vertices of graphs G and H .

5.2.3 Finding a Sequence of Partitions Compatible with the Target

Once we have the sequences of partitions for both graphs, we choose the one with less backtracking points as the target. Since both sequences of partitions have two points of backtracking, any of them would do. In this example, the algorithm chooses \mathcal{Q}_G as the target, and uses algorithm *Match* to try to find a sequence of partitions for graph H that is compatible with the target.

The algorithm starts from the degree partition $\mathcal{T} = \mathcal{D}_H = (\{0, 1, 2, 3, 4, 5, 6, 7\})$, that is already known to be compatible with \mathcal{D}_G . Since $R^0 = \text{BACKTRACK}$, the vertices of the only cell in partition $(\{0, 1, 2, 3, 4, 5, 6, 7\})$ will be tried in a loop until one succeeds, or all of them fail, in which case the graphs are not isomorphic. For simplicity, we try the vertices in lexicographic order. Then, with vertex 0 as the pivot vertex, a vertex refinement is performed, obtaining a partition $\mathcal{T}' = (T'_1, T'_2)$, where $W' = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 = \{1, 2, 7\} & \quad \text{with } ADeg(\{1, 2, 7\}, \{0\}, H) = (1, 0, 0) \\
T'_2 = \{3, 4, 5, 6\} & \quad \text{with } ADeg(\{3, 4, 5, 6\}, \{0\}, H) = (0, 0, 0)
\end{aligned}$$

Since this partition is compatible with the target partition \mathcal{S}^1 , a recursive call is made to algorithm *Match* to generate the next partition in the sequence. At this new stage, $R^1 = \text{SET}$. Hence, a set refinement is performed using cell $\{1, 2, 7\}$ as the pivot set (remember that $P^1 = 1$). This yields a partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where $W' = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 = \{1, 2\} & \quad \text{with } ADeg(\{1, 2\}, \{1, 2, 7\}, H) = (1, 0, 0) \\
T'_2 = \{7\} & \quad \text{with } ADeg(\{7\}, \{1, 2, 7\}, H) = (0, 0, 0) \\
T'_3 = \{3, 4, 5, 6\} & \quad \text{with } ADeg(\{3, 4, 5, 6\}, \{1, 2, 7\}, H) = (1, 0, 0)
\end{aligned}$$

This partition is not compatible with the target. In the target, the second cell was split, while in this one, it is the first one which has been split. Hence, the algorithm backtracks (returns FALSE) to a step l , such that $R^l = \text{BACKTRACK}$. In this example, this happens for $l = 0$. In

line 15 in algorithm *Match*, variable *success* gets value FALSE, and the loop is repeated since there are still other vertices to be tried. Next, vertex 1 is chosen as the pivot vertex and a vertex refinement is performed, yielding a new partition $\mathcal{T}' = (T'_1, T'_2)$, where $W' = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 2, 4\} & \text{with } ADeg(\{0, 2, 4\}, \{1\}, H) &= (1, 0, 0) \\ T'_2 &= \{3, 5, 6, 7\} & \text{with } ADeg(\{3, 5, 6, 7\}, \{1\}, H) &= (0, 0, 0) \end{aligned}$$

This partition is compatible with the target, so a recursive call is made to generate the next partition in the sequence. Now, recall that $R^1 = \text{SET}$ and $P^1 = 1$. Hence, a set refinement is performed using cell $\{0, 2, 4\}$ as the pivot set for a set refinement. Thus, we obtain a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where $W' = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 2\} & \text{with } ADeg(\{0, 2\}, \{0, 2, 4\}, H) &= (1, 0, 0) \\ T'_2 &= \{4\} & \text{with } ADeg(\{4\}, \{0, 2, 4\}, H) &= (0, 0, 0) \\ T'_3 &= \{3\} & \text{with } ADeg(\{3\}, \{0, 2, 4\}, H) &= (2, 0, 0) \\ T'_4 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{0, 2, 4\}, H) &= (1, 0, 0) \\ T'_5 &= \{6\} & \text{with } ADeg(\{6\}, \{0, 2, 4\}, H) &= (0, 0, 0) \end{aligned}$$

Again, we obtain a partition that is not compatible with the target. Hence, the algorithm backtracks to level $l = 0$, to choose a different pivot vertex. This time it takes vertex 2 and performs a vertex refinement. This yields a partition $\mathcal{T}' = (T'_1, T'_2)$, where $W' = \{0, 1, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 1, 3\} & \text{with } ADeg(\{0, 1, 3\}, \{2\}, H) &= (1, 0, 0) \\ T'_2 &= \{4, 5, 6, 7\} & \text{with } ADeg(\{4, 5, 6, 7\}, \{2\}, H) &= (0, 0, 0) \end{aligned}$$

This partition is again compatible with the target (since the graph is regular, every vertex in this graph has three adjacent vertices), so a recursive call is made to perform the next refinement. Since $R^1 = \text{SET}$ and $P^1 = 1$, a set refinement is performed with pivot cell $\{0, 1, 3\}$, obtaining a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where $W' = \{0, 1, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 1\} & \text{with } ADeg(\{0, 1\}, \{0, 1, 3\}, H) &= (1, 0, 0) \\ T'_2 &= \{3\} & \text{with } ADeg(\{3\}, \{0, 1, 3\}, H) &= (0, 0, 0) \\ T'_3 &= \{4\} & \text{with } ADeg(\{4\}, \{0, 1, 3\}, H) &= (2, 0, 0) \\ T'_4 &= \{6, 7\} & \text{with } ADeg(\{6, 7\}, \{0, 1, 3\}, H) &= (1, 0, 0) \\ T'_5 &= \{5\} & \text{with } ADeg(\{5\}, \{0, 1, 3\}, H) &= (0, 0, 0) \end{aligned}$$

This is an incompatible partition, and backtracking is again necessary. Note that this partition looks much like the one just discarded. It looks like with automorphism detection, this path could have been avoided in advance. Now, another vertex must be tried for a vertex refinement at backtracking point $l = 0$. Proceeding in lexicographic order, vertex 3 is chosen. After the vertex refinement, a new partition $\mathcal{T}' = (T'_1, T'_2)$ is obtained, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{2, 4, 6\} & \text{with } ADeg(\{2, 4, 6\}, \{3\}, H) &= (1, 0, 0) \\ T'_2 &= \{0, 1, 5, 7\} & \text{with } ADeg(\{0, 1, 5, 7\}, \{3\}, H) &= (0, 0, 0) \end{aligned}$$

Since this partition is compatible with the target, a recursive call is made to process the next partition in the sequence. Now, a set refinement will be performed with pivot set $\{2, 4, 6\}$. Thus we get a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 &= \{2, 4, 6\} && \text{with } ADeg(\{2, 4, 6\}, \{2, 4, 6\}, H) = (0, 0, 0) \\
T'_2 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{2, 4, 6\}, H) = (2, 0, 0) \\
T'_3 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{2, 4, 6\}, H) = (1, 0, 0)
\end{aligned}$$

This partition is finally compatible with the target and a recursive call is made to proceed to the next level. Here, since $P^2 = 2$ and $R^2 = \text{SET}$, pivot cell $\{1, 5\}$ is used for a set refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4)$, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 &= \{4\} && \text{with } ADeg(\{4\}, \{1, 5\}, H) = (2, 0, 0) \\
T'_2 &= \{2, 6\} && \text{with } ADeg(\{2, 6\}, \{1, 5\}, H) = (1, 0, 0) \\
T'_3 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{1, 5\}, H) = (0, 0, 0) \\
T'_4 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{1, 5\}, H) = (1, 0, 0)
\end{aligned}$$

Again, we get a compatible partition, and recursively proceed to the next level. Here, $P^3 = 1$ and $R^3 = \text{VERTEX}$. Hence, using vertex 4, a vertex refinement is performed, and a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$ is obtained, where $W' = \{0, 1, 2, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 &= \{2, 6\} && \text{with } ADeg(\{2, 6\}, \{4\}, H) = (0, 0, 0) \\
T'_2 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{4\}, H) = (1, 0, 0) \\
T'_3 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{4\}, H) = (0, 0, 0)
\end{aligned}$$

This partition is also compatible with the target. Therefore, a recursive call is made to process the next level. At this level, $P^4 = 1$ and $R^4 = \text{BACKTRACK}$. Hence, in lexicographic order, vertex 2 will be chosen for a vertex refinement. This yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where $W' = \{0, 1, 5, 6, 7\}$, and:

$$\begin{aligned}
T'_1 &= \{6\} && \text{with } ADeg(\{6\}, \{2\}, H) = (0, 0, 0) \\
T'_2 &= \{1\} && \text{with } ADeg(\{1\}, \{2\}, H) = (1, 0, 0) \\
T'_3 &= \{5\} && \text{with } ADeg(\{5\}, \{2\}, H) = (0, 0, 0) \\
T'_4 &= \{0\} && \text{with } ADeg(\{0\}, \{2\}, H) = (1, 0, 0) \\
T'_5 &= \{7\} && \text{with } ADeg(\{7\}, \{2\}, H) = (0, 0, 0)
\end{aligned}$$

Thus we get to the final partition, which is compatible with the target. Since $t = 5$, the final test in line 5 of algorithm *Match* is performed. Note that although the test has been defined in terms of available degree (for uniformity with the rest) it is equivalent to test the adjacencies among the vertices in the corresponding cells. Figure 5.2 shows these adjacencies, where it is easy to see the correspondence between the vertices of this final partition with the target. This completes the search for the isomorphism and algorithm *Match* returns TRUE.

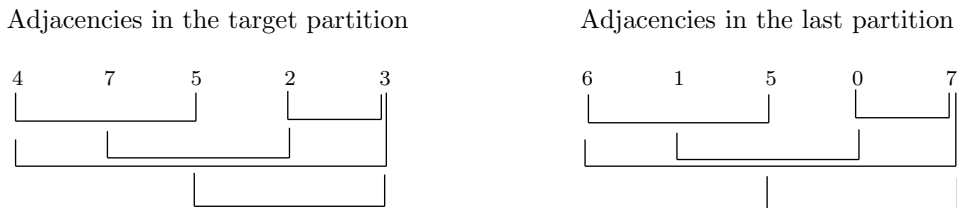


Figure 5.2: Equivalence of the final partitions.

Although the algorithm does not generate a correspondence between the vertices in graph G and the vertices in graph H , it is easy to derive it from the orderings induced by the sequences

of partitions generated. This correspondence is the following:

Graph G	0	6	1	4	7	5	2	3
Graph H	3	4	2	6	1	5	0	7

It is easy to imagine that testing a graph with itself for isomorphism with this algorithm is straightforward. However, depending on the way the vertices of the second graph are numbered (with respect to the first) finding the isomorphism between them can be much harder, and much more backtracking may be needed. Randomizing the order in which vertices are tried at the points where backtracking is needed may lead to a more uniform behavior.

5.3 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 5.1 *Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two isomorphic graphs. Then, there are two compatible sequences of partitions $\mathbf{Q}_G = (\mathbf{S}_G, \mathbf{R}_G, \mathbf{P}_G)$, and $\mathbf{Q}_H = (\mathbf{S}_H, \mathbf{R}_H, \mathbf{P}_H)$ for graphs G and H , respectively.*

Proof: Let $\mathbf{Q}_G = (\mathbf{S}_G, \mathbf{R}_G, \mathbf{P}_G)$ be any sequence of partitions for graph G . Let $\mathbf{S}_G = (\mathcal{S}^0, \dots, \mathcal{S}^t)$, $\mathcal{S}^0 = \text{DegreePartition}(G)$, $\mathbf{R}_G = (R_G^0, \dots, R_G^{t-1})$, and $\mathbf{P}_G = (P_G^0, \dots, P_G^{t-1})$, and for all $i \in \{0, \dots, t\}$, let $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$, 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 $\mathbf{Q}_H = (\mathbf{S}_H, \mathbf{R}_H, \mathbf{P}_H)$ for graph H which is compatible with \mathbf{Q}_G . Let $\mathbf{S}_H = (\mathcal{T}^0, \dots, \mathcal{T}^t)$, $\mathbf{R}_H = (R_H^0, \dots, R_H^{t-1})$ (the same type of refinement is performed at each step), and $\mathbf{P}_H = (P_H^0, \dots, P_H^{t-1})$ (corresponding pivot sets are used at each refinement). Moreover, for all $i \in \{0, \dots, t\}$, m maps the vertices in corresponding cells of \mathcal{S}^i and \mathcal{T}^i .

Let $\mathcal{T}^0 = \text{DegreePartition}(H)$. 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 $\mathcal{S}^0 = (S_1^0, \dots, S_r^0)$, and $\mathcal{T}^0 = (T_1^0, \dots, T_r^0)$. Then, m maps the vertices in S_i to the vertices in T_i , for all $i \in \{1, \dots, r\}$.

Now, by induction, we assume that compatibility exists up to partitions \mathcal{S}^l and \mathcal{T}^l , i.e. for all $i \in \{0, \dots, l\}$ 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. $R^l = \text{VERTEX}$.
2. $R^l = \text{SET}$.

3. $R^l = \text{BACKTRACK}$.

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 $T_{p^l}^l$, which contains a single vertex, image under m of the only vertex in $S_{p^l}^l$ (from the induction hypothesis). Let $\mathcal{S}^l = (S_1^l, \dots, S_r^l)$, and $\mathcal{T}^l = (T_1^l, \dots, T_r^l)$. 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 $i \in \{1, \dots, r\}$. Hence, if the pivot vertex in $S_{p^l}^l$ has a certain kind of link with k vertices in some cell S_i^l , then the vertex in $T_{p^l}^l$ 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 $T_{p^l}^l$. By the induction hypothesis, cells $S_{p^l}^l$ and $T_{p^l}^l$ 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 $S_{p^l}^l$ for the vertex refinement applied to partition \mathcal{S}^l . This vertex could be mapped by m to any vertex in $T_{p^l}^l$. 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 $S_{p^l}^l$ can only be mapped to vertices in $T_{p^l}^l$. 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. \blacksquare

Lemma 5.2 *Let $G = (V_G, R_G)$ and $H = (V_H, R_H)$ be two graphs, $|V_G| = |V_H| = n$. Let $\mathcal{Q}_G = (\mathcal{S}_G, \mathcal{R}_G, \mathcal{P}_G)$, and $\mathcal{Q}_H = (\mathcal{S}_H, \mathcal{R}_H, \mathcal{P}_H)$ be two compatible sequences of partitions for graphs G and H respectively. Let $\leq_{\mathcal{Q}_G}$ be the order induced by \mathcal{Q}_G on the vertices of V_G , and let $\leq_{\mathcal{Q}_H}$ be the order induced by \mathcal{Q}_H on the vertices of V_H . Then, graphs G and H are isomorphic, and mapping m defined as $m(\omega_{\mathcal{Q}_G}(i)) = \omega_{\mathcal{Q}_H}(i)$ for all $i \in \{1, \dots, |V_G|\}$ is an isomorphism of G and H .*

Proof: Let $\text{Adj}(G) = A$, and $\text{Adj}(H) = B$. Let $\mathcal{S}_G = (\mathcal{S}^0, \dots, \mathcal{S}^t)$ and $\mathcal{T}_G = (\mathcal{T}^0, \dots, \mathcal{T}^t)$. Since \mathcal{Q}_G and \mathcal{Q}_H are compatible sequences of partitions, their final partitions must also be compatible. Let $\mathcal{S}^t = (S_1^t, \dots, S_r^t)$ and $\mathcal{T}^t = (T_1^t, \dots, T_r^t)$ be the final partitions, and let $|V_G^t| = |V_H^t| = s$. Then, from Definition 4.14, we know that for all $x, y \in \{1, \dots, r\}$, $\text{ADeg}(S_x^t, S_y^t, G) = \text{ADeg}(T_x^t, T_y^t, H)$. Since all non-singleton cells in the final partitions have no remaining links, this means that for all $i, j \in \{1, \dots, s\}$, $A_{\omega_{\mathcal{Q}_G}(n-s+i), \omega_{\mathcal{Q}_G}(n-s+j)} = B_{\omega_{\mathcal{Q}_H}(n-s+i), \omega_{\mathcal{Q}_H}(n-s+j)}$. 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 $W_G^{l-1} = \{v \in V_G^{l-1} : \neg HasLinks(\{v\}, V_G^{l-1}, G)\}$ the set of vertices with no remaining links in V_G^{l-1} , and $W_H^{l-1} = \{v \in V_H^{l-1} : \neg HasLinks(\{v\}, V_H^{l-1}, H)\}$ 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 \leq_{Q_G} , and the vertices in W_H^{l-1} precede all the vertices in V_H^l in order \leq_{Q_H} . Hence, m maps the 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 $x \in \{1, \dots, r\}$, $ADeg(\{v\}, \mathcal{S}_x^l, G) = ADeg(\{w\}, \mathcal{T}_x^l, H)$. 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 \leq_{Q_G} , and w precedes all the vertices in V_H^l in order \leq_{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 5.1 *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 5.1 and 5.2. ■

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 5.2 *Two graphs G and H are isomorphic if and only if $AreIsomorphic(G, H)$ returns TRUE.*

Proof: If graphs G and H do not have the same number of vertices and arcs, or their degree partitions are not compatible, they can not be isomorphic, and algorithm *AreIsomorphic* returns FALSE. If their degree partitions are compatible, a sequence of partitions is generated for each graph. The one with less backtracking points is chosen as the target. Then algorithm *Match*

is used to search for a sequence of partitions, that is compatible with the target, for the other graph. This sequence of partitions starts from its degree partition, which is already known to be compatible with the target degree partition. In the search process, three cases may arise:

1. $R^l = \text{VERTEX}$.
2. $R^l = \text{SET}$.
3. $R^l = \text{BACKTRACK}$.

In Case 1, *Match* performs a vertex refinement with the corresponding pivot set, and tests the new partition for compatibility with the target. If they are compatible, it follows that branch in the search tree. If they are not compatible, it backtracks looking for an unexplored branch in the search tree. In this case, this branch can not yield a compatible sequence of partitions and there is no other alternative at this point, since the pivot cell has only one vertex and the target pivot vertex must be mapped to the vertex just tried.

In Case 2, *Match* applies a set refinement with the corresponding pivot set, testing the new partition for compatibility, and taking the same actions as in the previous case. Here again, there is no other possible choice.

In Case 3, any of the vertices in the pivot set may match the target pivot vertex, and only those that belong to the pivot set, since a vertex can only be mapped to vertices in its corresponding cell (if some of them does). Therefore, every vertex in the pivot set will be tried, generating every possible sequence of partitions from this point in the search (all possible branches are explored). If none of them matches the target sequence of partitions, backtracking is performed.

This simple backtracking algorithm clearly follows every feasible path in the search tree. Hence, if it is possible to generate a sequence of partitions compatible with the target, it will find it and return TRUE. If it can not find such a sequence of partitions, that is because such a sequence does not exist, and it returns FALSE. Hence, from Theorem 5.1, graphs G and H are isomorphic if and only if *AreIsomorphic*(G, H) returns TRUE. ■

5.4 Performance Evaluation

In this section we compare the performance of an implementation of our basic algorithm, which we call *sinauto*, with the two other programs of reference: *nauty* and *vf2*. The tests have been carried out on a Pentium III at 1.0 GHz with 256 MB of main memory, under Linux RedHat 9.0. All the programs have been compiled with the same compiler, GNU's gcc, and using the same optimization options. The execution time considered is the real time (not CPU time) consumed by the programs, excluding the loading time (the time needed by the programs to load from disk the graphs being tested). The CPU time limit for each program run was set to 10000 seconds. If a program was unable to finish within this CPU time limit for a pair of graphs of some size, no more tests for that or bigger size were performed for that program, and all previous results for that graph size and program were discarded.

The simplest graphs included in the tests are random graphs. They are very simple, unstructured, and are not likely to have automorphisms. Hence, simple vertex classification is, with very high probability, enough to verify isomorphism. We have only considered isomorphic graphs, since non-isomorphism of random graphs is very easy to discover. The results for directed and undirected graphs are shown in Figure 5.3. These graphs are easy for all algorithms considered,

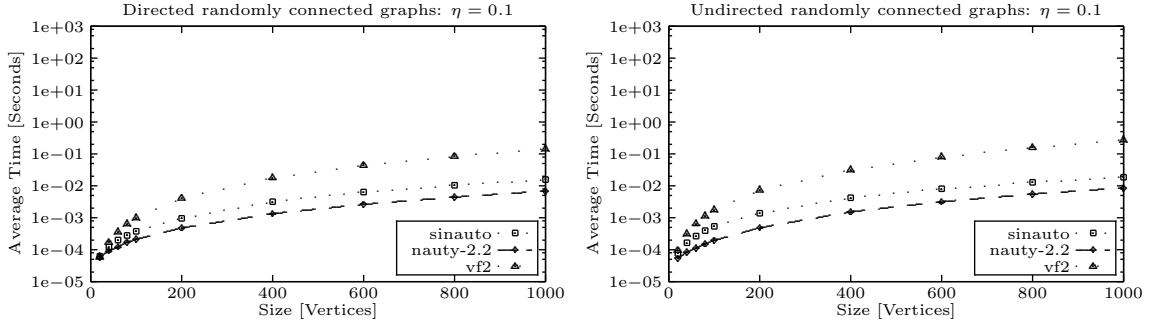


Figure 5.3: Performance of sinauto with isomorphic randomly connected graphs.

as expected. However, vf2 is one order of magnitude worse than the other two programs, nauty being the fastest. The explanation may be that nauty is faster than sinauto in obtaining the discrete partition of the vertices.

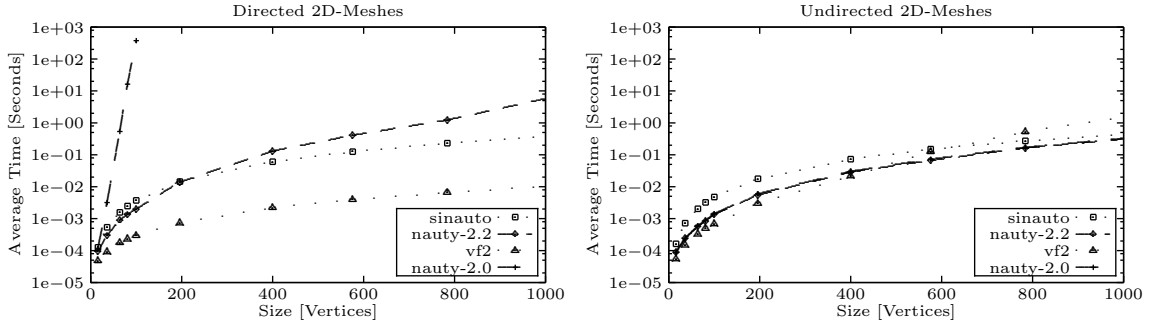


Figure 5.4: Performance of sinauto with isomorphic 2D-meshes.

While the behavior of the three programs is very similar for the directed and the undirected versions of the random graphs, this is not the case for 2D-meshes, as it can be seen in figure 5.4. First of all, the poor performance of version 2.0 of nauty with directed 2D-meshes is quite remarkable (it seems exponential) as it was already known [62]. However, version 2.2 fixes the problem and seems to perform polynomially for this family of graphs (which is a considerable improvement). Nevertheless, it is still, in the largest case considered, two orders of magnitude worse than vf2, and also worse than sinauto. What is surprising to us is that nauty works much better with the undirected versions of the graphs (both 2.0 and 2.2 versions), than with the directed versions, although in the latter, it has more information usable for vertex classification. Directed graphs are a well known weakness of nauty that is being overcome in some cases. On the contrary, vf2 is much faster (two orders of magnitude in the largest case considered) with the directed version than with the undirected version. This means that, while it is much better than the others for the directed version, it is the worst for the undirected one. With a uniform behavior combined with good overall performance, we have sinauto.

The performance for Miyazaki's graphs is shown in Figures 5.5 (isomorphic cases) and 5.6 (non-isomorphic cases). Miyazaki's graphs are known to be very hard graphs for nauty [54]. In fact, with the directed version, it was not able to handle graphs with only 40 vertices in 10000 seconds. That is why there is only one point in the corresponding plots for nauty. In the case of undirected graphs, nauty performs better, but although it starts quite well, its performance degrades considerably when graphs reach 400 vertices. Since nauty computes canonical forms of both graphs, it behaves uniformly for isomorphic and non-isomorphic graphs. This is not the

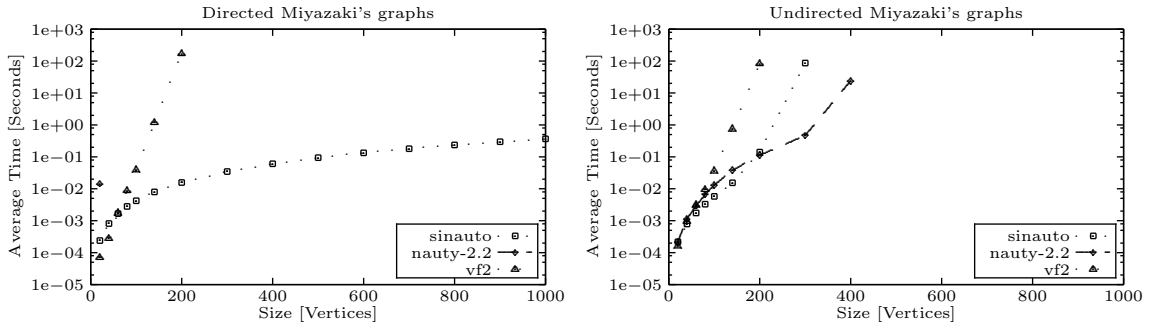


Figure 5.5: Performance of sinauto with isomorphic Miyazaki's Furer gadgets.

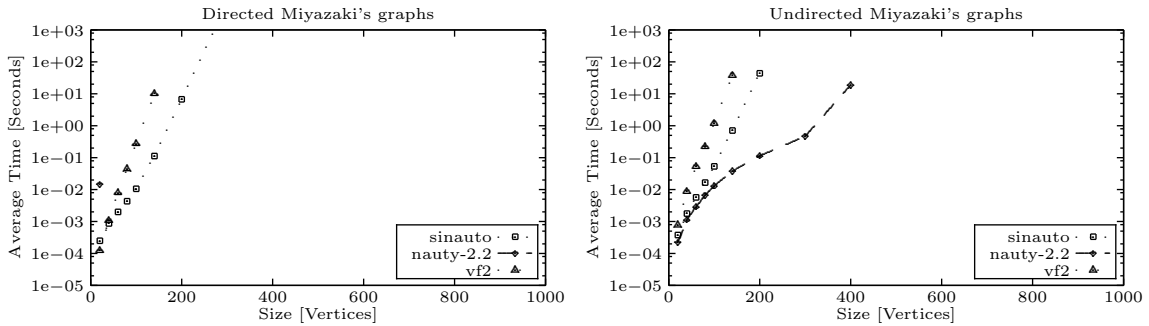


Figure 5.6: Performance of sinauto with non-isomorphic Miyazaki's Furer gadgets.

case for the other two algorithms.

These graphs also seem to make vf2 exponential in time. However, perhaps due to the way (order) it selects the pivot sets, the positive cases of directed graphs are relatively simple for sinauto. Yet, when the graphs are not isomorphic, sinauto seems also exponential in time. This difference in its behavior is due to its inability to detect automorphisms, what makes it explore automorphic unsuccessful paths in the search tree. Clearly, adding automorphism management to sinauto would certainly improve its performance. It is also clear that the undirected versions are much harder for sinauto than the directed ones.

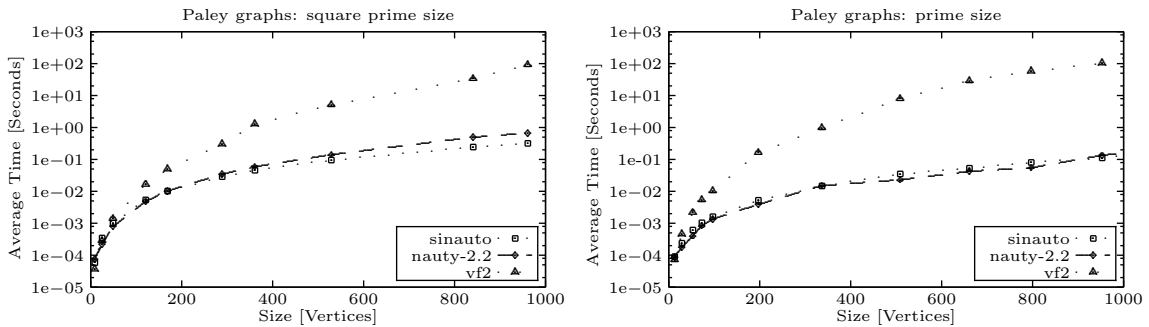


Figure 5.7: Performance of sinauto with Paley graphs.

The results obtained for Paley graphs are shown in Figure 5.7. It can be seen that vf2 is much slower than sinauto and nauty. For Paley graphs generated from finite fields of prime size it is three orders of magnitude slower than the others, for the largest case considered. For Paley graphs from finite fields of prime square size, the difference is a bit smaller since these graphs

are slightly harder for `sinauto` and `nauty`. Yet, all of them seem to be polynomial in time for this family of graphs.

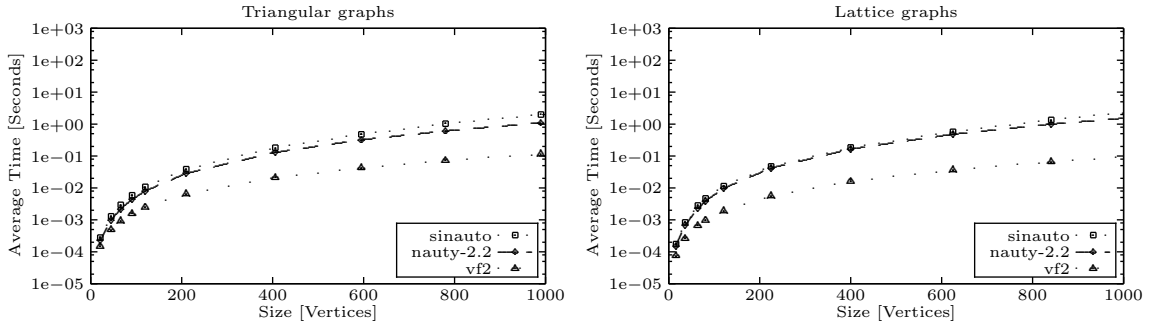


Figure 5.8: Performance of `sinauto` with triangular and lattice graphs.

Figure 5.8 shows the results for triangular graphs and lattice graphs. For these families of graphs, `vf2` is one order of magnitude faster than `sinauto` and `nauty`, which have a very similar behavior. It seems that the high degree of regularity makes it easy to find the automorphism in a direct way. Applying refinement techniques may be slower in this case, but should yield a more uniform behavior among different families of graphs.

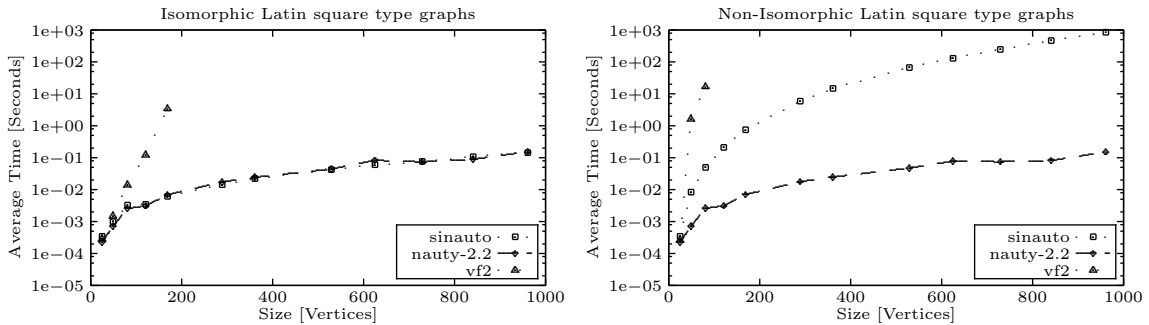


Figure 5.9: Performance of `sinauto` with Latin square graphs.

The next family we explore are Latin square graphs. These are much harder than other families of strongly regular graphs when automorphism detection is not used. Besides, it is possible to generate non-isomorphic pairs of Latin square graphs with the same parameters. This makes no difference for `nauty`, since it computes canonical forms of the graphs and it does not compare them directly. Consequently, it has exactly the same behavior for positive and negative tests, as it can be seen in Figure 5.9. That is not the case for the others. In particular, `vf2` seems to be exponential in time for both cases, and specifically with the negative tests, it is four orders of magnitude slower than with the positive ones. This is due to the high degree of similarity of the graphs being compared (`vf2` can not easily discard possible matchings between pairs of vertices).

The case of `sinauto` is quite different from the others. It seems to be polynomial in both cases, but it finds much harder to deal with non-isomorphic graphs (almost four orders of magnitude). This is due to the large number of unsuccessful automorphic paths in the search tree it follows. Adding automorphism detection must improve its performance considerably.

In Figures 5.10 and 5.11, the results for unions of tripartite graphs are shown. Again, like in the case of Latin square graphs, we have made both positive and negative tests. Besides, for

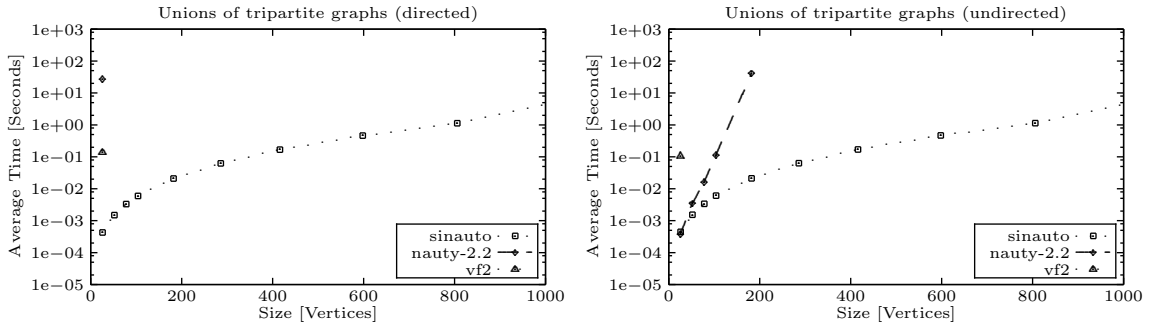


Figure 5.10: Performance of sinauto with isomorphic unions of tripartite graphs.

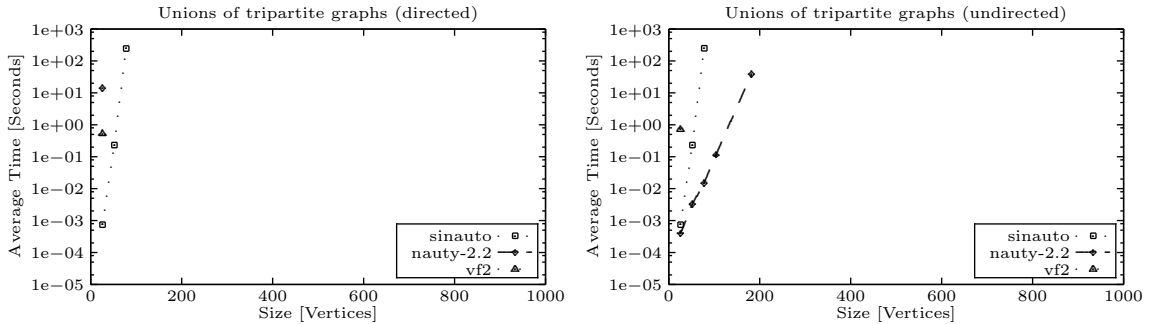


Figure 5.11: Performance of sinauto with non-isomorphic unions of tripartite graphs.

this family, directed and undirected versions of the graphs have also been generated. For the negative tests, we have used graphs that do not have the same number of isomorphic connected components. One graph has the same number of each connected component while the other has one less of one of the components and one more of the other. This difference explains the difference in the behavior of nauty between the positive and the negative tests, unlike other cases (in general, the behavior of nauty does not depend on the graphs tested being isomorphic or not).

With this family of graphs, vf2 is only able to finish within the 10000 seconds time limit imposed for the tests with the smallest graphs (those with only two components). This shows how hard it is to process these graphs for this algorithm.

Among positive tests (Figure 5.10), nauty is much faster for the undirected version of the graphs, than for the directed ones (for the directed version it can only finish within the time limit with graphs with only one component). Yet, it seems to be exponential in time in both cases. This is not the case of sinauto. For these positive tests, it looks polynomial.

The negative cases (Figure 5.11) are amongst the hardest cases for all the algorithms considered in the tests. Only sinauto can deal with graphs of more than two components. However, all of them seem to be exponential in time. Since sinauto looks polynomial for positive tests, we believe that adding automorphism detection to sinauto should make it polynomial, also for negative tests.

Figure 5.12 shows the results for unions of strongly regular graphs with the same parameters. These graphs are already known to make nauty exponential in time (cf. [54]). For vf2, they are so hard, that it can only finish within time with graphs with one component. Since sinauto looks polynomial for the positive cases, we expect it to be also polynomial for the negative ones,

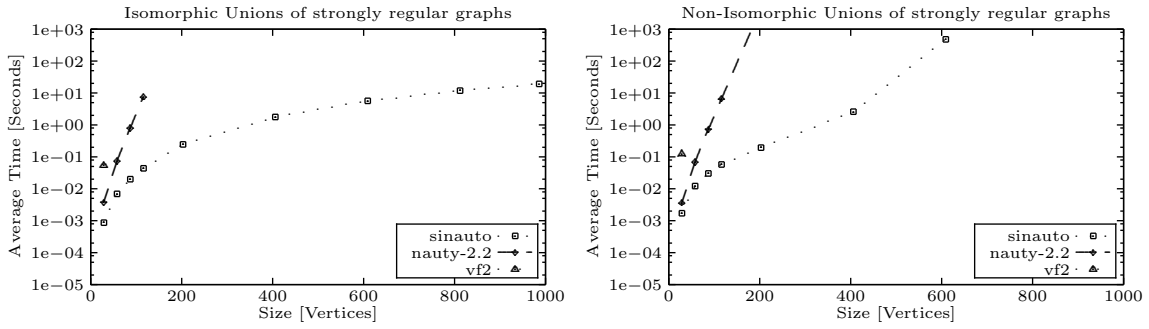


Figure 5.12: Performance of sinauto with unions of strongly regular graphs.

when adding automorphism detection to it.

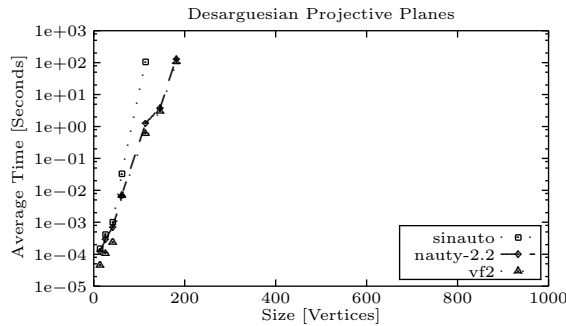


Figure 5.13: Performance of sinauto with point-line graphs of Desarguesian projective planes.

The results for the point-line graphs of Desarguesian projective planes are shown in Figure 5.13. They are known to be very hard for isomorphism, and, although we only consider positive cases, all the algorithms seem to be exponential for this family of graphs, being sinauto the slowest of the three. None of them is able to deal with graphs of more than 200 vertices. The number of paths in the search tree to be followed by these algorithms may be related with Miller's $O(n^{\log \log n + O(1)})$ bound for isomorphism of projective planes [52].

In the evaluation presented, only the average time has been shown. However, it may be very interesting to study other statistical parameters like the standard deviation, the maximum and the minimum time needed for each graph size, etc., since uniform behavior would be a desirable property of an isomorphism testing program. This will be done for the last algorithm proposed.

Chapter 6

Using Automorphisms

In this chapter, an improved version of the algorithm is presented, in which a partial search for automorphisms will be performed on the graphs, after their respective sequences of partitions have been generated. This automorphism discovery process will be used to eliminate some backtracking points, and to discard, during the search for a sequence of partitions compatible with the target, vertices that are known to be equivalent to other vertices already discarded, at some backtracking point. First, the theoretic background that supports this improvement will be exposed.

6.1 Theoretical Background

The notion of *equivalence* among vertices in a graph is essential for automorphism management. It will be used to eliminate backtracking points, to prune the search for automorphisms, and to prune the search for a sequence of partitions compatible with the target.

Definition 6.1 Let $G = (V, R)$ be a graph, and let $u, v \in V$. Vertices u and v are equivalent if there is an automorphism m of G that permutes u and v , i.e. $m(u) = v$. A vertex $w \in V$ is fixed by m if $m(w) = w$.

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, 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.

Our algorithm performs a partial computation of the orbit partition. The orbit partition will be computed incrementally, starting from the singleton partition. Since our algorithm performs a limited search for automorphisms, it is possible that it stops before the orbit partition is really found. Therefore, we will introduce the notion of *semiorbit*.

Definition 6.2 Let $G = (V, R)$ be a graph, and let $v \in V$. A semiorbit S of v is any subset of V such that for all $u \in S$, u and v are equivalent, i.e., a semiorbit of v is any subset of the orbit of v .

Definition 6.3 Let $G = (V, R)$ be a graph. A semiorbit partition of G is any partition $\mathcal{O} = \{O_1, \dots, O_n\}$ of V , such that for all $i \in \{1, \dots, n\}$, $v, u \in O_i$ implies that v and u are equivalent.

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 6.1 *Let $Q_G = (S, R, P)$ be a sequence of partitions for graph G , such that $S = (S^0, \dots, S^t)$, $R = (R^0, \dots, R^{t-1})$, and $P = (P^0, \dots, P^{t-1})$. Let $l \in \{1, \dots, t\}$, $S^l = (S_1^l, \dots, S_{r_l}^l)$, and $V^l = \bigcup_{i=1}^{r_l} S_i^l$, such that there is some $k \in \{1, \dots, r_l\}$ with $\text{NumLinks}(S_k^l, V^l, G) = 0$ and $|S_k^l| > 1$. Then, for all $u, v \in S_k^l$, u and v are equivalent.*

Proof: If u and v belong to the same cell S_k^l , 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 Lemma 5.2 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 $Q_G = (S, R, P)$ be a sequence of partitions for graph G . Let $S = (S^0, \dots, S^t)$, $R = (R^0, \dots, R^{t-1})$, $P = (P^0, \dots, P^{t-1})$. Let us assume that $R^l = \text{BACKTRACK}$ for some $l \in \{0, \dots, t-1\}$. Then, let $S^l = (S_1^l, \dots, S_{r_l}^l)$ and let $u \in S_{p_l}^l$ be the pivot vertex used for the vertex refinement at stage l . Let $v \in S_{p_l}^l, u \neq v$, 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_{Q_G} be the order induced by Q_G on the vertices of V , and let $\leq_{Q'_G}$ be the order induced by Q'_G on the same set of vertices V . Let $\omega_{Q_G}(i)$ denote the i^{th} vertex with respect to \leq_{Q_G} , and $\omega_{Q'_G}(i)$ denote the i^{th} vertex with respect to $\leq_{Q'_G}$. Then, mapping m , defined as $m(\omega_{Q_G}(i)) = \omega_{Q'_G}(i)$ for all $i \in \{1, \dots, |V|\}$, is an automorphism of G . Mapping m satisfies that $u = \omega_{Q_G}(k), v = \omega_{Q'_G}(k)$, for some $k \in \{1, \dots, |V|\}$. Then:

Lemma 6.2 *For all $j \in \{k, \dots, |V|\}$, $\omega_{Q_G}(j)$ and $\omega_{Q'_G}(j)$ are equivalent, and m fixes vertices $\omega_{Q_G}(1), \dots, \omega_{Q_G}(k-1)$, i.e. they are pairwise equal to $\omega_{Q'_G}(1), \dots, \omega_{Q'_G}(k-1)$.*

While the equivalence stated in Lemma 6.1 is somewhat universal, i.e., the automorphism discovered fixes the rest of the vertices in the graph, the equivalences stated in Lemma 6.2 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.

Definition 6.4 *Let $G = (V, R)$ be a graph. Two vertices $u, v \in V$ are equivalent at level l if there is an automorphism of G that permutes them, and fixes all the vertices in $V \setminus V^l$.*

Lemma 6.3 *If two vertices u and v are equivalent at level l , then they are equivalent at any level $i \in \{0, \dots, l-1\}$.*

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 6.2, 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 $i \in \{0, \dots, l-1\}$, $(V \setminus V^i) \subseteq (V \setminus V^l)$. Hence, there is an automorphism that maps u and v , and fixes the vertices in $V \setminus V^i$, e.g. m . ■

Remark 6.1 *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 6.5 *Let $G = (V, R)$ be a graph, and let $\mathcal{O} = \{O_1, \dots, O_n\}$ be a partition of V . Then, $\text{merge}(\mathcal{O}, O_i, O_j) = \mathcal{O} \setminus \{O_i, O_j\} \cup \{O_i \cup O_j\}$.*

Besides, we will use $\text{Orb}(v, \mathcal{O})$ to denote the semiorbit to which vertex v belongs in a semiorbit partition \mathcal{O} . Let us now extend the concept of sequence of partitions to include the semiorbit partition.

Definition 6.6 *Let $G = (V, R)$ be a graph. An extended sequence of partitions \mathbf{E} for graph G is a tuple $(\mathbf{Q}, \mathcal{O})$, where \mathbf{Q} is a sequence of partitions, denoted as $\text{SeqPart}(\mathbf{E})$, and \mathcal{O} is a semiorbit partition of G , denoted as $\text{Orbits}(\mathbf{E})$.*

We observe now that when all the vertices in a pivot set used at a backtracking point ($R^l = \text{BACKTRACK}$) are proved to be equivalent, R^l can be set to VERTEX, eliminating a backtracking point in the search for a sequence of partitions compatible with the target. This is a consequence of the fact that automorphisms are preserved under isomorphisms, which is stated in the following lemma:

Lemma 6.4 *If the vertices of a pivot set in a sequence of partitions \mathbf{Q}_G for graph G are equivalent, then in a compatible sequence of partitions \mathbf{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 struc-

ture. 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 6.1 *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 [40, 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.

In the following sections, for simplicity, we will use the terms *orbit* and *orbit partition* to refer to *semiorbits* and *semiorbit partitions*.

6.2 Algorithm *conauto-v0*

We present now an algorithm, called *conauto-v0*, which is based on the previous algorithm *sinauto*. This new algorithm makes use of discovered automorphisms to eliminate backtracking points, and to discard vertices in advance at the remaining backtracking points as described. Since the information used about automorphisms is rather limited, we do not expect it to have a huge impact as it does in *nauty*, where more sophisticated automorphism management is used.

6.2.1 Main Algorithm

Algorithm 5 describes the behavior of *conauto-v0*. Like the previous algorithm (*sinauto*), it receives two graphs G and H as parameters and returns TRUE if both graphs are isomorphic, and FALSE if they are not.

Algorithm 5 makes initial tests to discard trivial cases, just like Algorithm 1. Next it generates the sequences of partitions Q_G and Q_H for graphs G and H respectively, using Algorithm 2 (the same used in Algorithm 1). Then, using a new Algorithm 6, it looks for automorphisms, and obtains two extended sequences of partitions E_G and E_H for graphs G and H respectively, which include the discovered orbits for each graph. Finally, it takes the sequence of partitions with least remaining backtracking points as the target, and the orbits of the other graph, and

Algorithm 5 Test whether G and H are isomorphic (*conauto-v0*).

```

AreIsomorphic2( $G, H$ ) : 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    $\mathcal{D}_G \leftarrow \text{DegreePartition}(G)$ 
6    $\mathcal{D}_H \leftarrow \text{DegreePartition}(H)$ 
7   if  $\mathcal{D}_G$  and  $\mathcal{D}_H$  are not compatible under  $G$  and  $H$  respectively then
8     return FALSE
9   else
10     $\mathbf{Q}_G \leftarrow \text{GenerateSequenceOfPartitions}(G, \mathcal{D}_G)$ 
11     $\mathbf{Q}_H \leftarrow \text{GenerateSequenceOfPartitions}(H, \mathcal{D}_H)$ 
12     $\mathbf{E}_G \leftarrow \text{FindAutomorphisms}(G, \mathbf{Q}_G)$ 
13     $\mathbf{E}_H \leftarrow \text{FindAutomorphisms}(H, \mathbf{Q}_H)$ 
14    if  $\text{BacktrackAmount}(\text{SeqPart}(\mathbf{E}_G)) \leq \text{BacktrackAmount}(\text{SeqPart}(\mathbf{E}_H))$  then
15      return  $\text{Match2}(0, G, H, \text{SeqPart}(\mathbf{E}_G), \mathcal{D}_H, \text{Orbits}(\mathbf{E}_H))$ 
16    else
17      return  $\text{Match2}(0, H, G, \text{SeqPart}(\mathbf{E}_H), \mathcal{D}_G, \text{Orbits}(\mathbf{E}_G))$ 
18    end if
19  end if
20 end if

```

calls Algorithm 10, which is a new version of the *Match* algorithm of Algorithm 1, to look for a sequence of partitions compatible with the target. These orbits will help Algorithm 10 prune the search space, making it possible to discard some vertices at a backtracking point without the need of testing them.

6.2.2 Search for Automorphisms

The search for automorphisms is performed by Algorithm 6. It starts from the last partition in the sequence, and traverses the sequence up to the first. This way, Lemma 6.3 will be applicable, so the automorphisms already found may be used when processing previous partitions in the sequence. At each level, the algorithm tries to use Lemmas 6.1 and 6.2 to find equivalences among vertices.

Lemma 6.1 is applicable at each level l in the sequence of partitions provided that there are cells with no remaining links. This is done by Algorithm 7. However, Lemma 6.2 is only applicable when $R^l = \text{BACKTRACK}$. In this case, the algorithm proceeds as follows:

First, 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} .

Then, for each vertex in the pivot cell that belongs to an orbit different from that of the original pivot vertex, and which is valid, an alternative sequence of partitions is generated using Algorithm 9. If it is compatible with the original one, Lemma 6.2 is applied using Algorithm 8 to establish vertex equivalences. As a consequence of this, the orbit of the alternative pivot vertex is merged with that of the original one. Hence, the vertices that were equivalent to the alternative pivot vertex will belong to the orbit of the original one, and they will be ignored in subsequent walks through the loop. If the alternative sequence of partitions is not compatible

Algorithm 6 Look for automorphisms.

```
FindAutomorphisms( $G, Q$ ) : extended sequence of partitions
1 -- let  $G = (V, R)$ 
2 -- let  $Q = (S, R, P)$ , let  $S = (S^0, \dots, S^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $P = (P^0, \dots, P^{t-1})$ 
3 -- let  $\mathcal{S}^l = (S_1^l, \dots, S_{r_l}^l)$ ,  $V^l = \bigcup_{j=1}^{r_l} S_j^l$ , for all  $l \in \{0, \dots, t\}$ 
4  $O \leftarrow \{\{v_i\} : v_i \in V\}$ 
5  $O \leftarrow \text{ProcessCellsWithNoLinks}(G, Q, O)$ 
6  $l \leftarrow t - 1$ 
7 while  $l \geq 0$  do
8   if  $R^l = \text{BACKTRACK}$  then
9     -- let  $p$  be the pivot vertex used to generate partition  $\mathcal{S}^{l+1}$ 
10    for each  $v \in (S_{P^l}^l \setminus \{p\})$  do
11       $\text{Valid}(\text{Orb}(v, O)) \leftarrow \text{TRUE}$ 
12    end for
13     $\text{success} \leftarrow \text{TRUE}$ 
14    for each  $v \in (S_{P^l}^l \setminus \{p\})$  do
15      if  $\text{Orb}(p, O) \neq \text{Orb}(v, O) \wedge \text{Valid}(\text{Orb}(v, O))$  then
16         $Q' \leftarrow \text{GenerateAlternativeSequenceOfPartitions}(l, v, \mathcal{S}^l, G, Q)$ 
17        if  $Q'$  is a sequence of partitions compatible with  $Q$  then
18           $O \leftarrow \text{ProcessCompatibleSequencesOfPartitions}(G, Q, Q', O)$ 
19        else
20           $\text{Valid}(\text{Orb}(v, O)) \leftarrow \text{FALSE}$ 
21           $\text{success} \leftarrow \text{FALSE}$ 
22        end if
23      end if
24    end for
25    if  $\text{success}$  then
26       $R^l \leftarrow \text{VERTEX}$ 
27    end if
28  end if
29   $l \leftarrow l - 1$ 
30 end while
31 return ( $Q, O$ )
```

with the original one, the orbit of the alternative pivot vertex is marked non-valid, and all the vertices in its orbit will be ignored in subsequent walks through the loop.

When, at a backtracking point, all the vertices in the pivot cell are found to be equivalent, R^l is changed from BACKTRACK to VERTEX. Recall that, from Lemma 6.4, 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 9. In this search, no backtracking is performed. This imposes some restrictions on the algorithm. The new sequence of partitions Q' is the old one up to level l , where a different pivot vertex (from the one used in the original sequence Q) is used for the refinement. To compute the rest of the sequence, the algorithm first generates a new partition \mathcal{T}^{l+1} as an alternative to \mathcal{S}^{l+1} , using the pivot vertex v received as a parameter. If these partitions are compatible, then the process can go on. Otherwise, it is not possible to generate a compatible sequence of partitions for that vertex.

If partition \mathcal{T}^{l+1} is compatible with \mathcal{S}^{l+1} , then new partitions in the sequence Q' are generated using the same kind of refinement R^k with the corresponding pivot cell $T_{P^k}^k$. If at some step, a

Algorithm 7 Apply Lemma 6.1.

ProcessCellsWithNoLinks(G, Q, O) : orbit partition

- 1 -- let $G = (V, R)$
- 2 -- let $Q = (S, R, P)$, let $S = (S^0, \dots, S^t)$, $R = (R^0, \dots, R^{t-1})$, $P = (P^0, \dots, P^{t-1})$
- 3 -- let $S^l = (S^l_1, \dots, S^l_{r_l})$, $V^l = \bigcup_{j=1}^{r_l} S^l_j$, for all $l \in \{0, \dots, t\}$
- 4 **for each** $l \in \{0, \dots, t\}$ **do**
- 5 **for each** $S^l_x \in S^l : \neg HasLinks(S^l_x, V^l, G)$ **do**
- 6 **for each** $u, v \in S^l_x, u \neq v$ **do**
- 7 $O \leftarrow merge(O, Orb(u, O), Orb(v, O))$
- 8 **end for**
- 9 **end for**
- 10 **end for**
- 11 **return** O

Algorithm 8 Apply Lemma 6.2.

ProcessCompatibleSequencesOfPartitions(G, Q, Q', O) : orbit partition

- 1 -- let $G = (V, R)$
- 2 -- let \leq_Q be the order induced by Q
- 3 -- let $\leq_{Q'}$ be the order induced by Q'
- 4 -- let $\omega_Q(i)$ denote the i^{th} vertex with respect to \leq_Q
- 5 -- let $\omega_{Q'}(i)$ denote the i^{th} vertex with respect to $\leq_{Q'}$
- 6 **for each** $i \in \{1, \dots, |V|\}$ **do**
- 7 $O \leftarrow merge(O, Orb(\omega_Q(i), O), Orb(\omega_{Q'}(i), O))$
- 8 **end for**
- 9 **return** O

partition \mathcal{T}^k is found that is not only compatible with S^k , but identical, then it is not necessary to go further, since the rest of both sequences must be the same. If an incompatibility is found, then it is also unnecessary to go on, since the sequences of partitions can not be compatible.

In case a vertex refinement ($R^k = VERTEX$) is performed with a pivot cell with more than one vertex ($|S^k_{P^k}| > 1$), that must be the case that, previously, all the vertices in that pivot cell of the original partition were found to be equivalent. Lemma 6.4 can also be applied in this case. Hence, a vertex refinement will be performed and the algorithm will proceed as in the case where $|S^k_{P^k}| = 1$.

If the algorithm finds $R^k = BACKTRACK$ 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. However, if there are more than one such levels, then backtracking would be needed. Since we do not want backtracking in this algorithm, if such a level is found, an incompatible sequence of partitions will be returned, and no automorphism will be found. This is not an optimal solution, but we hope it does not imply an important loss of performance. Nevertheless, since we are not trying to compute the automorphism group of the graphs, missing some automorphisms should not be too problematic.

6.2.3 Search for a Sequence of Partitions Compatible with the Target

Algorithm *Match2* is a recursive algorithm that receives a level l to process in the sequence of partitions, the graphs G and H to test, the sequence of partitions Q_G for graph G , the partition at level l for graph H , \mathcal{T} , and the orbit partition O_H previously obtained for graph H . It returns

Algorithm 9 Try to generate a compatible sequence of partitions without backtracking.

```

GenerateAlternativeSequenceOfPartitions( $l, v, \mathcal{S}^l, G, \mathcal{Q}$ ) : sequence of partitions
1 -- let  $G = (V, R)$ 
2 -- let  $\mathcal{Q} = (\mathcal{S}, R, \mathcal{P})$ , let  $\mathcal{S} = (\mathcal{S}^0, \dots, \mathcal{S}^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $\mathcal{P} = (P^0, \dots, P^{t-1})$ 
3 -- let  $\mathcal{Q}' = (\mathcal{T}, R, \mathcal{P})$ , let  $\mathcal{T} = (\mathcal{T}^0, \dots, \mathcal{T}^t)$ 
4 -- for all  $i \in \{0, \dots, t\}$ , let  $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$ ,  $V^i = \bigcup_{j=1}^{r_i} S_j^i$ 
5 -- for all  $i \in \{0, \dots, l\}$ , let  $\mathcal{T}^i = \mathcal{S}^i$ 
6 -- for all  $i \in \{l+1, \dots, t\}$ , if  $\mathcal{T}^i$  is defined, let  $\mathcal{T}^i = (T_1^i, \dots, T_{r_i}^i)$ ,  $W^i = \bigcup_{j=1}^{r_i} T_j^i$ 
7  $\mathcal{T}^{l+1} \leftarrow \text{VertexRefinement}(\mathcal{S}^l, v, G_{V^l})$ 
8  $k \leftarrow l+1$ 
9 while  $k < t$  and  $\mathcal{S}^k$  and  $\mathcal{T}^k$  are compatible under  $G_{V^k}$  and  $G_{W^k}$  respectively do
10   if  $\mathcal{T}^k = \mathcal{S}^k$  then
11     for each  $i \in \{k+1, \dots, t\}$  do
12        $\mathcal{T}^i \leftarrow \mathcal{S}^i$ 
13     end for
14      $k \leftarrow t$ 
15   else
16     if  $R^k = \text{BACKTRACK}$  then
17        $\mathcal{T}^k \leftarrow \emptyset$ 
18        $k \leftarrow t$ 
19     else
20       if  $R^k = \text{VERTEX}$  then
21          $v \leftarrow$  any vertex in  $T_{P^k}^k$ 
22          $\mathcal{T}^{k+1} \leftarrow \text{VertexRefinement}(\mathcal{T}^k, v, G_{W^k})$ 
23          $k \leftarrow k+1$ 
24       else (i.e.  $R^k = \text{SET}$ )
25          $\mathcal{T}^{k+1} \leftarrow \text{SetRefinement}(\mathcal{T}^k, T_{P^k}^k, G_{W^k})$ 
26          $k \leftarrow k+1$ 
27       end if
28     end if
29   end if
30 end while
31 return  $\mathcal{Q}'$ 

```

TRUE if it is able to find a sequence of partitions for graph H starting with partition \mathcal{T} that is compatible with \mathcal{Q}_G , and FALSE otherwise.

Algorithm 10 is based on Algorithm 4. It starts with a partition \mathcal{T} that is compatible with \mathcal{S}^l . Then, if the type of refinement used to generate the next partition in the sequence is VERTEX, it applies a vertex refinement to partition \mathcal{T} . Then, if the new partition generated \mathcal{T}' is compatible with \mathcal{S}^{l+1} , it recursively calls itself to process the next partition in the sequence. In the case of a set refinement, it works in a similar way, just like Algorithm 4 does.

If $R^l = \text{BACKTRACK}$, it makes use of the discovered vertex equivalences to prune the search space. It uses the attribute *Valid* of the orbits of the vertices in the pivot cell T_{P^l} to discard vertices that are equivalent to some other vertex that has already been discarded. Note that all the vertices in $V_H \setminus W$ have been fixed in the previous levels. If all the vertices in $V_H \setminus W$ belong to singleton orbits, from Lemma 6.2 the discovered orbits for graph H may be applied, but we can not be sure if they are valid when some vertices in non-singleton orbits have been previously fixed. More sophisticated isomorphism management may help here, but we have discarded for now that possibility in favor of simplicity. Hence, vertex equivalence will only be applied when all the previously fixed vertices belong to singleton orbits. It is easy to see that the orbits are

Algorithm 10 Find a sequence of partitions compatible with the target (*conauto-v0*).

```

Match2( $l, G, H, Q_G, T, O_H$ ) : boolean
1 -- let  $Q_G = (S, R, P)$ , let  $S = (S^0, \dots, S^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $P = (P^0, \dots, P^{t-1})$ 
2 -- let  $S^l = (S^l_1, \dots, S^l_{r_l})$ ,  $V^l = \bigcup_{j=1}^{r_l} S^l_j$ , for all  $l \in \{0, \dots, t\}$ 
3 -- let  $H = (V_H, R_H)$ , let  $T = (T_1, \dots, T_{r_1})$ ,  $W = \bigcup_{j=1}^{r_1} T_j$ 
4 if  $l = t$  then
5    $success \leftarrow \forall x, y \in \{1, \dots, r_l\}, ADeg(S^t_x, S^t_y, G) = ADeg(T_x, T_y, H)$ 
6 else
7    $X \leftarrow T_{P^l}$ 
8   if  $R^l = \text{BACKTRACK}$  then
9      $orbitsApplicable \leftarrow \forall v \in V_H \setminus W, |Orb(v, O_H)| = 1$ 
10    for each  $v \in X$  do
11       $Valid(Orb(v, O_H)) \leftarrow \text{TRUE}$ 
12    end for
13    repeat
14       $v \leftarrow$  any vertex in  $X$ 
15       $X \leftarrow X \setminus \{v\}$ 
16      if  $\neg orbitsApplicable \vee Valid(Orb(v, O_H))$  then
17         $T' \leftarrow VertexRefinement(T, v, H_W)$ 
18        -- let  $T' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
19        if  $S^{l+1}$  and  $T'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
20           $success \leftarrow Match2(l + 1, G, H, Q_G, T', O_H)$ 
21        else
22           $success \leftarrow \text{FALSE}$ 
23        end if
24         $Valid(Orb(v, O_H)) \leftarrow \text{FALSE}$ 
25      end if
26    until  $X = \emptyset \vee success$ 
27  else
28    if  $R^l = \text{VERTEX}$  then
29       $v \leftarrow$  any vertex in  $X$ 
30       $T' \leftarrow VertexRefinement(T, v, H_W)$ 
31    else (i.e.  $R^l = \text{SET}$ )
32       $T' \leftarrow SetRefinement(T, X, H_W)$ 
33    end if
34    -- let  $T' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
35    if  $S^{l+1}$  and  $T'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
36       $success \leftarrow Match2(l + 1, G, H, Q_G, T', O_H)$ 
37    else
38       $success \leftarrow \text{FALSE}$ 
39    end if
40  end if
41 end if
42 return  $success$ 

```

always applicable at the first level in the sequence of partitions, since at this first level no vertex has been fixed yet.

To start with, all the orbits of the vertices in the pivot cell are considered valid, since none of them has been discarded yet. Then, the vertices in the pivot cell are tried until one of them yields a sequence of partitions compatible with the target (just like it was done in Algorithm 4), or all of them have been tried unsuccessfully (or discarded). If orbits are applicable, then the vertices that belong to the orbit of a previously discarded vertex are directly discarded.

6.3 Example

To illustrate the behavior of algorithm `conauto-v0`, we will use the sample graphs in Figure 5.1. These sample graphs will show how the algorithm discovers automorphisms, and how the search for a sequence of partitions compatible with the target is pruned. We will not reproduce the generation of the sequences of partitions for graphs G and H , since that was already shown in Sections 5.2.1 and 5.2.2.

In the following sections we show the search for automorphisms for the sample graphs G and H which is performed by Algorithm 6, the choice of the target, and the search for a sequence of partitions compatible with the target, for the other graph, which is performed by Algorithm 10.

6.3.1 Search for Automorphisms in graph G

First, the trivial orbit partition $\mathbf{O} = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$ for graph G is computed. Then, the sequence of partitions obtained for graph G is traversed backwards starting with partition \mathcal{S}^5 . Since there are no non-singleton cells (with no remaining links), Algorithm 7 yields no orbit merges.

Since $R^4 = \text{BACKTRACK}$, the vertices in the pivot cell $S_1^4 = \{1, 4\}$ other than 1 (the pivot vertex used in the original sequence of partitions) are tested for equivalence. This leaves only vertex 4 to be tested. Its orbit is marked valid and, since vertices 1 and 4 belong to different orbits, vertex 4 is used to generate an alternative sequence of partitions. This is done with Algorithm 9. In this case, only one new partition needs to be generated. A vertex refinement of partition $\mathcal{S}^4 = (\{1, 4\}, \{5, 7\}, \{2, 3\})$ yields the partition $\mathcal{T}^5 = (T_1^5, T_2^5, T_3^5, T_4^5, T_5^5)$, where $W^5 = \{1, 2, 3, 5, 7\}$, and:

$$\begin{aligned} T_1^5 &= \{1\} && \text{with } ADeg(\{1\}, \{4\}, G) = (0, 0, 0) \\ T_2^5 &= \{5\} && \text{with } ADeg(\{5\}, \{4\}, G) = (1, 0, 0) \\ T_3^5 &= \{7\} && \text{with } ADeg(\{7\}, \{4\}, G) = (0, 0, 0) \\ T_4^5 &= \{3\} && \text{with } ADeg(\{3\}, \{4\}, G) = (1, 0, 0) \\ T_5^5 &= \{2\} && \text{with } ADeg(\{2\}, \{4\}, G) = (0, 0, 0) \end{aligned}$$

This partition is compatible with \mathcal{S}^5 . Since \mathcal{T}^5 and \mathcal{S}^5 are the final partitions in their respective sequences of partitions, it must be tested that mapping vertex 1 to 4, 5 to 7, 7 to 5, 3 to 2, and 2 to 3 is an isomorphism of G_{V^5} and G_{W^5} . This is best illustrated in Figure 6.1.

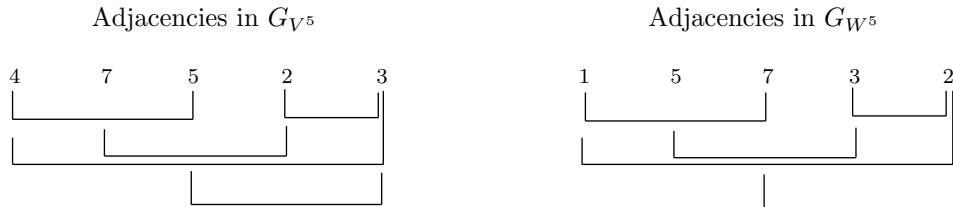


Figure 6.1: Adjacencies in G_{V^5} and G_{W^5} .

It is easy to see that the defined mapping is, in fact, an isomorphism of G_{V^5} and G_{W^5} . Furthermore, it leads to an automorphism of G . The alternative sequence of partitions generated by Algorithm 9 is compatible with the original one, and Algorithm 8 is applied to merge the

corresponding orbits. Thus, the orbits of vertices 1 and 4 are merged, and the same applies to the orbits of vertices 5 and 7, and to vertices 2 and 3. This yields a new orbit partition $\mathcal{O} = \{\{0\}, \{1, 4\}, \{2, 3\}, \{5, 7\}, \{6\}\}$.

Since vertices 1 and 4 are equivalent, then R^l is changed to VERTEX. Next, for $l = 4$, there are no cells with no remaining links, so Algorithm 7 yields no result either, and the algorithm proceeds to level 3. At this level, $R^l = \text{VERTEX}$ and there is no cell with no remaining links, so the algorithm proceeds to level 2. Here again, $R^l = \text{SET}$ and there are no cells with no remaining links, so l is decremented. The same case applies to level 1, so l is again decremented. Thus we get to level 0, where $R^0 = \text{BACKTRACK}$.

First, the orbits $\{1, 4\}$, $\{2, 3\}$, $\{5, 7\}$, and $\{6\}$ are marked valid. Then, alternative sequences of partitions are generated for the other vertices in the pivot cell (the only one in the partition). Let us assume they are tried in lexicographical order. Hence, the candidate vertices will be 1, 2, then vertex 3 will be discarded since it is equivalent to 2 (if 0 and 2 are equivalent, then 0 and 3 are also equivalent from Lemma 6.3, and similarly if 0 and 2 are not equivalent, then 0 and 3 can not be either), vertex 4 is also discarded for the same reason, 5, 6, and 7 will also be discarded since it is equivalent to 5. Since the graph is regular, the first vertex refinement with the alternative pivot vertex will always yield a partition that is compatible with the original one.

Vertex 1 yields a partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{0, 2, 7\} & \text{with } ADeg(\{0, 2, 7\}, \{1\}, G) &= (1, 0, 0) \\ T_2^1 &= \{3, 4, 5, 6\} & \text{with } ADeg(\{3, 4, 5, 6\}, \{1\}, G) &= (0, 0, 0) \end{aligned}$$

Refining this partition by set, using T_1^1 as the pivot cell, we obtain a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2, T_4^2, T_5^2)$, where $W^2 = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{2, 7\} & \text{with } ADeg(\{2, 7\}, \{0, 2, 7\}, G) &= (1, 0, 0) \\ T_2^2 &= \{0\} & \text{with } ADeg(\{0\}, \{0, 2, 7\}, G) &= (0, 0, 0) \\ T_3^2 &= \{6\} & \text{with } ADeg(\{6\}, \{0, 2, 7\}, G) &= (2, 0, 0) \\ T_4^2 &= \{3, 4\} & \text{with } ADeg(\{3, 4\}, \{0, 2, 7\}, G) &= (1, 0, 0) \\ T_5^2 &= \{5\} & \text{with } ADeg(\{5\}, \{0, 2, 7\}, G) &= (0, 0, 0) \end{aligned}$$

Clearly, vertex 1 does not generate a sequence of partitions compatible with the original one, since \mathcal{T}^2 is not compatible with \mathcal{S}^2 . Hence, *success* is set to FALSE and the orbit of vertex 1 is marked not valid.

Next the algorithm tries vertex 2, since its orbit is valid. After a vertex refinement, a new partition $\mathcal{T}^1 = (T_1^1, T_2^1)$ is obtained, where $W^1 = \{0, 1, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{1, 3, 7\} & \text{with } ADeg(\{1, 3, 7\}, \{2\}, G) &= (1, 0, 0) \\ T_2^1 &= \{0, 4, 5, 6\} & \text{with } ADeg(\{0, 4, 5, 6\}, \{2\}, G) &= (0, 0, 0) \end{aligned}$$

As expected, this partition is compatible with the original one \mathcal{S}^1 . Hence, a set refinement is performed, using $S_1^1 = \{1, 3, 7\}$ as the pivot cell. Thus we get a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2)$, where $W^2 = \{0, 1, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{1, 7\} & \text{with } ADeg(\{1, 7\}, \{1, 3, 7\}, G) &= (1, 0, 0) \\ T_2^2 &= \{3\} & \text{with } ADeg(\{3\}, \{1, 3, 7\}, G) &= (0, 0, 0) \end{aligned}$$

$$T_3^2 = \{0, 4, 5, 6\} \quad \text{with } ADeg(\{0, 4, 5, 6\}, \{1, 3, 7\}, G) = (1, 0, 0)$$

This partition is not compatible with \mathcal{S}^2 , so vertex 2 does not generate a sequence of partitions compatible with the original one. Hence, its orbit is marked not valid, and the algorithm goes on to vertex 3. However, this vertex belongs to the orbit of 2, which has just been discarded, so vertex 3 is also discarded without being tried. Next is vertex 4, which belongs to the orbit of 1, so it is also directly discarded. Thus we get to vertex 5.

A vertex refinement of partition \mathcal{S}^0 using 5 as the pivot vertex yields a new partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 1, 2, 3, 4, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{3, 4, 6\} & \text{with } ADeg(\{3, 4, 6\}, \{5\}, G) &= (1, 0, 0) \\ T_2^1 &= \{0, 1, 2, 7\} & \text{with } ADeg(\{0, 1, 2, 7\}, \{5\}, G) &= (0, 0, 0) \end{aligned}$$

The subsequent set refinement using $T_1^1 = \{3, 4, 6\}$ as the pivot cell yields a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2, T_4^2, T_5^2)$, where $W^2 = \{0, 1, 2, 3, 4, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{3, 4\} & \text{with } ADeg(\{3, 4\}, \{3, 4, 6\}, G) &= (1, 0, 0) \\ T_2^2 &= \{6\} & \text{with } ADeg(\{6\}, \{3, 4, 6\}, G) &= (0, 0, 0) \\ T_3^2 &= \{0\} & \text{with } ADeg(\{0\}, \{3, 4, 6\}, G) &= (2, 0, 0) \\ T_4^2 &= \{2, 7\} & \text{with } ADeg(\{2, 7\}, \{3, 4, 6\}, G) &= (1, 0, 0) \\ T_5^2 &= \{1\} & \text{with } ADeg(\{1\}, \{3, 4, 6\}, G) &= (0, 0, 0) \end{aligned}$$

This is not compatible with partition \mathcal{S}^2 and, hence, vertex 5 can not generate a sequence of partitions compatible with the original one, so the orbit of vertex 5 is marked not valid, and the algorithm proceeds to vertex 6.

A vertex refinement using 6 as the pivot vertex yields a partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 1, 2, 3, 4, 5, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{0, 5, 7\} & \text{with } ADeg(\{0, 5, 7\}, \{6\}, G) &= (1, 0, 0) \\ T_2^1 &= \{1, 2, 3, 4\} & \text{with } ADeg(\{1, 2, 3, 4\}, \{6\}, G) &= (0, 0, 0) \end{aligned}$$

Refining this partition by set, using the pivot cell $\{0, 5, 7\}$, yields a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2)$, where $W^2 = \{0, 1, 2, 3, 4, 5, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{0, 5, 7\} & \text{with } ADeg(\{0, 5, 7\}, \{0, 5, 7\}, G) &= (0, 0, 0) \\ T_2^2 &= \{1, 4\} & \text{with } ADeg(\{1, 4\}, \{0, 5, 7\}, G) &= (2, 0, 0) \\ T_3^2 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{0, 5, 7\}, G) &= (1, 0, 0) \end{aligned}$$

Since this partition is compatible with the original one, the algorithm proceeds to the next level. Since $P^2 = 2$ and $R^2 = \text{SET}$, a set refinement is performed using $T_2^2 = \{1, 4\}$ as the pivot cell. Thus, a new partition $\mathcal{T}^3 = (T_1^3, T_2^3, T_3^3, T_4^3)$ is obtained, where $W^3 = \{0, 1, 2, 3, 4, 5, 7\}$, and:

$$\begin{aligned} T_1^3 &= \{0\} & \text{with } ADeg(\{0\}, \{1, 4\}, G) &= (2, 0, 0) \\ T_2^3 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{1, 4\}, G) &= (1, 0, 0) \\ T_3^3 &= \{1, 4\} & \text{with } ADeg(\{1, 4\}, \{1, 4\}, G) &= (0, 0, 0) \\ T_4^3 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{1, 4\}, G) &= (1, 0, 0) \end{aligned}$$

Again, this partition is compatible with the original one, so a new refinement is performed,

using 0 as the pivot vertex, since $P^3 = 1$ and $R^3 = \text{VERTEX}$. This yields a new partition $T^4 = (T_1^4, T_2^4, T_3^4)$, where $W^4 = \{1, 2, 3, 4, 5, 7\}$, and:

$$\begin{aligned} T_1^4 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{0\}, G) &= (0, 0, 0) \\ T_2^4 &= \{1, 4\} & \text{with } ADeg(\{1, 4\}, \{0\}, G) &= (1, 0, 0) \\ T_3^4 &= \{2, 3\} & \text{with } ADeg(\{2, 3\}, \{0\}, G) &= (0, 0, 0) \end{aligned}$$

This partition is compatible with the original \mathcal{S}^4 . At the next step, we find that $P^4 = 1$, and $R^4 = \text{VERTEX}$, although initially it was **BACKTRACK**. Recall that it was changed when looking for automorphisms at this level. Hence, a vertex refinement is performed using any of the vertices in cell $T_1^4 = \{5, 7\}$. Let us choose vertex 5 for this example. Then, we obtain a new partition $T^5 = (T_1^5, T_2^5, T_3^5, T_4^5, T_5^5)$, where $W^4 = \{1, 2, 3, 4, 7\}$, and:

$$\begin{aligned} T_1^5 &= \{7\} & \text{with } ADeg(\{7\}, \{5\}, G) &= (0, 0, 0) \\ T_2^5 &= \{4\} & \text{with } ADeg(\{4\}, \{5\}, G) &= (1, 0, 0) \\ T_3^5 &= \{1\} & \text{with } ADeg(\{1\}, \{5\}, G) &= (0, 0, 0) \\ T_4^5 &= \{3\} & \text{with } ADeg(\{3\}, \{5\}, G) &= (1, 0, 0) \\ T_5^5 &= \{2\} & \text{with } ADeg(\{2\}, \{5\}, G) &= (0, 0, 0) \end{aligned}$$

This partition is compatible with \mathcal{S}^5 . Since T^5 and \mathcal{S}^5 are the final partitions in their respective sequences of partitions, it must be tested that mapping vertex 7 to 4, 4 to 7, 1 to 5, 3 to 2, and 2 to 3 is an isomorphism of G_{V^5} and G_{W^5} . This is best illustrated in Figure 6.2.

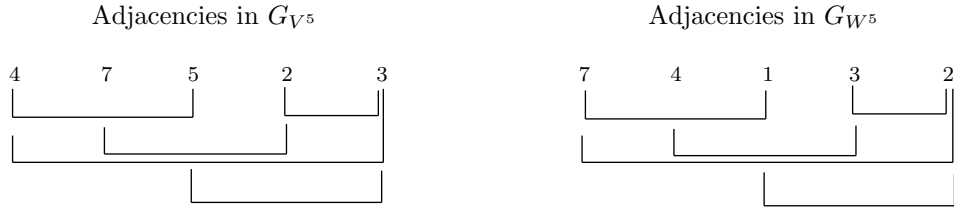


Figure 6.2: Adjacencies in G_{V^5} and G_{W^5} .

It is easy to see that the mapping defined above is an isomorphism of G_{V^5} and G_{W^5} . Furthermore, it leads to an automorphism of G . The alternative sequence of partitions generated by Algorithm 9 is compatible with the original one, and induces the following order on the vertices of graph G : $(6, 0, 5, 7, 4, 1, 3, 2)$. The original sequence of partitions induces the following order: $(0, 6, 1, 4, 7, 5, 2, 3)$. Algorithm 8 is applied to merge the corresponding orbits. This yields a new orbit partition $\mathcal{O} = \{\{0, 6\}, \{1, 4, 5, 7\}, \{2, 3\}\}$.

Next, vertex 7 is considered, but discarded since it is in the orbit of 1, which was previously discarded. Finally, since not all the vertices in the only cell in \mathcal{S}^0 are equivalent, R^0 remains with its original value **BACKTRACK**. The search for automorphisms in graph G has eliminated a backtracking point, and has classified the vertices in three orbits. In fact, that is the true orbit partition of this graph.

6.3.2 Search for Automorphisms in Graph H

As in the case of graph G , first the trivial orbit partition $\mathcal{O} = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$ is computed. Since the last partition in the sequence has no cells with no remaining links and

more than one vertex, Algorithm 7 does not merge any orbits. Then the original sequence of partitions is traversed backwards looking for automorphisms.

For $l = 4$, $R^4 = \text{VERTEX}$, so no alternative sequences of partitions will be generated. Likewise, there are no cells with no remaining links, so Algorithm 7 does not merge any orbits, either.

Since $R^3 = \text{BACKTRACK}$ and $P^3 = 1$, vertex 2 is used to generate an alternative sequence of partitions (recall that vertex 1 was the pivot vertex in the original sequence of partitions). Algorithm 9 generates a new partition $\mathcal{T}^4 = (T_1^4, T_2^4, T_3^4, T_4^4)$, where $W^4 = \{1, 3, 4, 5, 6\}$, and:

$$\begin{aligned} T_1^4 &= \{1\} && \text{with } ADeg(\{1\}, \{2\}, H) = (1, 0, 0) \\ T_2^4 &= \{5, 6\} && \text{with } ADeg(\{5, 6\}, \{2\}, H) = (0, 0, 0) \\ T_3^4 &= \{3\} && \text{with } ADeg(\{3\}, \{2\}, H) = (1, 0, 0) \\ T_4^4 &= \{4\} && \text{with } ADeg(\{4\}, \{2\}, H) = (0, 0, 0) \end{aligned}$$

This partition is compatible with \mathcal{S}^4 , and $R^4 = \text{VERTEX}$ and $P^4 = 4$, so a vertex refinement is performed with pivot vertex 4. This yields a new partition $\mathcal{T}^5 = (T_1^5, T_2^5, T_3^5, T_4^5)$, where $W^5 = \{1, 3, 5, 6\}$, and:

$$\begin{aligned} T_1^5 &= \{1\} && \text{with } ADeg(\{1\}, \{4\}, H) = (1, 0, 0) \\ T_2^5 &= \{5\} && \text{with } ADeg(\{5\}, \{4\}, H) = (1, 0, 0) \\ T_3^5 &= \{6\} && \text{with } ADeg(\{5\}, \{4\}, H) = (0, 0, 0) \\ T_4^5 &= \{3\} && \text{with } ADeg(\{3\}, \{4\}, H) = (1, 0, 0) \end{aligned}$$

This partition is compatible with \mathcal{S}^5 and, as it can be easily seen in Figure 6.3, matching the vertices in the corresponding cells we obtain an isomorphism of H_{V^5} and H_{W^5} . Furthermore, it extends to an automorphism of H , using the orders induced by both sequences of partitions, which are $(0, 7, 1, 3, 2, 6, 5, 4)$ and $(0, 7, 2, 4, 1, 5, 6, 3)$. Hence, the orbits of 1 and 2, of 3 and 4, 2 and 1, 5 and 6, and 4 and 3 are merged, obtaining a new orbit partition $\{\{0\}, \{1, 2\}, \{3, 4\}, \{5, 6\}, \{7\}\}$.



Figure 6.3: Adjacencies in H_{V^5} and H_{W^5} .

Next, R^3 is changed from BACKTRACK to VERTEX , since all the vertices in the original pivot cell $\{1, 2\}$ are equivalent. Finally, Algorithm 7 is called, but since there are no cells in \mathcal{S}^3 with more than one vertex and no remaining links, this has no effect on the orbit partition of the graph.

$R^2 = \text{VERTEX}$. Hence, the only process at this level is performed by Algorithm 7, but it does not merge any orbits, since there are no cells without links. For $l = 1$, the situation is analogous to that of $l = 2$, with the exception that $R^1 = \text{SET}$, while $R^2 = \text{VERTEX}$. Thus we come to level $l = 0$.

This is a backtracking point, so there will be a search for alternative sequences of partitions. Since there are 5 orbits in this moment, at most 4 vertices will be tested. Note that 7 tests would be necessary if we had no knowledge of orbits. Since the graph is regular, the first vertex refinement with the alternative pivot vertex will always yield a partition that is compatible with

the original one. In this example, the vertices will be tested in lexicographical order. Since vertex 0 was used in the original sequence of partitions, we will start with vertex 1.

After a vertex refinement using vertex 1 as the pivot, we obtain a partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{0, 2, 4\} & \text{with } ADeg(\{0, 2, 4\}, \{1\}, H) &= (1, 0, 0) \\ T_2^1 &= \{3, 5, 6, 7\} & \text{with } ADeg(\{3, 5, 6, 7\}, \{1\}, H) &= (0, 0, 0) \end{aligned}$$

Then, refining this partition by set using T_1^1 as the pivot cell, we get a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2, T_4^2, T_5^2)$, where $W^2 = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{0, 2\} & \text{with } ADeg(\{0, 2\}, \{0, 2, 4\}, H) &= (1, 0, 0) \\ T_2^2 &= \{4\} & \text{with } ADeg(\{4\}, \{0, 2, 4\}, H) &= (0, 0, 0) \\ T_3^2 &= \{3\} & \text{with } ADeg(\{3\}, \{0, 2, 4\}, H) &= (2, 0, 0) \\ T_4^2 &= \{5, 7\} & \text{with } ADeg(\{5, 7\}, \{0, 2, 4\}, H) &= (1, 0, 0) \\ T_5^2 &= \{6\} & \text{with } ADeg(\{6\}, \{0, 2, 4\}, H) &= (0, 0, 0) \end{aligned}$$

This partition is not compatible with \mathcal{S}^2 , so, following this path, it is not possible to generate a sequence of partitions compatible with the original one. Hence the orbit of vertex 1 is marked not valid. Thus, we proceed to vertex 2. However, vertex 2 is in the orbit of vertex 1, which has just been discarded. Hence, vertex 2 is not considered, and the algorithm goes on to vertex 3. After the corresponding vertex refinement we get a partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{2, 4, 6\} & \text{with } ADeg(\{2, 4, 6\}, \{3\}, H) &= (1, 0, 0) \\ T_2^1 &= \{0, 1, 5, 7\} & \text{with } ADeg(\{0, 1, 5, 7\}, \{3\}, H) &= (0, 0, 0) \end{aligned}$$

Then, refining by set this partition using cell $\{2, 4, 6\}$ as the pivot cell, a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2)$ is obtained, where $W^2 = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{2, 4, 6\} & \text{with } ADeg(\{2, 4, 6\}, \{2, 4, 6\}, H) &= (0, 0, 0) \\ T_2^2 &= \{1, 5\} & \text{with } ADeg(\{1, 5\}, \{2, 4, 6\}, H) &= (2, 0, 0) \\ T_3^2 &= \{0, 7\} & \text{with } ADeg(\{0, 7\}, \{2, 4, 6\}, H) &= (1, 0, 0) \end{aligned}$$

Unfortunately this partition is not compatible with \mathcal{S}^2 , so this alternative sequence of partitions will not be compatible with the original one. Hence, the orbit of vertex 3 is also marked as not valid, and the algorithm goes on to vertex 4. However, since this vertex is in the orbit of 3, it is directly discarded, and vertex 5 will be the next considered. A vertex refinement using vertex 5 as the pivot yields a new partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 1, 2, 3, 4, 6, 7\}$, and:

$$\begin{aligned} T_1^1 &= \{4, 6, 7\} & \text{with } ADeg(\{4, 6, 7\}, \{5\}, H) &= (1, 0, 0) \\ T_2^1 &= \{0, 1, 2, 3\} & \text{with } ADeg(\{0, 1, 2, 3\}, \{5\}, H) &= (0, 0, 0) \end{aligned}$$

Then a set refinement is performed using cell $\{4, 6, 7\}$ as the pivot set, thus obtaining a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2, T_4^2, T_5^2)$, where $W^2 = \{0, 1, 2, 3, 4, 6, 7\}$, and:

$$\begin{aligned} T_1^2 &= \{6, 7\} & \text{with } ADeg(\{6, 7\}, \{4, 6, 7\}, H) &= (1, 0, 0) \\ T_2^2 &= \{4\} & \text{with } ADeg(\{4\}, \{4, 6, 7\}, H) &= (0, 0, 0) \end{aligned}$$

$$\begin{aligned}
T_3^2 &= \{3\} & \text{with } ADeg(\{3\}, \{4, 6, 7\}, H) &= (2, 0, 0) \\
T_4^2 &= \{0, 1\} & \text{with } ADeg(\{0, 1\}, \{4, 6, 7\}, H) &= (1, 0, 0) \\
T_5^2 &= \{2\} & \text{with } ADeg(\{2\}, \{4, 6, 7\}, H) &= (0, 0, 0)
\end{aligned}$$

This partition is not compatible with \mathcal{S}^2 , so this vertex does not generate a sequence of partitions compatible with the original one. Hence its orbit is marked not valid, and the algorithm switches to vertex 6. However, since vertex 6 belongs to the orbit of 5, and vertex 5 has been discarded already, vertex 6 is directly discarded, proceeding to vertex 7.

With vertex 7 as the pivot, the algorithm performs a vertex refinement that yields a new partition $\mathcal{T}^1 = (T_1^1, T_2^1)$, where $W^1 = \{0, 1, 2, 3, 4, 5, 6\}$, and:

$$\begin{aligned}
T_1^1 &= \{0, 5, 6\} & \text{with } ADeg(\{0, 5, 6\}, \{7\}, H) &= (1, 0, 0) \\
T_2^1 &= \{1, 2, 3, 4\} & \text{with } ADeg(\{1, 2, 3, 4\}, \{7\}, H) &= (0, 0, 0)
\end{aligned}$$

Then it is refined by set using cell $\{0, 5, 6\}$ as the pivot set. This generates a new partition $\mathcal{T}^2 = (T_1^2, T_2^2, T_3^2)$, where $W^2 = \{0, 1, 2, 3, 4, 5, 6\}$, and:

$$\begin{aligned}
T_1^2 &= \{5, 6\} & \text{with } ADeg(\{5, 6\}, \{0, 5, 6\}, H) &= (1, 0, 0) \\
T_2^2 &= \{0\} & \text{with } ADeg(\{0\}, \{0, 5, 6\}, H) &= (0, 0, 0) \\
T_3^2 &= \{1, 2, 3, 4\} & \text{with } ADeg(\{1, 2, 3, 4\}, \{0, 5, 6\}, H) &= (1, 0, 0)
\end{aligned}$$

Since this partition is compatible with \mathcal{S}^2 , and $R^2 = \text{VERTEX}$ and $P^2 = 2$, a vertex refinement is performed using vertex 0 as the pivot. Thus we get a partition $\mathcal{T}^3 = (T_1^3, T_2^3, T_3^3)$, where $W^2 = \{1, 2, 3, 4, 5, 6\}$, and:

$$\begin{aligned}
T_1^3 &= \{5, 6\} & \text{with } ADeg(\{5, 6\}, \{0\}, H) &= (0, 0, 0) \\
T_2^3 &= \{1, 2\} & \text{with } ADeg(\{1, 2\}, \{0\}, H) &= (1, 0, 0) \\
T_3^3 &= \{3, 4\} & \text{with } ADeg(\{3, 4\}, \{0\}, H) &= (0, 0, 0)
\end{aligned}$$

This partition is compatible with \mathcal{S}^3 . Although initially $R^3 = \text{BACKTRACK}$, it was changed to VERTEX before, during this search for automorphisms. Hence, a vertex refinement with any vertex in T_1^3 (recall that $P^3 = 1$) is performed. For this example, let us assume it is vertex 5. This yields a new partition $\mathcal{T}^4 = (T_1^4, T_2^4, T_3^4, T_4^4)$, where $W^4 = \{1, 2, 3, 4, 6\}$, and:

$$\begin{aligned}
T_1^4 &= \{6\} & \text{with } ADeg(\{6\}, \{5\}, H) &= (1, 0, 0) \\
T_2^4 &= \{1, 2\} & \text{with } ADeg(\{1, 2\}, \{5\}, H) &= (0, 0, 0) \\
T_3^4 &= \{4\} & \text{with } ADeg(\{4\}, \{5\}, H) &= (1, 0, 0) \\
T_4^4 &= \{3\} & \text{with } ADeg(\{3\}, \{5\}, H) &= (0, 0, 0)
\end{aligned}$$

Again, the partition obtained is compatible with the original one. Since $R^4 = \text{VERTEX}$ and $P^4 = 4$, a vertex refinement is performed using vertex 3 as the pivot. This generates a new partition $\mathcal{T}^5 = (T_1^5, T_2^5, T_3^5, T_4^5)$, where $W^5 = \{1, 2, 4, 6\}$, and:

$$\begin{aligned}
T_1^5 &= \{6\} & \text{with } ADeg(\{6\}, \{3\}, H) &= (1, 0, 0) \\
T_2^5 &= \{2\} & \text{with } ADeg(\{2\}, \{3\}, H) &= (1, 0, 0) \\
T_3^5 &= \{1\} & \text{with } ADeg(\{1\}, \{3\}, H) &= (0, 0, 0) \\
T_4^5 &= \{4\} & \text{with } ADeg(\{4\}, \{3\}, H) &= (1, 0, 0)
\end{aligned}$$

This last partition is compatible with the original one, and mapping vertex 6 to vertex 2, vertex

2 to vertex 6, vertex 1 to vertex 5, and vertex 4 to vertex 4, we get an isomorphism of H_{W^5} and H_{V^5} , as it can be easily seen in Figure 6.4.



Figure 6.4: Adjacencies in H_{V^5} and H_{W^5} .

A sequence of partitions compatible with the original one has been found, and the orbits of the corresponding vertices in the orders induced by both sequences of partitions must be merged. The order induced by the original sequence of partitions was $(0, 7, 1, 3, 2, 6, 5, 4)$, and the order induced by the alternative sequence of partitions is $(7, 0, 5, 3, 6, 2, 1, 4)$. Hence, the orbits of 0 and 7 are merged, and also the orbits of 1 and 5. The other merges are redundant, though Algorithm 8 performs all of them. Thus we get a new orbit partition $\{\{0, 7\}, \{1, 2, 5, 6\}, \{3, 4\}\}$.

Since not all the vertices in the pivot cell are equivalent, P^0 remains with its initial value BACKTRACK. Besides, Algorithm 7 is called, but it has no effect, since there are no cells with more than one vertex and without links. Nevertheless, the search for automorphisms has eliminated one backtracking point, and has classified the vertices in the graph in three orbits. This will certainly reduce the search space when looking for a sequence of partitions compatible with the chosen target.

6.3.3 Match Graphs G and H

Once both graphs have been searched for automorphisms, Algorithm 5 chooses one of them as the target (in this case G), and calls Algorithm 10 trying to find a sequence of partitions for graph H that is compatible with the one for graph G . If it is possible to find one such sequence of partitions, it returns TRUE, whereas if it is not possible to find it, it returns FALSE.

Algorithm *Match2* is called with parameters 0 for the starting level, G for the graph whose sequence of partitions is $SeqPart(Q'_G)$, and H for the graph whose initial partition is $\mathcal{D}_H = (\{0, 1, 2, 3, 4, 5, 6, 7\})$ and whose known orbits are $Orbits(Q'_H) = \{\{0, 7\}, \{1, 2, 5, 6\}, \{3, 4\}\}$.

The sequence of partitions for graph G was generated in Section 5.2.1. Then it was reviewed in Section 6.3.1, where the last backtracking point (for $l = 4$) was eliminated (turned into VERTEX). Recall also that $t = 5$ (the length of the sequence of partitions for graph G). Let us start executing Algorithm 10 with this data.

Since $l = 0 \neq 5 = t$, and $R^0 = \text{BACKTRACK}$, the vertices in \mathcal{D}_H will be tried in the search for a sequence of partitions compatible with Q_G . First it is tested whether the orbits are applicable at this level. Since it is the first level and no vertex has been fixed yet, they are applicable. Then, the three orbits in \mathcal{O}_H are marked valid, and the search starts. Although the algorithm does not impose any order on the vertices, we will assume a lexicographical order.

First, vertex 0 is chosen. Since orbits are applicable and the orbit of 0 is valid, a new partition $\mathcal{T}' = (T'_1, T'_2)$ is generated refining the initial partition $\mathcal{T} = \mathcal{D}_H$ by vertex using 0 as the pivot vertex, where $W' = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{1, 2, 7\} & \text{with } ADeg(\{1, 2, 7\}, \{0\}, H) &= (1, 0, 0) \\ T'_2 &= \{3, 4, 5, 6\} & \text{with } ADeg(\{3, 4, 5, 6\}, \{0\}, H) &= (0, 0, 0) \end{aligned}$$

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 $R^1 = \text{SET}$, a set refinement is performed using cell $\{1, 2, 7\}$ as the pivot set (recall that $P^1 = 1$). This yields a partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where $W' = \{1, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{1, 2\} && \text{with } ADeg(\{1, 2\}, \{1, 2, 7\}, H) = (1, 0, 0) \\ T'_2 &= \{7\} && \text{with } ADeg(\{7\}, \{1, 2, 7\}, H) = (0, 0, 0) \\ T'_3 &= \{3, 4, 5, 6\} && \text{with } ADeg(\{3, 4, 5, 6\}, \{1, 2, 7\}, H) = (1, 0, 0) \end{aligned}$$

This partition is not compatible with \mathcal{S}^1 . Hence, the algorithm returns FALSE, backtracking to the previous invocation. Next, the orbit of vertex 0 is marked as not valid and, lexicographically, vertex 1 is considered. Since it belongs to a valid orbit, a vertex refinement is performed using it as the pivot vertex. Thus we get a partition $\mathcal{T}' = (T'_1, T'_2)$, where $W' = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 2, 4\} && \text{with } ADeg(\{0, 2, 4\}, \{1\}, H) = (1, 0, 0) \\ T'_2 &= \{3, 5, 6, 7\} && \text{with } ADeg(\{3, 5, 6, 7\}, \{1\}, H) = (0, 0, 0) \end{aligned}$$

This partition is compatible with the target. Hence a recursive call is made to process the next partition. Since $R^1 = \text{SET}$ and $P^1 = 1$, the corresponding set refinement is performed, yielding a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where $W' = \{0, 2, 3, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{0, 2\} && \text{with } ADeg(\{0, 2\}, \{0, 2, 4\}, H) = (1, 0, 0) \\ T'_2 &= \{4\} && \text{with } ADeg(\{4\}, \{0, 2, 4\}, H) = (0, 0, 0) \\ T'_3 &= \{3\} && \text{with } ADeg(\{3\}, \{0, 2, 4\}, H) = (2, 0, 0) \\ T'_4 &= \{5, 7\} && \text{with } ADeg(\{5, 7\}, \{0, 2, 4\}, H) = (1, 0, 0) \\ T'_5 &= \{6\} && \text{with } ADeg(\{6\}, \{0, 2, 4\}, H) = (0, 0, 0) \end{aligned}$$

Since this partition is not compatible with \mathcal{S}^1 , the algorithm returns FALSE, backtracking to the previous invocation. Then the orbit of 1 is marked not valid. The next vertex to consider would be 2, but since it belongs to the orbit of 1, which has just been marked not valid, it is discarded, and the algorithm proceeds to vertex 3. This is the first point where the behavior of Algorithm 10 yields a difference with Algorithm 4. The vertex refinement with pivot vertex 3 generates a partition $\mathcal{T}' = (T'_1, T'_2)$ is obtained, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{2, 4, 6\} && \text{with } ADeg(\{2, 4, 6\}, \{3\}, H) = (1, 0, 0) \\ T'_2 &= \{0, 1, 5, 7\} && \text{with } ADeg(\{0, 1, 5, 7\}, \{3\}, H) = (0, 0, 0) \end{aligned}$$

Since this partition is compatible with the target, a recursive call is made to process the next partition in the sequence. Now, a set refinement will be performed with pivot set $\{2, 4, 6\}$. Thus we get a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{2, 4, 6\} && \text{with } ADeg(\{2, 4, 6\}, \{2, 4, 6\}, H) = (0, 0, 0) \\ T'_2 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{2, 4, 6\}, H) = (2, 0, 0) \\ T'_3 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{2, 4, 6\}, H) = (1, 0, 0) \end{aligned}$$

This partition is finally compatible with the target and a recursive call is made to proceed to the next partition in the sequence. Here, since $P^2 = 2$ and $R^2 = \text{SET}$, pivot cell $\{1, 5\}$ is used for a

set refinement, what yields a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4)$, where $W' = \{0, 1, 2, 4, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{4\} && \text{with } ADeg(\{4\}, \{1, 5\}, H) = (2, 0, 0) \\ T'_2 &= \{2, 6\} && \text{with } ADeg(\{2, 6\}, \{1, 5\}, H) = (1, 0, 0) \\ T'_3 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{1, 5\}, H) = (0, 0, 0) \\ T'_4 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{1, 5\}, H) = (1, 0, 0) \end{aligned}$$

Again, we get a compatible partition, and recursively proceed to the next partition. Here, $P^3 = 1$ and $R^3 = \text{VERTEX}$. Hence, using vertex 4, a vertex refinement is performed, and a new partition $\mathcal{T}' = (T'_1, T'_2, T'_3)$ is obtained, where $W' = \{0, 1, 2, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{2, 6\} && \text{with } ADeg(\{2, 6\}, \{4\}, H) = (0, 0, 0) \\ T'_2 &= \{1, 5\} && \text{with } ADeg(\{1, 5\}, \{4\}, H) = (1, 0, 0) \\ T'_3 &= \{0, 7\} && \text{with } ADeg(\{0, 7\}, \{4\}, H) = (0, 0, 0) \end{aligned}$$

This partition is also compatible with the target. Hence, a recursive call is made to process the next partition. Recall that R^4 was changed from BACKTRACK to VERTEX during the search for automorphisms. Then, a vertex refinement is performed, using any vertex in T_1 . Let us lexicographically choose vertex 2 as the pivot. Thus we get a partition $\mathcal{T}' = (T'_1, T'_2, T'_3, T'_4, T'_5)$, where $W' = \{0, 1, 5, 6, 7\}$, and:

$$\begin{aligned} T'_1 &= \{6\} && \text{with } ADeg(\{6\}, \{2\}, H) = (0, 0, 0) \\ T'_2 &= \{1\} && \text{with } ADeg(\{1\}, \{2\}, H) = (1, 0, 0) \\ T'_2 &= \{5\} && \text{with } ADeg(\{5\}, \{2\}, H) = (0, 0, 0) \\ T'_3 &= \{0\} && \text{with } ADeg(\{0\}, \{2\}, H) = (1, 0, 0) \\ T'_3 &= \{7\} && \text{with } ADeg(\{7\}, \{2\}, H) = (0, 0, 0) \end{aligned}$$

This partition is compatible with \mathcal{S}^5 . Since it is the last partition in the sequence, the adjacencies among the vertices in the different cells must be compared for both partitions. This partition is the same obtained by Algorithm 4, and the correspondence of vertices with the original partition was shown in Figure 5.2. The isomorphism found by both algorithms is the same. The difference achieved comes from the fact that vertex 3 has been discarded by Algorithm 10 without the need to generate its corresponding sequence of partitions.

6.4 Proof of correctness

In this section we will show that algorithm conauto-v0 correctly determines if two graphs are isomorphic or not. This algorithm may have eliminated some backtracking points that were present in algorithm sinauto. However, we will show that the certainty of finding a sequence of partitions compatible with the target, if it exists, still holds after this elimination. Besides, we will prove that discarding vertices due to vertex equivalence does not reduce the certainty of finding an isomorphism, if it exists.

Theorem 6.1 *Two graphs G and H are isomorphic if and only if $AreIsomorphic2(G, H)$ returns TRUE.*

Proof: First, Algorithm *AreIsomorphic2* 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, they 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 6.2, this equivalence holds.
2. It may have been found at some other level l' such that $l' > l$. In this case, from Lemma 6.3, it also holds at level l .
3. It may have been inferred applying Remark 6.1, in which case it also holds.

Since R^l is changed to VERTEX only at Line 26 in Algorithm 6, 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 6.4, 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 10, the orbit partition of graph H is used to prune the search at the remaining backtracking points. However, these equivalences among vertices are considered only in the case that all the vertices already discarded belonged to singleton orbits. From Observation 6.1, 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 6 and 10 do not eliminate paths in the search tree that might lead to an isomorphism of graphs G and H . Therefore, if Algorithm 10 returns FALSE, there is no sequence of partitions compatible with the target, and, from Lemma 5.1, graphs G and H are not isomorphic. If Algorithm 10 finds a sequence of partitions for one graph which is compatible with the one generated for the other graph, from Lemma 5.2, graphs G and H are isomorphic. Hence, Algorithm *AreIsomorphic2* returns TRUE if and only if graphs G and H are isomorphic. ■

6.5 Performance Evaluation

In this section, we compare the performance of algorithm *conauto-v0* with *sinauto*, *nauty-2.2*, and *vf2*. The tests have been carried out under the same environment described in Section 5.4 using the same benchmark.

Random graphs are not likely to have automorphisms. Therefore, adding automorphism detection and management should not have a noticeable effect in the algorithm. In fact, most

probably, no alternative sequences of partitions will be generated, since there will be no backtracking points in the sequences of partitions. This explains the results shown in Figure 6.5, where the plots for `sinauto` and `conauto-v0` overlap.

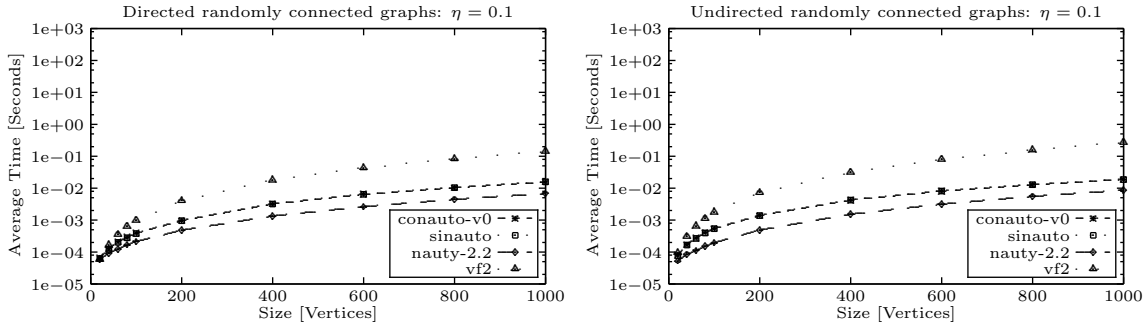


Figure 6.5: Performance of `conauto-v0` with isomorphic randomly connected graphs.

2D-meshes are graphs that have only a few automorphisms, and whose structure is quite symmetric. This makes their automorphisms easy to find. As it can be seen in Figure 6.6, `conauto-v0` improves the performance of `sinauto` for both the directed and the undirected versions of the graphs. In fact, for the undirected meshes of the biggest size tested (graphs of 1024 vertices) `conauto-v0` becomes the fastest. Like `sinauto`, `conauto-v0` offers a uniform behavior for both the directed and the undirected versions of the graphs.

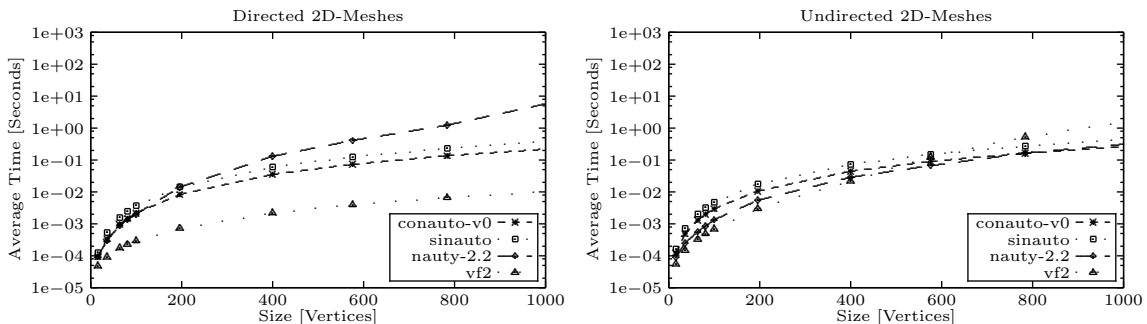


Figure 6.6: Performance of `conauto-v0` with isomorphic 2D-meshes.

The results for Miyazaki’s Furer gadgets are shown in Figures 6.7 and 6.8. The improvement achieved is spectacular. First, there is no exponential behavior in `conauto-v0`, for any of the cases considered. (Note that for the directed version of the graphs, `nauty` was unable to process the graphs of 40 vertices within the time limit imposed.) Second, the behavior of `conauto-v0` with isomorphic and non-isomorphic graphs is very similar, which was not the case with `sinauto`. Again, we have that `conauto-v0` has a quite uniform behavior. Last, we observe that the directed graphs are quite easier to process than their undirected versions for `conauto-v0`. That is because the directed version of the graphs gives more information than the undirected version, which allows a finer vertex classification, and helps in finding automorphisms.

In Figure 6.9, we show the results obtained for Paley graphs. Both for graphs of prime size and square prime size, `conauto-v0` is the fastest for the graphs of the biggest size considered (improving the performance of `sinauto`). Besides, it exhibits the most uniform behavior, although graphs of square prime size are harder than those of prime size (except for `vf2`).

Figure 6.10 shows the results for triangular and lattice graphs. Although `conauto-v0` improves

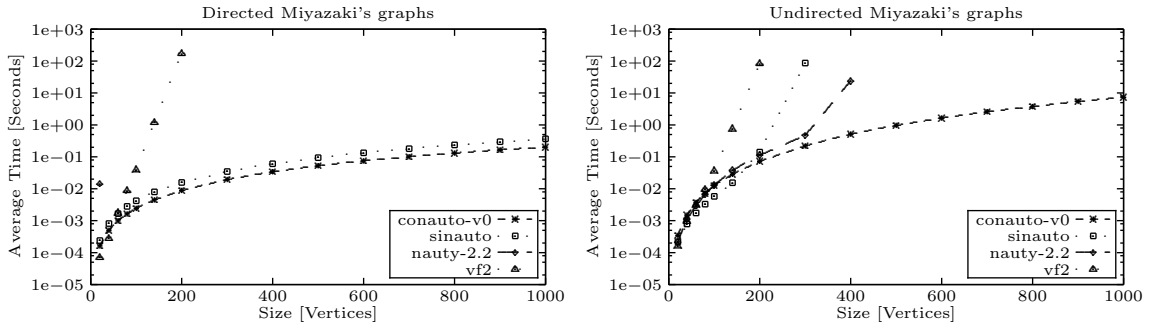


Figure 6.7: Performance of conauto-v0 with isomorphic Miyazaki's Furer gadgets.

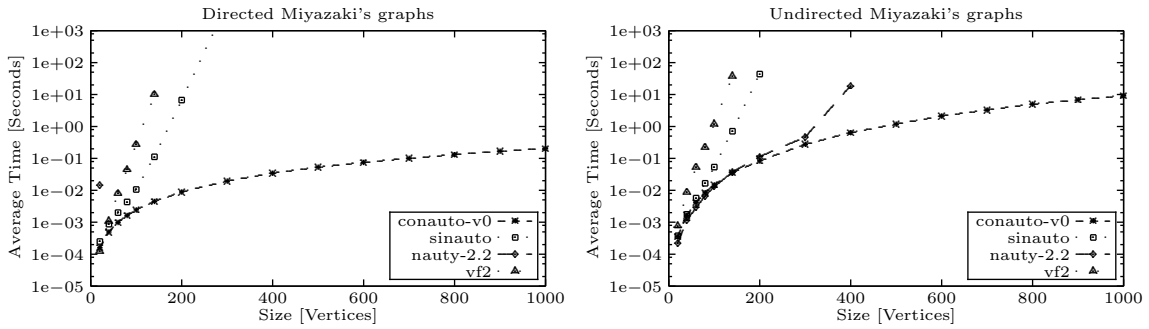


Figure 6.8: Performance of conauto-v0 with non-isomorphic Miyazaki's Furer gadgets.

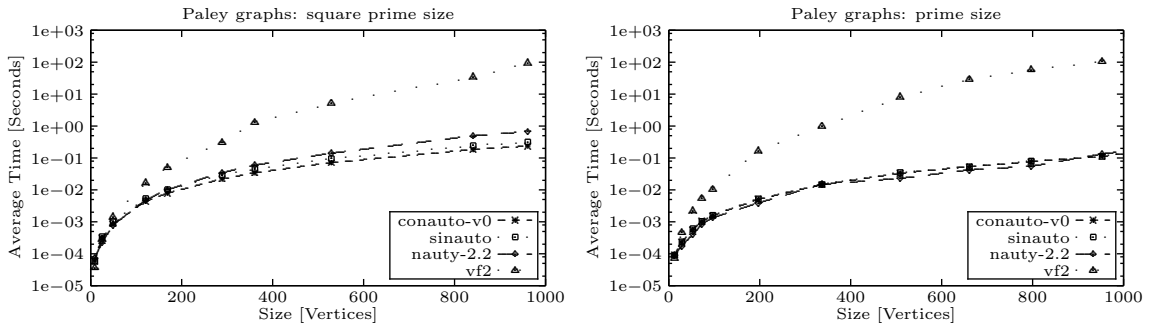


Figure 6.9: Performance of conauto-v0 with Paley graphs.

the results of sinauto for both families of graphs, it is roughly one order of magnitude slower than vf2. However, the results of conauto-v0 are much like those of nauty, what we consider a good result.

Latin square graphs seem to be a harder family of strongly regular graphs than those just considered (Paley, triangular, and lattice graphs). The results for these graphs are shown in Figure 6.11. For this family of graphs, conauto-v0 is twice slower than sinauto for the isomorphic graphs of the biggest size considered. This implies that looking for automorphisms in this family of graphs is quite time-consuming for conauto-v0, since the search for the compatible sequence of partitions can not be harder for conauto-v0 than it was for sinauto.

However, the results for non-isomorphic pairs of graphs show that, while sinauto was almost four orders of magnitude above nauty, conauto-v0 is only two orders of magnitude above nauty. This shows a considerable improvement. Nevertheless, we would like this performance to be further

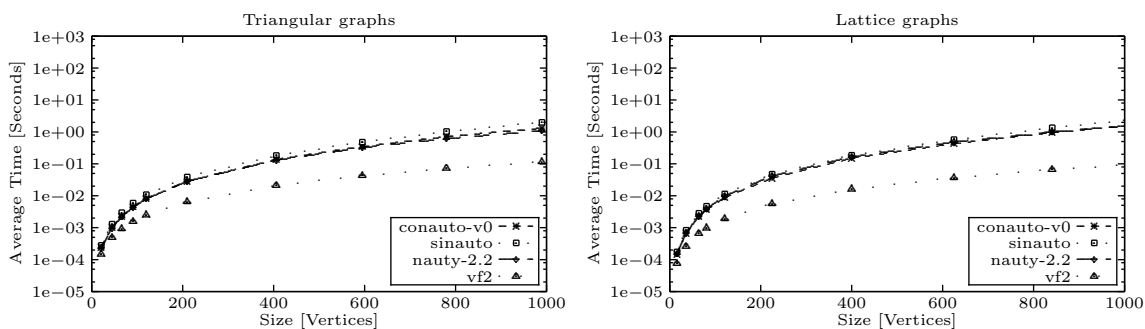


Figure 6.10: Performance of conauto-v0 with triangular and lattice graphs.

improved. Certainly, the automorphism management of nauty must be much more powerful than ours (but also harder to implement). Anywhere, these results show that we are in the good path.

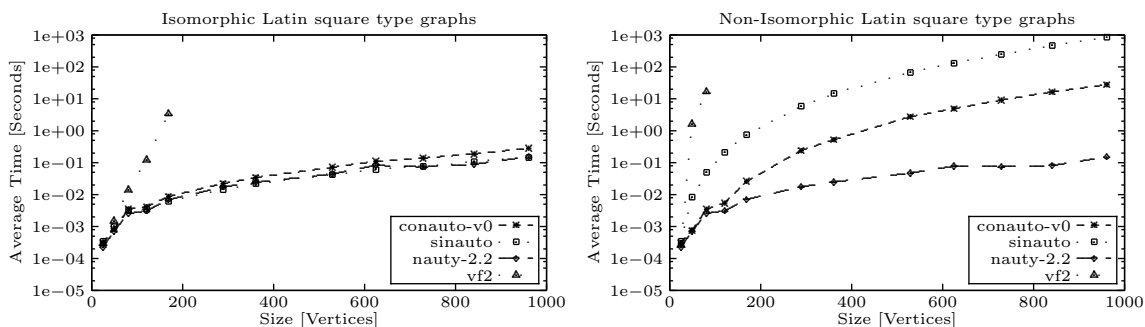


Figure 6.11: Performance of conauto-v0 with Latin square graphs.

The results for the unions of tripartite graphs are shown in Figures 6.12 and 6.13. This is a family of graphs that has many isomorphic components. This means that there are many automorphisms. However, this type of automorphisms is not easy to discover. As in the case of Latin square graphs, conauto-v0 is slower than sinauto for the isomorphic tests, but is faster for the non-isomorphic ones. It still seems exponential, though. This is due to the fact that it can not discover that two components are isomorphic and processes them again and again. The improvement achieved comes from the automorphisms found among vertices of the same component. Hence, something else must be done to make our algorithm faster.

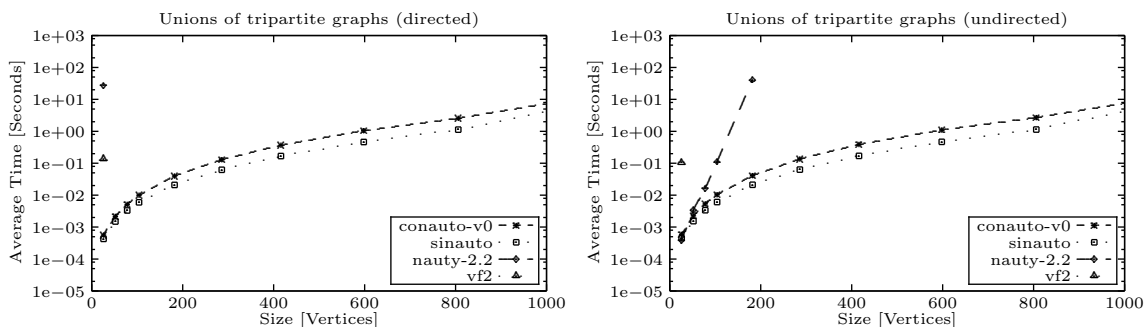


Figure 6.12: Performance of conauto-v0 with isomorphic unions of tripartite graphs.

Unions of strongly regular graphs are a very hard family of graphs that is similar to the previous

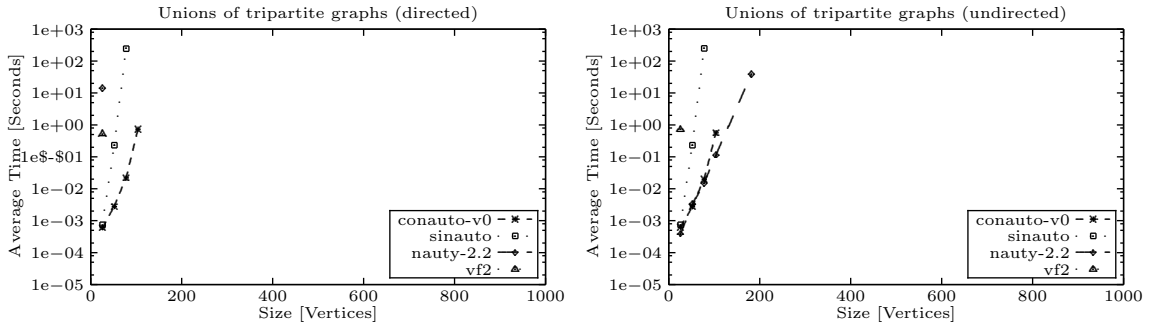


Figure 6.13: Performance of conauto-v0 with non-isomorphic unions of tripartite graphs.

one. As it is shown in Figure 6.14, conauto-v0 is slower than sinauto (but still polynomial) with positive tests, while it can not find an answer for graphs above 600 vertices for non-isomorphic pairs of graphs. The explanation is the same given for the unions of tripartite graphs: dealing with components in the graphs. This will be our main issue in the next version of the algorithm, conauto-v1.

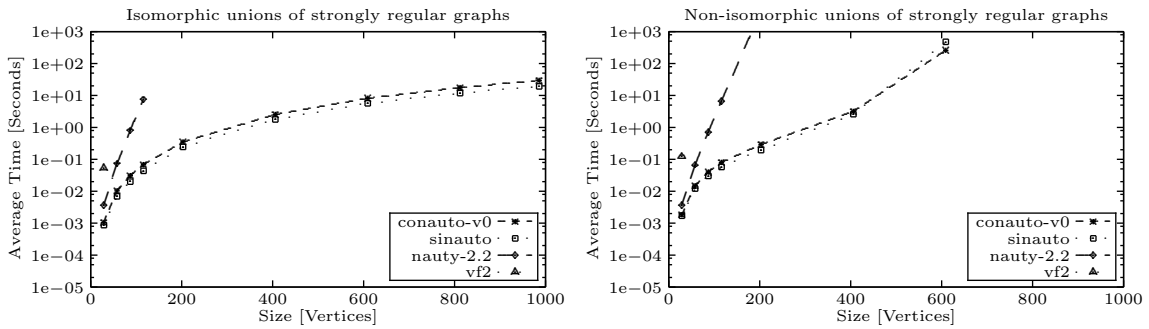


Figure 6.14: Performance of conauto-v0 with unions of strongly regular graphs.

The results for the point-line graphs of Desarguesian projective planes are shown in Figure 6.15. As it can be seen, conauto-v0 improves the results of sinauto. While sinauto was able to process graphs of up to 114 vertices, conauto-v0 can deal with graphs of 146 vertices. However, this is a very small improvement, and nauty goes one step beyond (graphs of 182 vertices). The tricky thing is that vf2 is as fast as nauty. If we were able to find an explanation for this, perhaps we would be able to improve our algorithm. However, we conjecture that the answer might be in finding more powerful refinements (based on more powerful invariants).

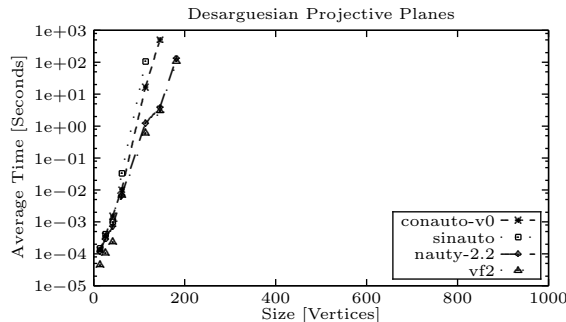


Figure 6.15: Performance of conauto-v0 with point-line graphs of Desarguesian projective planes.

Chapter 7

Detecting Components

Algorithm `conauto-v0` has overcome some of the weaknesses of `sinauto`. However, there are still some graph families in our benchmark that are hard for `conauto-v0`. Among them are the ones that are built as unions of simple components. These are the unions of tripartite graphs and the unions of strongly regular graphs. While the positive tests for these graphs run fast, the negative ones become very much harder.

However, there are some hints that should help us find a solution for that problem. For instance, in the case of graphs that are the disjoint union of connected components, when matching a graph, once a component of one graph has been found isomorphic to a component of the other graph, it is of no use trying to match that same component to another component of the other graph. Besides, if a component of a graph can not be matched against any component of the other graph, it is of no use trying to match the rest of components, since, at the end, the graphs can not be isomorphic. We have a particular interest in this kind of graphs, since they are especially hard for `nauty`, and we want to find a way to overcome this handicap of `nauty`.

After a thorough study of the behavior of `conauto-v0` for these graph families, we have concluded that its performance can be drastically improved in these cases by directly applying the following theorem:

Theorem 7.1 *During the search for a sequence of partitions compatible with the target, backtracking from a level l to a level $k < l$, such that each cell of level l is contained in a different cell of level k , can not provide a compatible partition.*

Note that, when backtracking, the algorithm goes back until it reaches a previous backtracking point, to try an alternative path in the search tree. Hence, in order to reduce the search space, we want to backtrack directly to the highest backtracking point possible. With Theorem 7.1, in some cases it will be possible to skip some backtracking points that are already known to be unable to lead to a solution, hence improving performance.

Next we prove Theorem 7.1. Then, we describe algorithm `conauto-v1` which applies Theorem 7.1. We conclude this chapter with the evaluation of the practical performance of the new algorithm `conauto-v1`, which meets our expectations.

7.1 Proving Theorem 7.1

A backtracking point 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 a new concept that will be useful in the following discussion.

Definition 7.1 Let $G = (V, R)$ be a graph, and let $\mathcal{S} = (S_1, \dots, S_r)$ be a partition of V . \mathcal{S} is said to be equitable (with respect to G) if for all $i \in \{1, \dots, r\}$, for all $u, v \in S_i$, for all $j \in \{1, \dots, r\}$, $ADeg(u, S_j, G) = ADeg(v, S_j, G)$.

Observation 7.1 The partition at a backtracking point 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 backtracking point, no set refinement has succeeded. ■

Observation 7.2 Let l be a backtracking level. Let $\mathcal{S}^l = (S_1^l, \dots, S_r^l)$ be the partition at that level. Then, for all $i \in \{1, \dots, r\}$, $G_{S_i^l}$ is regular.

Proof: From Observation 7.1, \mathcal{S}^l is equitable. Fix $i \in \{1, \dots, r\}$, then, from Definition 22, for all $u, v \in S_i^l$, $ADeg(u, S_i^l, G) = ADeg(v, S_i^l, G)$. Therefore, $G_{S_i^l}$ is regular, for all $i \in \{1, \dots, r\}$. ■

Let $\mathbf{Q} = (\mathbf{S}, \mathbf{R}, \mathbf{P})$ be a sequence of partitions for graph $G = (V, R)$ where $\mathbf{S} = (\mathcal{S}^0, \dots, \mathcal{S}^t)$, $\mathbf{R} = (R^0, \dots, R^{t-1})$, and $\mathbf{P} = (P^0, \dots, P^{t-1})$. For all $i \in \{0, \dots, t\}$ let $\mathcal{S}^i = (S_1^i, \dots, S_{r_i}^i)$, and $V^i = \bigcup_{j=1}^{r_i} S_j^i$. We consider two backtracking levels k and l that satisfy the preconditions of Theorem 7.1, i.e., $k < l$ and each cell of \mathcal{S}^l is contained in a different cell of \mathcal{S}^k .

Let $p \in S_{P^k}^k$ be the pivot vertex used for the vertex refinement at level k . Assume there is a vertex $q \in S_{P^k}^k, q \neq p$ that satisfies the following. $\mathcal{T}^{k+1} = \text{VertexRefinement}(\mathcal{S}^k, q, G_{V^k})$ is a partition that is compatible with \mathcal{S}^{k+1} . Let $\mathcal{T}^{k+1} = (T_1^{k+1}, \dots, T_{r_{k+1}}^{k+1})$, $W^{k+1} = \bigcup_{j=1}^{r_{k+1}} T_j^{k+1}$. For all $i \in \{k+2, \dots, l\}$, let $\mathcal{T}^i = (T_1^i, \dots, T_{r_i}^i)$ be compatible with \mathcal{S}^i , where $W^i = \bigcup_{j=1}^{r_i} T_j^i$, $\mathcal{T}^i = \text{SetRefinement}(\mathcal{T}^{i-1}, T_{P^{i-1}}^{i-1}, G_{W^{i-1}})$ if $R^{i-1} = \text{SET}$, and $\mathcal{T}^i = \text{VertexRefinement}(\mathcal{T}^{i-1}, v, G_{W^{i-1}})$ for some $v \in T_{P^{i-1}}^{i-1}$ if $R^{i-1} \neq \text{SET}$. This generates an alternative sequence of partitions that is compatible with the original one up to level l .

Under these premises, we show in the rest of the section that G_{V^l} and G_{W^l} are isomorphic, and there is an isomorphism of them that matches the vertices in S_i^l to the vertices in T_i^l for all $i \in \{1, \dots, r_l\}$.

To simplify the notation, let us assume $r_k = r_l = r$. Note that in this case, for all $i \in \{1, \dots, r\}$, $S_i^l \subseteq S_i^k$. In case $r_k \neq r_l$ this correspondence is not trivial. However, we can safely assume that there may be some $S_i^l \in \mathcal{S}^l$ that are empty, and develop our argument considering this possibility, although we know that in the real sequence of partitions, these empty cells would have been discarded.

For all $i \in \{1, \dots, r\}$, let $E_i = S_i^k \setminus S_i^l$, $E'_i = S_i^k \setminus T_i^l$ be the vertices discarded in the refinements from S_i^k to S_i^l and T_i^l respectively, let $A_i = E_i \cap E'_i$ be the vertices discarded in both alternative refinements, $B_i = E_i \setminus A_i$ the vertices discarded only in the refinement from S_i^k to S_i^l , $C_i = E'_i \setminus A_i$

the vertices discarded only in the refinement from S_i^k to T_i^l , and $D = S_i^l \cap T_i^l$ the vertices remaining in both alternative partitions at level l . Let $A = \bigcup_{i=1}^r A_i$, $B = \bigcup_{i=1}^r B_i$, $C = \bigcup_{i=1}^r C_i$, $D = \bigcup_{i=1}^r D_i$, $E = \bigcup_{i=1}^r E_i$, and $E' = \bigcup_{i=1}^r E'_i$. Clearly, $E = A \cup B$, and $E' = A \cup C$. Observe that $|E_i| = |E'_i|$, and hence $|B_i| = |C_i|$ for all $i \in \{1, \dots, r\}$.



Figure 7.1: Partition of S_i^k into subsets A_i , B_i , C_i , and D_i for all $i \in \{1, \dots, r\}$.

Observation 7.3 G_E is isomorphic to $G_{E'}$, and there is an isomorphism of them that matches the vertices in E_i to those in E'_i , for all $i \in \{1, \dots, r\}$.

Proof: Direct from the construction of the sequences of partitions. ■

Lemma 7.1 Let $M = \text{Adj}(G)$. It is satisfied that:

- For each $u \in E$, for all $i \in \{1, \dots, r\}$, for all $v, w \in S_i^l$, $M_{uv} = M_{uw}$ and $M_{vu} = M_{wu}$.
- For each $u \in E'$, for all $i \in \{1, \dots, r\}$, for all $v, w \in T_i^l$, $M_{uv} = M_{uw}$ and $M_{vu} = M_{wu}$.

Proof: Since none of the vertices in E has been able to distinguish among the vertices in cell S_i^l , each of the discarded vertices has the same type of adjacency with all the vertices in S_i^l . Otherwise, consider vertex $u \in E$. Assume u has at least two different types of adjacency with the vertices in S_i^l . Since it was discarded during the refinements from S_i^k to S_i^l , that had to be for one of the following reasons:

1. It was discarded for having no links (i.e. links of type 0), what is impossible since it has two different types of adjacencies with the vertices in S_i^l .
2. It was used as the pivot set in a vertex refinement, what is impossible since it would have been able to split cell S_i^l .

The same argument applies to the vertices in E' with respect to the vertices in each cell T_i^l . ■

Consider the adjacency between vertex u and vertex v is $M_{uv} = a$ for some $a \in \{0, \dots, 3\}$. Then, we will denote the adjacency between v and u (M_{vu}) as a^{-1} . Note that if $a = 0$, $a^{-1} = 0$, if $a = 1$, $a^{-1} = 2$, if $a = 2$, $a^{-1} = 1$, and if $a = 3$, $a^{-1} = 3$.

Lemma 7.2 For each $i, j \in \{1, \dots, r\}$, there is some $a \in \{0, \dots, 3\}$ such that for all $u \in B_i$, $v \in C_j$, $w \in D_i$, $u' \in B_j$, $v' \in C_j$, and $w' \in D_j$, $M_{uv'} = M_{uw'} = M_{vu'} = M_{v'w'} = M_{w'u'} = M_{w'v'} = a$ and $M_{u'v} = M_{u'w} = M_{v'u} = M_{v'w} = M_{w'u} = M_{w'v} = a^{-1}$.

Proof: Let us take any $i \in \{1, \dots, r\}$ and any $j \in \{1, \dots, r\}$. Since $B_i \subseteq E$ and $C_j \subseteq S_j^l$, from Lemma 7.1, for each $u \in B_i$, for all $v' \in C_j$, $M_{uv'} = a$ for some $a \in \{0, \dots, 3\}$. Let us take any such $v' \in C_j$. Then, $M_{v'u} = a^{-1}$ for those particular v' and u . Besides, since $C_j \subseteq E'$ and $B_i \subseteq T_i^l$, from Lemma 7.1, for all $u \in B_i$, $M_{v'u} = b$ for some $b \in \{0, \dots, 3\}$. Since we already know that $M_{v'u} = a^{-1}$ for that particular pair of vertices, then we conclude that for all $u \in B_i$, $v' \in C_j$, $M_{uv'} = a$ and $M_{v'u} = a^{-1}$, for some $a \in \{0, \dots, 3\}$.

$S_j^l = C_j \cup D_j$ and $B_i \subseteq E$. Since for all $u \in B_i$, $v' \in C_j$, $M_{uv'} = a$ and $M_{v'u} = a^{-1}$, then from Lemma 7.1, for all $u \in B_i$, $w' \in D_j$, $M_{uw'} = a$ (clearly, the same a) and $M_{w'u} = a^{-1}$.

$T_i^l = B_i \cup D_i$ and $C_j \subseteq E'$. Since for all $u \in B_i, v' \in C_j, M_{uv'} = a$ and $M_{v'u} = a^{-1}$, then from Lemma 7.1, for all $v' \in C_j, w \in D_i, M_{v'w} = a^{-1}$ and $M_{ww'} = a$ (clearly, the same a).

Furthermore, all the vertices in $S_j^l = C_j \cup D_j$ have the same number of adjacent vertices of each type in $E_i = A_i \cup B_i$. Otherwise, they would have been distinguished in the refinement process from \mathcal{S}^k to \mathcal{S}^l . Likewise, all the vertices in $T_j^l = B_j \cup D_j$ have the same number of adjacent vertices of each type in $E'_i = A_i \cup C_i$. Otherwise, they would have been distinguished in the refinement process from \mathcal{S}^k to \mathcal{T}^l . Hence, the vertices of D_j must have the same number of adjacent vertices of each type in B_i and C_i . Hence, since for all $w' \in D_j$, and for all $u \in B_i, M_{uw'} = a$ and $M_{w'u} = a^{-1}$, then for all $w' \in D_j$, and for all $v \in C_i, M_{vw'} = a$ and $M_{w'v} = a^{-1}$ too.

A similar argument may be used to prove that for all $w \in D_i$, and for all $u' \in B_j, M_{wu'} = a$ and $M_{u'w} = a^{-1}$. Then, from Lemma 7.1, since $B_j \subseteq E$, for all $u' \in B_j, M_{u'x} = M_{u'y}$ for all $x, y \in S_i^l$. We already know that for all $u' \in B_j, M_{u'w} = a^{-1}$ for all $w \in D_i$, and $S_i^l = C_i \cup D_i$. Hence, for all $v \in C_i, M_{u'v} = a^{-1}$ too, and $M_{vu'} = a$.

Putting together all the partial results obtained, we get the assertion stated in the lemma. \blacksquare

Corollary 7.1 *Let $M = \text{Adj}(G)$. For each $i \in \{1, \dots, r\}$, it is satisfied that for all $u \in B_i, v \in C_i, w \in D_i, M_{uv} = M_{vu} = M_{uw} = M_{wu} = M_{vw} = M_{wv} = a$, where $a \in \{0, 3\}$.*

Proof: From Lemma 7.2, for the case $i = j$, we get that for all $u \in B_i, v \in C_i, w \in D_i, M_{uv} = M_{uw} = M_{vu} = M_{vw} = M_{wu} = M_{wv} = a$ and $M_{uv} = M_{uw} = M_{vu} = M_{vw} = M_{wu} = M_{wv} = a^{-1}$. Hence, it must hold that $a = a^{-1}$, so $a \in \{0, 3\}$. \blacksquare

Let us define two families of partitions of A_i for $i, j \in \{1, \dots, r\}$:

$$A_i^{c_j} = \{x \in A_i : \forall u \in B_i, v' \in C_j, M_{xv'} = M_{uv'}\}$$

$$A_i^{n_j} = \{x \in A_i : \forall u \in B_i, v' \in C_j, M_{xv'} \neq M_{uv'}\}$$

Note that, since the vertices of A_i are unable to distinguish among the vertices of C_j , then, if $M_{xv'} \neq M_{uv'}$ for some $u \in B_i$ or some $v' \in C_j$, then $M_{xv'} \neq M_{uv'}$ for all $u \in B_i$ and all $v' \in C_j$. Hence, each pair of sets $A_i^{c_j}$ and $A_i^{n_j}$ defines a partition of A_i . Note also that, since each vertex in A_i has the same type of adjacency with all the vertices in $B_i \cup C_i \cup D_i$ (from Lemma 7.1), then for all $x \in A_i^{c_j}, u \in B_i, v \in C_i, w \in D_i, u' \in B_j, v' \in C_j$, and $w' \in D_j, M_{xu'} = M_{xv'} = M_{xw'} = M_{uv'} = M_{uw'} = M_{vu'} = M_{vw'} = M_{wu'} = M_{wv'}$ (from Lemma 7.2).

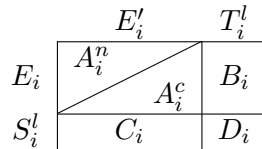


Figure 7.2: Partition of A_i into subsets A_i^c , and A_i^n .

Lemma 7.3 *For all $i \in \{1, \dots, r\}$, let $A_i^c = \bigcap_{j=1}^r A_i^{c_j}$, and let $A_i^n = \bigcup_{j=1}^r A_i^{n_j}$. Then, any isomorphism of G_E and $G_{E'}$ that maps G_{E_i} to $G_{E'_i}$, maps the vertices in A_i^n among themselves.*

Proof: From Observation 7.1, partition \mathcal{S}^k is equitable. Hence, for each $i, j \in \{1, \dots, r\}$, for all $u, v \in S_i^k, \text{ADeg}(u, S_j^k, G) = \text{ADeg}(v, S_j^k, G)$. Thus, for all $x \in A_i^{c_j}, y \in A_i^{n_j}, u \in B_i, v \in C_i, w \in D_i, \text{ADeg}(x, S_j^k, G) = \text{ADeg}(y, S_j^k, G) = \text{ADeg}(u, S_j^k, G) = \text{ADeg}(v, S_j^k, G) = \text{ADeg}(w, S_j^k, G)$.

Let us take any pair of values of i and j . From Lemma 7.2, all the vertices of B_i have the same type of adjacency with all the vertices of $S_j^l = C_j \cup D_j$. Assume this type of adjacency is a . From the definition of A_i^{cj} , all the vertices of A_i^{cj} have adjacency a with all the vertices of S_j^l . Hence, for $x \in A_i^{cj}$, $u \in B_i$, $ADeg(x, S_j^l, G) = ADeg(u, S_j^l, G)$. Since $ADeg(x, S_j^k, G) = ADeg(u, S_j^k, G)$ and $ADeg(x, S_j^l, G) = ADeg(u, S_j^l, G)$, then $ADeg(x, E_j, G) = ADeg(u, E_j, G)$ (note that $E_j = A_j^{ci} \cup A_j^{ni} \cup B_j$, $S_j^l = C_j \cup D_j$, and $S_j^k = E_j \cup S_j^l$).

However, from the definition of A_i^{nj} , for $y \in A_i^{nj}$, $ADeg(y, S_j^l, G) \neq ADeg(x, S_j^l, G)$. Hence, since $ADeg(y, S_j^k, G) = ADeg(x, S_j^k, G)$, $ADeg(y, E_j, G) \neq ADeg(x, E_j, G)$.

Since any isomorphism must match vertices with the same degree, every isomorphism of G_E and $G_{E'}$ that maps G_{E_i} to $G_{E'_i}$, maps the vertices in A_i^{nj} among themselves.

Applying this argument over all possible values of j , we get that any isomorphism of G_E and $G_{E'}$ that maps G_{E_i} to $G_{E'_i}$, maps the vertices in A_i^{nj} among themselves, for all $i \in \{1, \dots, r\}$. ■

Let us focus on any isomorphism π of G_E and $G_{E'}$ that maps G_{E_i} to $G_{E'_i}$ for all $i \in \{1, \dots, r\}$ (there is at least one from Observation 7.3).

Lemma 7.4 G_B is isomorphic to G_C , and there is an isomorphism of them that matches the vertices in B_i to those in C_i , for all $i \in \{1, \dots, r\}$.

Proof: Let us analyze the adjacencies between the vertices in A_i^c , B_i , C_i , A_j^c , B_j , and C_j for some values of i and j . From Corollary 7.1, for all $u \in B_i$, $v \in C_i$, $M_{uv} = M_{vu} = a$, where $a \in \{0, 3\}$. From the definition of A_i^c , for all $x \in A_i^c$, $M_{xu} = M_{xv} = M_{ux} = M_{vx} = M_{uv} = a$.

From Lemma 7.3, the vertices of A_i^n are mapped among themselves in any isomorphism π of G_E and $G_{E'}$ that maps G_{E_i} to $G_{E'_i}$. Hence, the vertices of $A_i^c \cup B_i$ must be mapped to the vertices of $A_i^c \cup C_i$. If $a = 0$, then A_i^c , B_i , and C_i are disconnected. Hence, G_{B_i} and G_{C_i} must be isomorphic. In the case $a = 3$, taking the inverses of the graphs leads to the same result.

From Lemma 7.2, for each $i, j \in \{1, \dots, r\}$, there is some $a \in \{0, \dots, 3\}$ such that for all $u \in B_i$, $v \in C_i$, $u' \in B_j$, $v' \in C_j$, $M_{uv'} = M_{v'u} = a$ and $M_{u'v} = M_{v'u} = a^{-1}$. From the definition of A_i^c , for all $x \in A_i^c$, for all $u \in B_i$, $v \in C_i$, $u' \in B_j$, $v' \in C_j$, $M_{xu'} = M_{xv'} = M_{uv'}$.

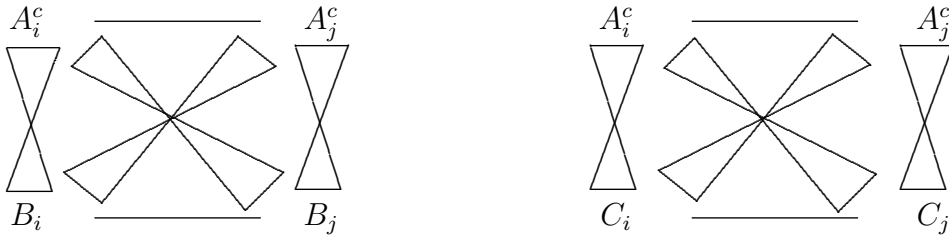


Figure 7.3: Adjacencies between E_i and E_j , and between E'_i and E'_j .

Putting all this together, we come to a picture of the adjacencies among A_i^c , B_i , C_i , A_j^c , B_j , and C_j as shown in Figure 7.3. The connections between the vertices of A_i^c and the vertices of B_i , and between the vertices of A_i^c and the vertices of C_i are all-to-all (all the same) of value 0 or 3. Similarly, the adjacencies between the vertices of A_j^c and the vertices of B_j , and the adjacencies between the vertices of A_j^c and the vertices of C_j are all the same, all-to-all 0 or 3 (not necessarily equal to those of A_i^c and B_i or C_i). The adjacencies between A_i^c and $B_j \cup C_j$

are all the same, all-to-all of any value in the set $\{0, \dots, 3\}$. This also applies to the adjacencies between A_j^c and $B_i \cup C_i$.

If $G_{B_i \cup B_j}$ is not isomorphic to $G_{C_i \cup C_j}$, the discrepancy must be in the adjacencies between vertices of B_i and B_j with respect to the adjacencies between vertices of C_i and C_j . In such a case, in the isomorphism π between $G_{E_i \cup E_j}$ and $G_{E'_i \cup E'_j}$ (recall that from Observation 7.3 there is an isomorphism of G_E and $G_{E'}$ that maps the vertices of E_i to the vertices in E'_i for all $i \in \{1, \dots, r\}$) some vertices of A_i^c should be mapped to vertices of C_i , and some of the vertices of B_i should be mapped to vertices of A_i^c . However, due to the adjacencies among A_i^c , B_i , C_i , A_j^c , B_j , and C_j , shown in Figure 7.3, that would imply that the adjacencies between the vertices of B_i and B_j had to match adjacencies between the vertices of A_i^c and A_j^c . But, in that case, the same adjacency pattern must exist between the vertices of C_i and C_j , to match the corresponding subgraph of $G_{E_i \cup E_j}$. Hence, the adjacencies between B_i and B_j could have been matched to the adjacencies between C_i and C_j .

Since this applies for all values of i and j , we conclude that G_B is isomorphic to G_C , and there is an isomorphism of them that matches the vertices in B_i to those in C_i , for all $i \in \{1, \dots, r\}$, completing the proof. \blacksquare

Lemma 7.5 G_{V^l} and G_{W^l} are isomorphic, and there is an isomorphism of them that maps the vertices in S_i^l to the vertices of T_i^l for all $i \in \{1, \dots, r\}$.

Proof: From Lemma 7.2, we know that for each $i, j \in \{1, \dots, r\}$, there is some $a \in \{0, \dots, 3\}$ 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 $w' \in D_j$, $M_{uw'} = M_{u'w} = M_{vv'} = M_{v'v} = M_{ww'} = M_{w'w} = a$ and $M_{u'v} = M_{u'w} = M_{v'u} = M_{v'w} = M_{w'u} = M_{w'v} = a^{-1}$. Note also that, from Corollary 7.1, for all $u \in B_i$, $v \in C_i$, $w \in D_i$, $M_{uw} = M_{vw} = M_{wu} = a$, where $a \in \{0, 3\}$. This adjacency pattern is graphically shown in Figure 7.4.

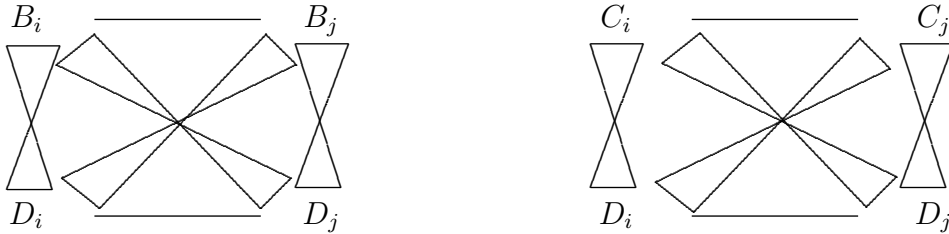


Figure 7.4: Adjacencies between S_i^l and S_j^l , and between T_i^l and T_j^l .

From Lemma 7.4, we know that G_B is isomorphic to G_C , and there is an isomorphism of them that matches the vertices in B_i to those in C_i , for all $i \in \{1, \dots, r\}$.

From the fact that G_D is isomorphic to itself, and the previous considerations on the adjacency pattern between the vertices in B_i , C_i , D_i , B_j , C_j , and D_j for all $i, j \in \{1, \dots, r\}$, shown in Figure 7.4, it is easy to see that the isomorphism of G_B and G_C obtained from Lemma 7.4, together with the trivial automorphism of G_D yields an isomorphism of G_{V^l} and G_{W^l} , what completes the proof. \blacksquare

We have shown that if two alternative sequences of partitions S^{k+1}, \dots, S^l and T^{k+1}, \dots, T^l lead to compatible partitions S^l and T^l , where all their cells are subcells of different cells of a previous common level k , then the remaining subgraphs are isomorphic, and the vertices in each cell of

one partition may be mapped to the vertices in its corresponding cell in the other partition by one such isomorphism. Thus, if during the search for a sequence of partitions compatible with the target, we have got an incompatibility at some point beyond level l , and we have to backtrack from one level l to another level k in which all the cells are different supersets of the cells in the current backtracking point, when trying a compatible path, we will get to the same dead-end. Hence, it is of no use to try another path from one such level k , and it will be necessary to backtrack to some point where at least two cells in the current backtracking point are subsets of the same cell in the previous backtracking point. This proves Theorem 7.1.

7.2 Algorithm *conauto-v1*

Algorithm *conauto-v1* is the result of applying Theorem 7.1 to algorithm *conauto-v0*. This new algorithm is only slightly different from the previous one. When backtracking is needed, the algorithm will try to backtrack directly to the highest possible level, i.e., the nearest upper level where at least two cells with links in the current level belong to the same cell.

This improvement has a limited field of application, since it can only speed up the cases of graphs built from components, either disconnected, fully connected, or even, in some special cases, partially connected. A family of graphs that will benefit from this new heuristic is the unions of strongly regular graphs, which are known to be very hard for nauty.

7.2.1 Main Algorithm

Algorithm 11 describes the behavior of *conauto-v1*. Like the previous algorithms, it receives two graphs G and H as parameters and returns TRUE if both graphs are isomorphic, and FALSE if they are not. The only difference with *conauto-v0* comes from the fact that Algorithm *Match3* returns a value that may be negative, in which case it has been impossible to find an isomorphism of graphs G and H , so FALSE is returned, or may be a positive value, in which case an isomorphism has been found and the algorithm returns TRUE. The rest of algorithms used are the same ones described in Section 6.2.2.

7.2.2 Search for a Sequence of Partitions Compatible with the Target

The new piece of Algorithm 11 is Algorithm *Match3*, described as Algorithm 12. Algorithm 12 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 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 $R^l = \text{VERTEX}$, 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

Algorithm 11 Test whether G and H are isomorphic (*conauto-v1*).

```

AreIsomorphic3( $G, H$ ) : 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    $\mathcal{D}_G \leftarrow \text{DegreePartition}(G)$ 
6    $\mathcal{D}_H \leftarrow \text{DegreePartition}(H)$ 
7   if  $\mathcal{D}_G$  and  $\mathcal{D}_H$  are not compatible under  $G$  and  $H$  respectively then
8     return FALSE
9   else
10     $\mathbf{Q}_G \leftarrow \text{GenerateSequenceOfPartitions}(G, \mathcal{D}_G)$ 
11     $\mathbf{Q}_H \leftarrow \text{GenerateSequenceOfPartitions}(H, \mathcal{D}_H)$ 
12     $\mathbf{E}_G \leftarrow \text{FindAutomorphisms}(G, \mathbf{Q}_G)$ 
13     $\mathbf{E}_H \leftarrow \text{FindAutomorphisms}(H, \mathbf{Q}_H)$ 
14    if  $\text{BacktrackAmount}(\text{SeqPart}(\mathbf{E}_G)) \leq \text{BacktrackAmount}(\text{SeqPart}(\mathbf{E}_H))$  then
15      return  $(0 \leq \text{Match3}(0, G, H, \text{SeqPart}(\mathbf{E}_G), \mathcal{D}_H, \text{Orbits}(\mathbf{E}_H)))$ 
16    else
17      return  $(0 \leq \text{Match3}(0, H, G, \text{SeqPart}(\mathbf{E}_H), \mathcal{D}_G, \text{Orbits}(\mathbf{E}_G)))$ 
18    end if
19  end if
20 end if

```

a previous level that satisfies the condition of Theorem 7.1 (lines 45 and 46). Note that, if the partition at the current level is equitable, then clearly Theorem 7.1 applies, and if the partition at the current level is not equitable, then the value $\max k \in Y$ would be the nearest backtracking point, yielding the same result of *Match2*.

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 7.1, 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. Lines 45 and 46 decide to which level it is necessary to backtrack from this level, just as if the incompatibility had been found at this level.

- If $R^l = \text{SET}$, then a set refinement is performed, and the algorithm behaves the same as in the previous case, where $R^l = \text{VERTEX}$.
- In the case where $R^l = \text{BACKTRACK}$, 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. 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

Algorithm 12 Find a sequence of partitions compatible with the target (*conauto-v1*).

```

Match3( $l, G, H, Q_G, \mathcal{T}, O_H$ ) : boolean
1 -- let  $Q_G = (S, R, P)$ , let  $S = (S^0, \dots, S^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $P = (P^0, \dots, P^{t-1})$ 
2 -- let  $S^l = (S^l_1, \dots, S^l_{r_l})$ ,  $V^l = \bigcup_{j=1}^{r_l} S^l_j$ , for all  $l \in \{0, \dots, t\}$ 
3 -- let  $H = (V_H, R_H)$ , let  $\mathcal{T} = (T_1, \dots, T_{r_1})$ ,  $W = \bigcup_{j=1}^{r_1} T_j$ 
4 if  $l = t$  then
5   if  $\forall x, y \in \{1, \dots, r_l\}$ ,  $ADeg(S^t_x, S^t_y, G) = ADeg(T_x, T_y, H)$  then
6     return  $t$ 
7   end if
8 else
9    $X \leftarrow T_{P^l}$ 
10  if  $R^l = \text{BACKTRACK}$  then
11     $orbitsApplicable \leftarrow \forall v \in V_H \setminus W, |Orb(v, O_H)| = 1$ 
12    for each  $v \in X$  do
13       $Valid(Orb(v, O_H)) \leftarrow \text{TRUE}$ 
14    end for
15    repeat
16       $v \leftarrow$  any vertex in  $X$ 
17       $X \leftarrow X \setminus \{v\}$ 
18      if  $\neg orbitsApplicable \vee Valid(Orb(v, O_H))$  then
19         $\mathcal{T}' \leftarrow VertexRefinement(\mathcal{T}, v, H_W)$ 
20        -- let  $\mathcal{T}' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
21        if  $S^{l+1}$  and  $\mathcal{T}'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
22           $m \leftarrow Match3(l+1, G, H, Q_G, \mathcal{T}', O_H)$ 
23          if  $l \neq m$  then
24            return  $m$ 
25          end if
26        end if
27         $Valid(Orb(v, O_H)) \leftarrow \text{FALSE}$ 
28      end if
29    until  $X = \emptyset$ 
30  else
31    if  $R^l = \text{VERTEX}$  then
32       $v \leftarrow$  any vertex in  $X$ 
33       $\mathcal{T}' \leftarrow VertexRefinement(\mathcal{T}, v, H_W)$ 
34    else (i.e.  $R^l = \text{SET}$ )
35       $\mathcal{T}' \leftarrow SetRefinement(\mathcal{T}, X, H_W)$ 
36    end if
37    -- let  $\mathcal{T}' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
38    if  $S^{l+1}$  and  $\mathcal{T}'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
39       $m \leftarrow Match3(l+1, G, H, Q_G, \mathcal{T}', O_H)$ 
40      if  $l \neq m$  then
41        return  $m$ 
42      end if
43    end if
44  end if
45  -- let  $X = \{S^l_i : HasLinks(S^l_i, V^l, G)\}$ ,  $Y = \{-1\} \cup \{k < l : \exists S^l_i, S^l_j \in X : \exists S^k_x \in \mathcal{S}^k : S^l_i, S^l_j \subseteq S^k_x\}$ 
46  return  $\max k \in Y$ 

```

is returned. This is done in lines 23 and 24.

If $m < l$, then an incompatibility was found in a subsequent level, and applying the result of Theorem 7.1, it is not necessary to try any other possibility at this level, so m is directly

returned. This is accomplished by lines 23 and 24.

When all the possibilities have been tried unsuccessfully, backtracking is needed. Then, applying Theorem 7.1 in lines 44 and 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.

7.3 Performance Evaluation

In this section we will show how the presented improvement benefits the families of graphs that are built making unions of simple regular graphs. These are the unions of strongly regular graphs, and the unions of tripartite graphs. Since this improvement has very little computation cost, and it is not useful for the other families of graphs, we will not show those results, since they are almost exactly the same as the ones presented for the previous version of the algorithm.

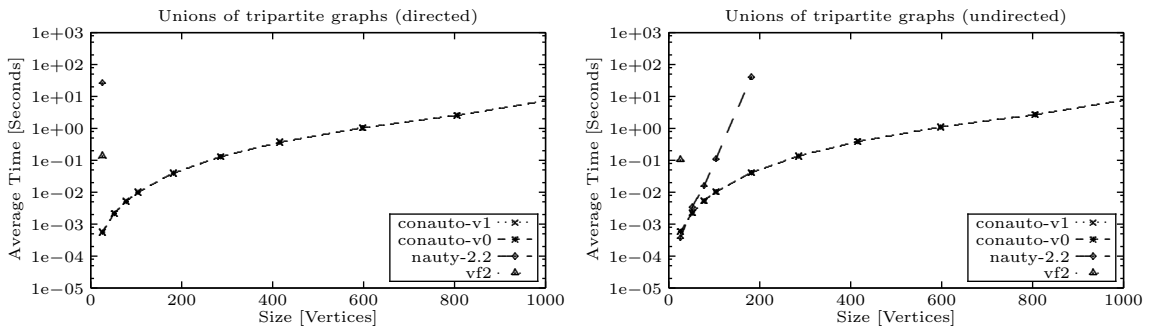


Figure 7.5: Performance of conauto-v1 with isomorphic unions of tripartite graphs.

Figure 7.5 compares the performance of conauto-v1 with conauto-v0, nauty, and vf2, for isomorphic unions of tripartite graphs. In this case, there is no improvement in the performance since the only backtracking that may be necessary in this case is that to match independent components, and that is not avoidable with the new technique. Hence, the performance of conauto-v1 is exactly the same as the one of conauto-v0. However, this is not a bad result, since the performance of conauto-v0 was already good.

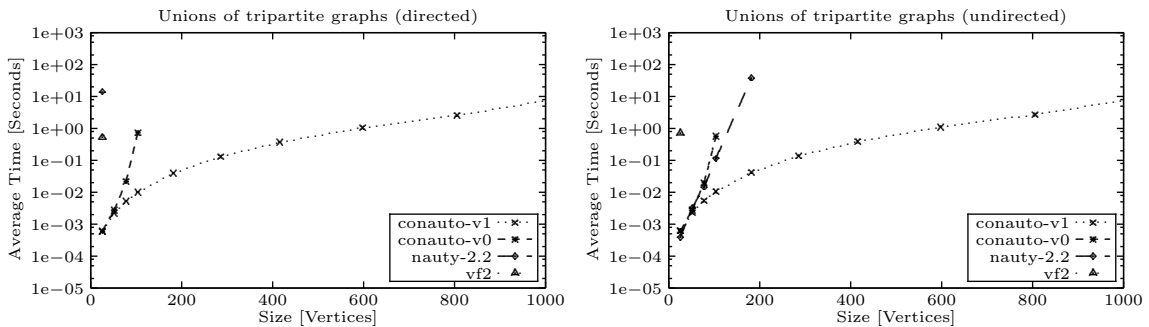


Figure 7.6: Performance of conauto-v1 with non-isomorphic unions of tripartite graphs.

For the case of non-isomorphic unions of tripartite graphs, algorithm conauto-v1 exhibits a much

better behavior than its predecessors. Figure 7.6 shows that the time required by conauto-v1 to process non-isomorphic graphs resembles a polynomial time complexity, what meets our expectations. Comparing the results of Figures 7.5 and 7.6, it can be seen that the time required for positive and negative cases has become almost exactly the same.

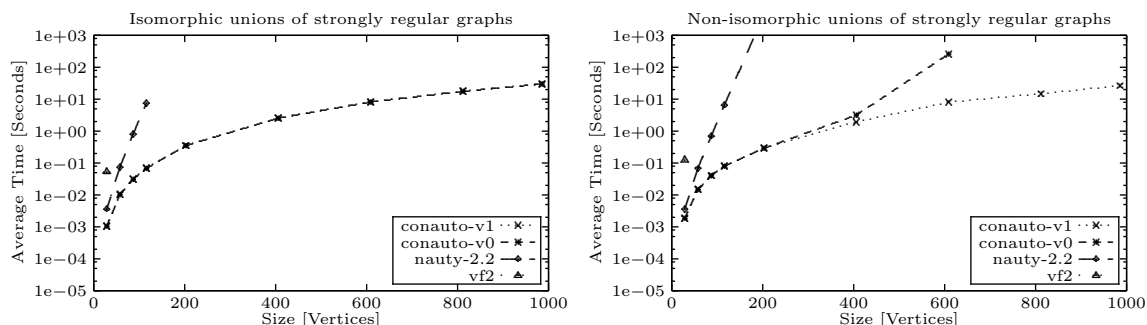


Figure 7.7: Performance of conauto-v1 with unions of strongly regular graphs.

Algorithm conauto-v1 exhibits, with unions of strongly regular graphs, a uniform behavior for the positive and negative cases, as can be seen in Figure 7.7. Again its time complexity looks polynomial, unlike nauty and vf2, for which this family of graphs is too hard.

Note also that the theoretical result on which conauto-v1 relies does not only apply to disjoint unions of graphs, and fully connected unions of graphs. The result is more general, and partial unions may also benefit from this result.

Chapter 8

Further Automorphism Detection

Algorithm `conauto-v1` has shown its effectiveness on unions of graphs. However, the improvement introduced in algorithm `conauto-v1` has no effect on other families of graphs. Hence, for instance, the Latin square graphs and the point-line graphs of Desarguesian projective planes remain as hard families for our algorithms. In this chapter we will focus on the former family of graphs. First of all, note that Latin square graphs are vertex-transitive. Algorithms `conauto-v0` and `conauto-v1` are unable to discover that, though. In fact, what happens is that they leave two backtracking points at the beginning of any sequence of partitions for a Latin square graph. Performing a limited search for automorphisms (like the one done in the previous versions of `conauto`) is very effective in the case of Miyazaki's F\"urer gadgets but, in this case, it limits the ability of those algorithms to find automorphisms that could greatly improve the performance.

In our new algorithm `conauto-v2`, we will add the possibility of performing a deeper search for automorphisms in some cases where it may help. This has to be done preventing at the same time the algorithm from doing that in the cases where it might be inadequate, like the case of Miyazaki's F\"urer gadgets, or union graphs. Since computing a complete sequence of partitions is quite expensive (though polynomial in time), it would be desirable to be able to establish vertex transitivity generating as few sequences of partitions, and subsequent searches for automorphisms as possible.

Something that we have observed for our family of Latin square graphs is that the sequences of partitions they yield have two levels of backtracking (the first two in the sequence). The other backtracking points in the sequences of partitions are eliminated during the search for automorphisms. Hence, we will apply this new technique only when the two first levels of the sequence of partitions are the only backtracking points in the sequence, after the basic search for automorphisms has been performed. This may be too restrictive, but we can not generalize the field of application of this technique yet.

Besides, since performing this additional search for automorphisms requires significant processing time, we will perform it only on one of the graphs being tested, the one in which the search for a sequence of partitions compatible with the target will be performed. To manage several orbit partitions and operate with them, some new notation needs to be introduced.

8.1 Algorithm *conauto-v2*

In the algorithms presented, only one orbit partition is used for each graph. However, in *conauto-v2*, more than one sequence of partitions may be generated if the algorithm considers it convenient to do so. Hence, it is necessary to represent and manage such collections of orbit partitions. One way to do so is to manage all the partitions for a graph as a sequence of orbit partitions. Note that, although for simplicity we refer to them as orbit partitions, we mean semiorbit partitions (see Definition 6.2), since they are not (necessarily) true orbit partitions.

Definition 8.1 *A sequence of semiorbit partitions SO is an ordered set of semiorbit partitions. \emptyset will denote the empty sequence of semiorbit partitions. We will use the operator $+$ to append a new semiorbit partition to a sequence of semiorbit partitions. Let O be a semiorbit partition. If $SO = \emptyset$, then $SO + O = (O)$. If $SO = (O_1, \dots, O_n)$, then $SO + O = (O_1, \dots, O_n, O)$.*

8.1.1 Main Algorithm

Algorithm 13 (*conauto-v2*) differs from the previously presented algorithms in that, after obtaining the extended sequences of partitions, with the basic orbits, for both graphs, it tests if searching for more automorphisms might help pruning the subsequent search for the compatible sequence of partitions (in particular, if the first two levels in the sequence of partitions $SeqPart(E_G)$ are the only backtrack levels). In such a case, it calls Algorithm 15 to search for more automorphisms, obtaining a sequence of orbit partitions for graph G , and then calls Algorithm 17 passing to it, instead of the basic orbits obtained in the first search for automorphisms, the sequence of orbit partitions computed by Algorithm 15. If it is not worth looking for more automorphisms, it calls Algorithm 17 passing to it a sequence of orbit partitions that only contains the basic orbits. In this case, the behavior of Algorithm 17 will be exactly the same as that of Algorithm 12.

Like in the case of *conauto-v1*, if the algorithm that performs the search for the compatible sequence of partitions, in this case *Match4*, returns -1 , the graphs are not isomorphic, and otherwise they may be.

8.1.2 Generation of a Sequence of Partitions

Algorithm 14 is used to generate a sequence of partitions. In addition to the parameters of Algorithm 2, it has a new parameter F used to generate sequences of partitions with different initial pivot vertices in the case it is used to find more automorphisms with Algorithm 15. This parameter indicates which vertices must be excluded from the pivot set selected by *IndexBestPivot* in the first level of backtracking. When $F = \emptyset$, this algorithm behaves exactly as Algorithm 2. When $F \neq \emptyset$, in the first level marked as BACKTRACK, only the vertices of the pivot cell that are not in F are considered as possible pivot vertices for the vertex refinement (line 27). Note that after its first use, F takes value \emptyset , so that it is not used again.

8.1.3 Extended Look for Automorphisms

Algorithm 15 (*FindMoreAutomorphisms*) receives the graph to be searched for automorphisms G , its degree partition \mathcal{D} , and its basic orbits O which were computed by Algorithm 6, and

Algorithm 13 Test whether G and H are isomorphic (*conauto-v2*).

```

AreIsomorphic4( $G, H$ ) : 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    $\mathcal{D}_G \leftarrow \text{DegreePartition}(G)$ 
6    $\mathcal{D}_H \leftarrow \text{DegreePartition}(H)$ 
7   if  $\mathcal{D}_G$  and  $\mathcal{D}_H$  are not compatible under  $G$  and  $H$  respectively then
8     return FALSE
9   else
10     $\mathcal{Q}_G \leftarrow \text{GenerateSequenceOfPartitions2}(G, \mathcal{D}_G, \emptyset)$ 
11     $\mathcal{Q}_H \leftarrow \text{GenerateSequenceOfPartitions2}(H, \mathcal{D}_H, \emptyset)$ 
12     $\mathbf{E}_G \leftarrow \text{FindAutomorphisms}(G, \mathcal{Q}_G)$ 
13     $\mathbf{E}_H \leftarrow \text{FindAutomorphisms}(H, \mathcal{Q}_H)$ 
14    -- let  $\text{SeqPart}(\mathbf{E}_G) = (\mathbf{S}, \mathbf{R}, \mathbf{P})$ , and let  $\mathbf{P} = (P^0, \dots, P^{t-1})$ 
15    if  $(\forall i \in \{0, 1\}, R^i = \text{BACKTRACK}) \wedge (\forall i \in \{2, \dots, t-1\}, R^i \neq \text{BACKTRACK})$  then
16      -- let  $\mathbf{S} = (\mathcal{S}^0, \dots, \mathcal{S}^t)$ 
17      -- let  $p$  be the pivot vertex used to generate partition  $\mathcal{S}^1$  from partition  $\mathcal{S}^0$ 
18       $\mathbf{SO} \leftarrow \text{FindMoreAutomorphisms}(G, \mathcal{D}_G, \text{Orbits}(\mathbf{E}_G), \{p\})$ 
19    else
20       $\mathbf{SO} \leftarrow (\text{Orbits}(\mathbf{E}_G))$ 
21    end if
22    return  $0 \leq \text{Match4}(0, H, G, \text{SeqPart}(\mathbf{E}_H), \mathcal{D}_G, \mathbf{SO})$ 
23  end if
24 end if

```

returns a sequence of orbit partitions, built from the basic orbits and the new orbit partitions found. The last orbit partition in the sequence will be what we will call the global orbit partition; that that comes from merging all the other orbit partitions generated. Since we can only use orbit partitions when all the vertices fixed during the search for a compatible sequence of partitions belong to a singleton orbit, it is important to keep the orbit partitions generated with some vertex fixed (that belongs to a singleton orbit). This way, they will be applicable when that vertex is fixed during the search.

The global orbit partition will be stored, temporarily, in \mathcal{O}' before it is added to the sequence returned by *FindMoreAutomorphisms*.

8.1.4 Computing Orbits Applicable

A feature that is essential to *conauto-v2* is that it operates with orbit partitions. It not only uses vertex equivalences explicitly found during the polynomial-time search for automorphisms, but it is also able to compute new ones applying automorphism composition. Given a sequence of orbit partitions, Algorithm 16 computes the vertex equivalences suitable for a certain point of backtracking, according to the information available on automorphisms.

If we have computed an orbit partition in which there are singleton orbits, that means that there are automorphisms that fix those vertices, and permute, in some way, the vertices in the orbits with more than one vertex. However, not all possible permutations of these latter vertices will yield a valid automorphism. The existence of an automorphism that permutes two vertices does not imply that at any point in the search for a compatible sequence of partitions,

Algorithm 14 Generate a sequence of partitions for a graph G (*conauto-v2*).

GenerateSequenceOfPartitions2(G, \mathcal{D}, F) : sequence of partitions

- 1 -- let $G = (V, R)$
- 2 -- for all $l > 0$, if \mathcal{S}^l is defined, let $\mathcal{S}^l = (S_1^l, \dots, S_{r_l}^l)$, $V^l = \bigcup_{j=1}^{r_l} S_j^l$
- 3 $\mathcal{S}^0 \leftarrow \mathcal{D}$
- 4 **for each** $S_x^0 \in \mathcal{S}^0$ **do**
- 5 $Valid(S_x^0) \leftarrow (|\mathcal{S}^0| > 1) \wedge HasLinks(S_x^0, V^0, G)$
- 6 **end for**
- 7 $l \leftarrow 0$
- 8 **while** $\exists S_x^l \in \mathcal{S}^l : (|S_x^l| > 1) \wedge HasLinks(S_x^l, V^l, G)$ **do**
- 9 $P^l \leftarrow IndexBestPivot(\mathcal{S}^l, G)$
- 10 **if** $|S_{P^l}^l| = 1$ **then**
- 11 $R^l \leftarrow VERTEX$
- 12 $v \leftarrow$ the only vertex in $S_{P^l}^l$
- 13 $\mathcal{S}^{l+1} \leftarrow VertexRefinement(\mathcal{S}^l, v, G_{V^l})$
- 14 **else**
- 15 $success \leftarrow FALSE$
- 16 **while** $Valid(S_{P^l}^l) \wedge \neg success$ **do**
- 17 $Valid(S_{P^l}^l) \leftarrow FALSE$
- 18 $R^l \leftarrow SET$
- 19 $\mathcal{S}^{l+1} \leftarrow SetRefinement(\mathcal{S}^l, S_{P^l}^l, G_{V^l})$
- 20 $success \leftarrow \exists S_x^{l+1}, S_{x+1}^{l+1} : S_x^{l+1}, S_{x+1}^{l+1} \subset S_y^l$ for some $S_y^l \in \mathcal{S}^l$
- 21 **if** $\neg success$ **then**
- 22 $P^l \leftarrow IndexBestPivot(\mathcal{S}^l, G)$
- 23 **end if**
- 24 **end while**
- 25 **if** $\neg success$ **then**
- 26 $R^l \leftarrow BACKTRACK$
- 27 $v \leftarrow$ any vertex in $S_{P^l}^l \setminus F$
- 28 $F \leftarrow \emptyset$
- 29 $\mathcal{S}^{l+1} \leftarrow VertexRefinement(\mathcal{S}^l, v, G_{V^l})$
- 30 **end if**
- 31 **end if**
- 32 $l \leftarrow l + 1$
- 33 **for each** $S_x^l \in \mathcal{S}^l$ **do**
- 34 -- let $S_x^l \subseteq S_y^{l-1}$, $S_y^{l-1} \in \mathcal{S}^{l-1}$
- 35 $Valid(S_x^l) \leftarrow HasLinks(S_x^l, V^l, G) \wedge (Valid(S_y^{l-1}) \vee (|S_x^l| < |S_y^{l-1}|))$
- 36 **end for**
- 37 **end while**
- 38 $t \leftarrow l$
- 39 $\mathcal{S} \leftarrow (\mathcal{S}^0, \dots, \mathcal{S}^t)$; $\mathcal{R} \leftarrow (R^0, \dots, R^{t-1})$; $\mathcal{P} \leftarrow (P^0, \dots, P^{t-1})$
- 40 **return** ($\mathcal{S}, \mathcal{R}, \mathcal{P}$)

if they are both in the pivot set of a backtracking point, they may be considered equivalent. That only applies when there is an automorphism that permutes them, and fixes all the vertices discarded during the generation of the sequence of partitions, in the previous levels. Recall that the automorphism induced by two compatible sequences of partitions is determined by the order it induces on the vertices of the graphs, and all the vertices previously discarded hold the same place in both orders.

Algorithm *FindMoreAutomorphisms* may have discovered that two given vertices are equivalent, and put them in the same orbit of the global orbit partition, but this equivalence might come from an automorphism that permutes all the other vertices in the graph. During the search for

Algorithm 15 Look for more automorphisms.

FindMoreAutomorphisms($G, \mathcal{D}, \mathcal{O}, F$) : sequence of orbit partitions

- 1 -- let $G = (V, R)$
- 2 $\mathcal{O}' \leftarrow \mathcal{O}$
- 3 $\text{SO} \leftarrow (\mathcal{O})$
- 4 $W \leftarrow F$
- 5 **while** $|\mathcal{O}'| > 1 \wedge |W| < |V|$ **do**
- 6 $Q \leftarrow \text{GenerateSequenceOfPartitions2}(G, \mathcal{D}_G, W)$
- 7 -- let $Q = (S, R, P)$, let $S = (S^0, \dots, S^t)$
- 8 -- let p be the pivot vertex used to generate partition S^1 from partition S^0
- 9 $W \leftarrow W \cup \{p\}$
- 10 $E \leftarrow \text{FindAutomorphisms}(G, Q)$
- 11 $\text{SO} \leftarrow \text{SO} + \text{Orbits}(E)$
- 12 **for each** $u, v \in V : \text{Orb}(u, \text{Orbits}(E)) = \text{Orb}(v, \text{Orbits}(E))$ **do**
- 13 $\mathcal{O}' \leftarrow \text{merge}(\mathcal{O}', \text{Orb}(u, \mathcal{O}'), \text{Orb}(v, \mathcal{O}'))$
- 14 **end for**
- 15 $W' \leftarrow \{v \in V : \exists w \in W : \text{Orb}(v, \mathcal{O}') = \text{Orb}(w, \mathcal{O}')\}$
- 16 $W \leftarrow W'$
- 17 **end while**
- 18 **return** $\text{SO} + \mathcal{O}'$

Algorithm 16 Compute the orbit partition applicable at this point.

OrbitsApplicable(X, SO) : orbit partition

- 1 -- let $\text{SO} = (\mathcal{O}_1, \dots, \mathcal{O}_n)$
- 2 -- let $V = \bigcup_{\mathcal{O}_i \in \text{SO}} \mathcal{O}_i$
- 3 **if** $X = \emptyset$ **then**
- 4 **return** \mathcal{O}_n
- 5 **else**
- 6 $\mathcal{O} \leftarrow \{\{v\} : v \in V\}$
- 7 **for each** $i \in \{1, \dots, n-1\}$ **do**
- 8 **if** $\forall x \in X, |\text{Orb}(x, \mathcal{O}_i)| = 1$ **then**
- 9 **for each** $u, v \in V : \text{Orb}(u, \mathcal{O}_i) = \text{Orb}(v, \mathcal{O}_i)$ **do**
- 10 $\mathcal{O} \leftarrow \text{merge}(\mathcal{O}, \text{Orb}(u, \mathcal{O}), \text{Orb}(v, \mathcal{O}))$
- 11 **end for**
- 12 **end if**
- 13 **end for**
- 14 **return** \mathcal{O}
- 15 **end if**

a compatible sequence of partitions, we make use of vertex equivalences among the vertices of both graphs.

Consider the first graph. If all the vertices in a potential backtracking point yield compatible sequences of partitions, then that backtracking point is eliminated because, if the other graph is isomorphic to this one, then there is a sequence of partitions compatible with that for the first graph, in which the vertices in the corresponding pivot cell must be equivalent. This comes from the fact that isomorphisms preserve automorphisms, and was discussed in Section 6.1. This way, even if we did not discover some vertex equivalence for the second graph, we know that it must hold if both graphs are isomorphic, and make use of that fact to prune the search.

When it has not been possible to eliminate a backtracking point, that does not mean that there is no automorphism that permutes some pair of vertices in the pivot set and fixes all the vertices

Algorithm 17 Find a sequence of partitions compatible with the target (*conauto-v2*).

```

Match4( $l, G, H, Q_G, \mathcal{T}, \text{SO}$ ) : boolean
1 -- let  $Q_G = (S, R, P)$ , let  $S = (S^0, \dots, S^t)$ ,  $R = (R^0, \dots, R^{t-1})$ ,  $P = (P^0, \dots, P^{t-1})$ 
2 -- let  $S^l = (S_1^l, \dots, S_{r_l}^l)$ ,  $V^l = \bigcup_{j=1}^{r_l} S_j^l$ , for all  $l \in \{0, \dots, t\}$ 
3 -- let  $H = (V_H, R_H)$ , let  $\mathcal{T} = (T_1, \dots, T_{r_t})$ ,  $W = \bigcup_{j=1}^{r_t} T_j$ 
4 if  $l = t$  then
5   if  $\forall x, y \in \{1, \dots, r_t\}$ ,  $ADeg(S_x^t, S_y^t, G) = ADeg(T_x, T_y, H)$  then
6     return  $t$ 
7   end if
8 else
9    $X \leftarrow T_{P^l}$ 
10  if  $R^l = \text{BACKTRACK}$  then
11     $O \leftarrow \text{OrbitsApplicable}(V_H \setminus W, \text{SO})$ 
12    for each  $v \in X$  do
13       $Valid(\text{Orb}(v, O)) \leftarrow \text{TRUE}$ 
14    end for
15    repeat
16       $v \leftarrow$  any vertex in  $X$ 
17       $X \leftarrow X \setminus \{v\}$ 
18      if  $Valid(\text{Orb}(v, O))$  then
19         $\mathcal{T}' \leftarrow \text{VertexRefinement}(\mathcal{T}, v, H_W)$ 
20        -- let  $\mathcal{T}' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
21        if  $S^{l+1}$  and  $\mathcal{T}'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
22           $m \leftarrow \text{Match4}(l+1, G, H, Q_G, \mathcal{T}', \text{SO})$ 
23          if  $l \neq m$  then
24            return  $m$ 
25          end if
26        end if
27         $Valid(\text{Orb}(v, O)) \leftarrow \text{FALSE}$ 
28      end if
29    until  $X = \emptyset$ 
30  else
31    if  $R^l = \text{VERTEX}$  then
32       $v \leftarrow$  any vertex in  $X$ 
33       $\mathcal{T}' \leftarrow \text{VertexRefinement}(\mathcal{T}, v, H_W)$ 
34    else (i.e.  $R^l = \text{SET}$ )
35       $\mathcal{T}' \leftarrow \text{SetRefinement}(\mathcal{T}, X, H_W)$ 
36    end if
37    -- let  $\mathcal{T}' = (T'_1, \dots, T'_r)$ ,  $W' = \bigcup_{j=1}^r T'_j$ 
38    if  $S^{l+1}$  and  $\mathcal{T}'$  are compatible under  $G_{V^{l+1}}$  and  $H_{W'}$  respectively then
39       $m \leftarrow \text{Match4}(l+1, G, H, Q_G, \mathcal{T}', \text{SO})$ 
40      if  $l \neq m$  then
41        return  $m$ 
42      end if
43    end if
44  end if
45  -- let  $X = \{S_i^l : \text{HasLinks}(S_i^l, V^l, G)\}$ ,  $Y = \{-1\} \cup \{k < l : \exists S_i^l, S_j^l \in X : \exists S_x^k \in \mathcal{S}^k : S_i^l, S_j^l \subseteq S_x^k\}$ 
46  return  $\max k \in Y$ 

```

discarded in the preceding levels. However, in this case, the equivalences discovered for the first graph are not transferable to the second graph. At this point, the automorphisms of the second graph are the only ones that may help. If we compute the whole automorphism group, we would be able to determine, for each pair of vertices, if they are equivalent at some point. However,

we have a limited knowledge of automorphisms. We only have a few (semi)orbit partitions, and the best we can do is trying to combine that partial knowledge to do our best. That is what Algorithm 16 tries to do. It combines all the available orbit partitions that fix all the vertices previously discarded. This is clearly not optimal, but it is the best we can do with our limited knowledge of automorphisms.

Algorithm 16 receives the set of vertices previously discarded, and operates in the following way: it starts from the trivial orbit partition where each vertex belongs to a different orbit. Then, it takes each orbit partition that fixes all those vertices, and returns the orbit partition obtained after applying all the vertex equivalences established by those orbit partitions. This process applies automorphism composition. Since, for now, we are applying this technique only in the case that there are only two levels of backtracking (the first two levels in the sequence), this algorithm will have a limited utility. However, it may be used in the future if more powerful automorphism management is added to the algorithm.

8.1.5 Finding a Sequence of Partitions Compatible with the Target

Algorithm *Match4* tries to find a sequence of partitions for graph H , that is compatible with the target Q_G . It uses the sequence of orbit partitions SO to prune the search. This algorithm differs from *Match3* in the way it handles the levels of backtracking (lines 11 and 18). The rest of the algorithm is identical to *Match3*.

When $R^l = \text{BACKTRACK}$, *Match4* uses Algorithm 16 (line 11) to compute the orbit partition that is applicable at level l . This orbit partition is computed from SO and the vertices that have already been fixed in the previous levels. Since the sequence of orbit partitions only has more than one orbit partition when there are only 2 levels of backtracking, then at most one vertex is fixed when the number of orbit partitions in SO is more than one.

Then, like *Match3*, it marks all the orbits of the vertices in the pivot set as valid, and tries them until it finds a sequence of partitions compatible with the target, or it has tested (or discarded) all possibilities. Each time a vertex is discarded, its orbit is marked as not valid, so no more vertices in its orbit are tested (since they would never yield a compatible sequence of partitions).

8.2 Performance Evaluation

Most of the results obtained for conauto-v2 are almost the same ones obtained for conauto-v0 and conauto-v1. Nevertheless, they show that the complexity introduced in the more sophisticated versions has, in general, low impact on the performance for the families of graphs that are not the target of the added functionalities.

The results obtained for conauto-v2 are very similar to those obtained for conauto-v0 for randomly connected graphs (see Figure 8.1, and compare it with Figure 6.5), and for 2D-meshes (see Figure 8.2). Algorithm conauto-v2 is never the worst for these families of graphs, and, in the case of undirected 2D-meshes, it is the best for the graphs of the biggest size.

With Miyazaki's Furer gadgets, conauto-v2 behaves uniformly for both positive and negative tests (see Figures 8.3 and 8.4), and finds the undirected version harder than the directed one, while nauty has better results for undirected graphs. This is a known problem of nauty. In

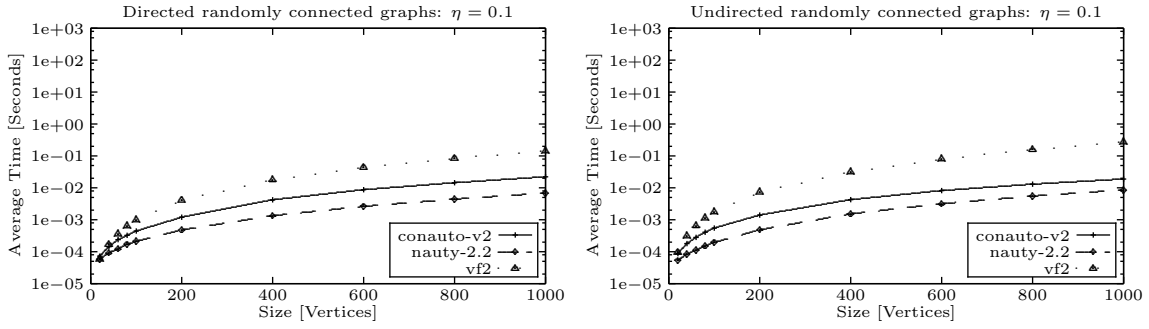


Figure 8.1: Performance of conauto-v2 with isomorphic randomly connected graphs.

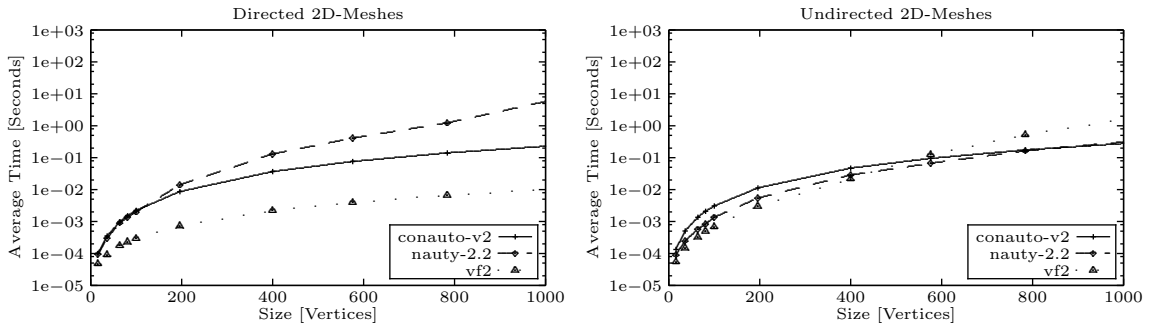


Figure 8.2: Performance of conauto-v2 with isomorphic 2D-meshes.

fact, nauty is unable to process, within our time-limit, the directed graphs of this family with 40 vertices.

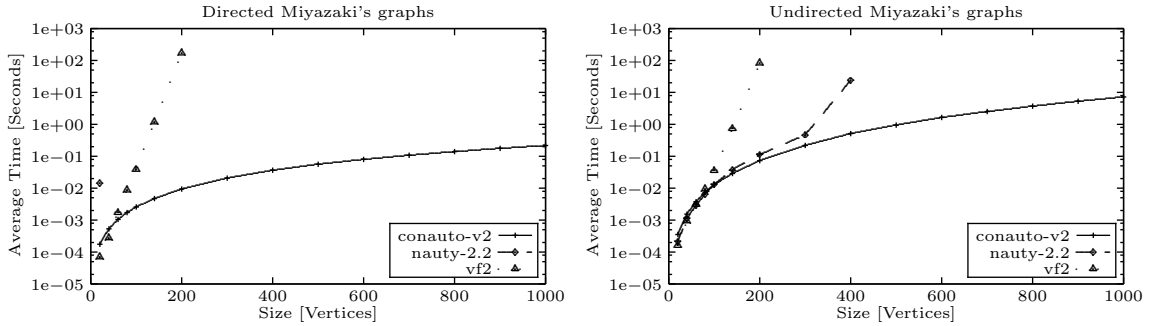


Figure 8.3: Performance of conauto-v2 with isomorphic Miyazaki's Fürer gadgets.

The results for most of the sub-families of strongly regular graphs (Paley, triangular, and lattice graphs), shown in Figures 8.5 and 8.6 are like the ones for conauto-v0. For these families of graphs, conauto-v2 and nauty have similar results, while vf2 finds triangular and lattice graphs much easier than Paley graphs.

Latin square graphs were the aim of the improvement introduced in conauto-v2. The low performance of conauto-v0 and conauto-v1 with respect to nauty was due to the fact that the former algorithms do not manage automorphisms in a sophisticated way, but quite simply. These algorithms are unable to eliminate all the backtracking points, and the equivalences found among vertices are not always useful. However, algorithm conauto-v2 benefits from its further search for automorphisms. Thus, it is able to eliminate the backtracking point at level 0, discovering

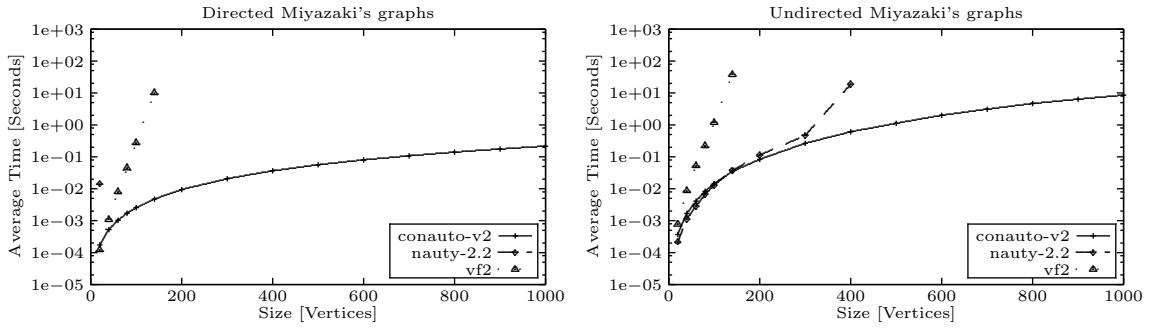


Figure 8.4: Performance of conauto-v2 with non-isomorphic Miyazaki's Fürer gadgets.

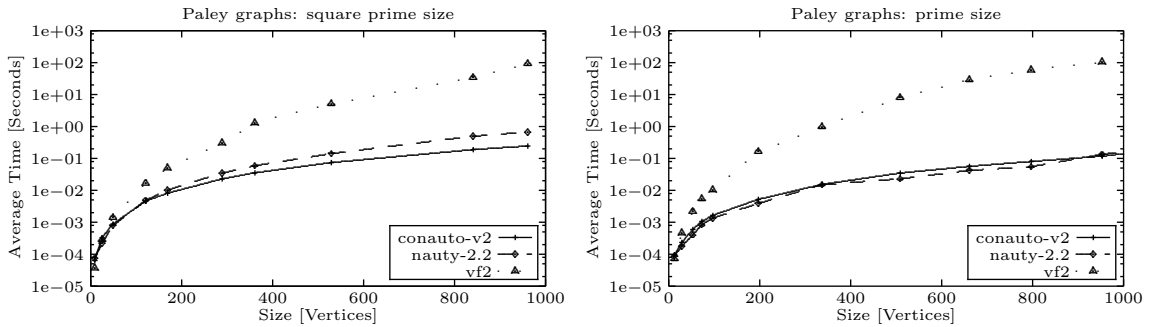


Figure 8.5: Performance of conauto-v2 with Paley graphs.

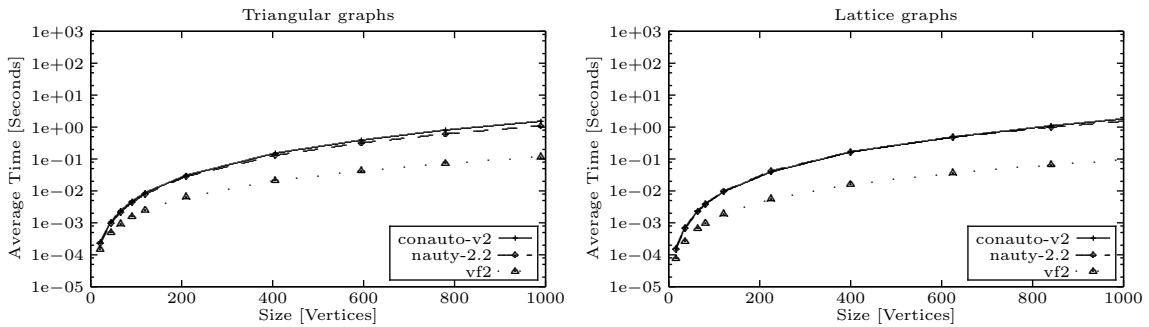


Figure 8.6: Performance of conauto-v2 with triangular and lattice graphs.

the vertex transitivity of Latin square graphs, improving the performance, as it can be seen in Figure 8.7. Although conauto-v2 is still slower than nauty, there is less than one order of magnitude of difference, what is an improvement of three orders of magnitude compared with sinauto. Since the performance for positive and negative tests is the same, it is not likely to be easily improved without a thorough revision of the algorithm.

The results for the family of tripartite graphs are shown in Figures 8.8 and 8.9. Algorithm conauto-v2 runs a bit faster than its previous version conauto-v1 (conauto-v1 generates sequences of partitions for both graphs before calling algorithm *Match4* while conauto-v2 only generates one of them), and has very similar results for positive and negative tests, and also very similar for the directed and the undirected versions of the graphs. It is important to note that nauty is unable to give an answer within the time-limit of 10000 seconds for directed graphs with 52 vertices, and for undirected graphs with more than 200 vertices.

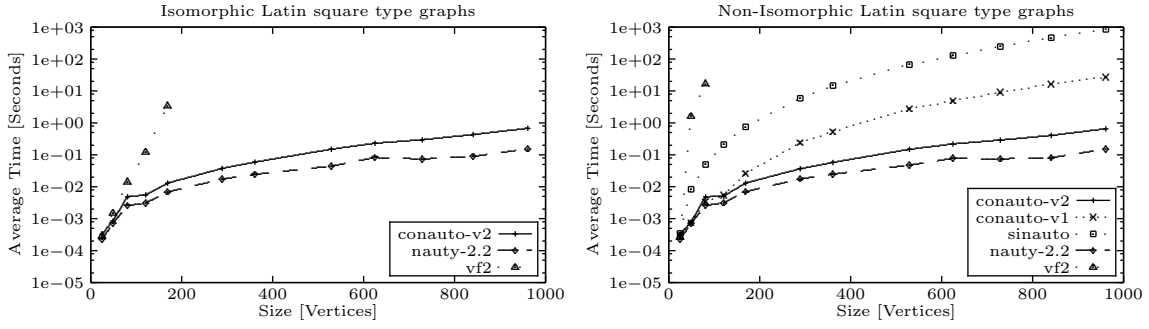


Figure 8.7: Performance of conauto-v2 with Latin square graphs.

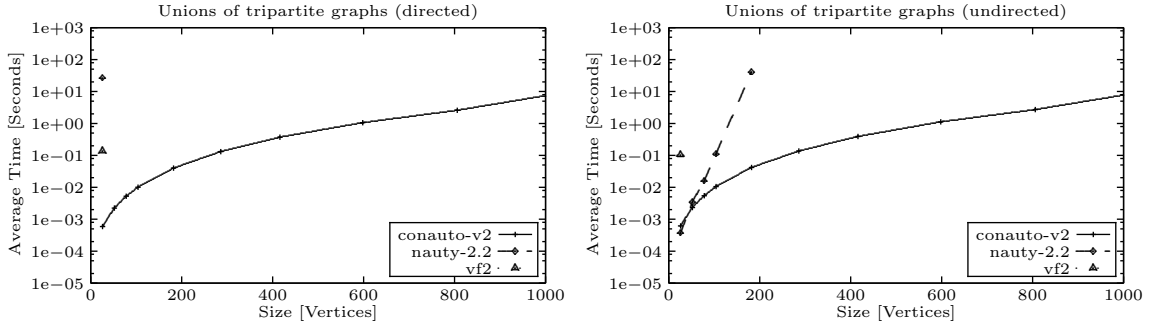


Figure 8.8: Performance of conauto-v2 with isomorphic unions of tripartite graphs.

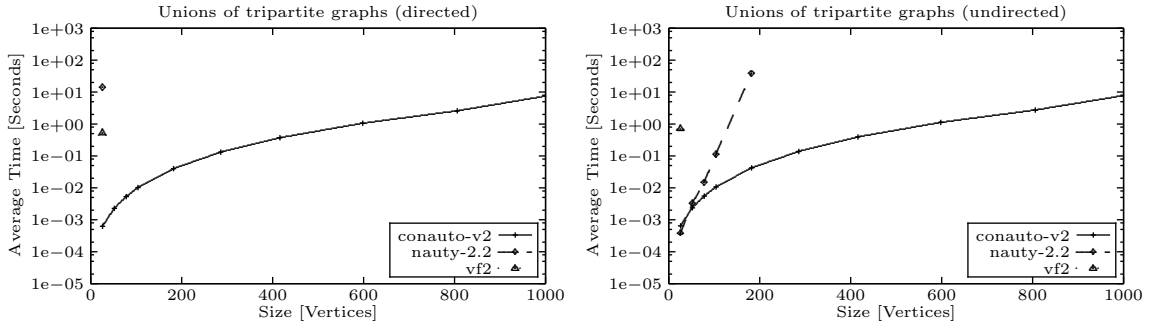


Figure 8.9: Performance of conauto-v2 with non-isomorphic unions of tripartite graphs.

Figure 8.10 shows the results for the unions of small strongly regular graphs. Comparing these with those of Figure 7.7, conauto-v2 shows an improvement with respect to conauto-v1. As in the case of unions of tripartite graphs, this is due to the fact that conauto-v1 generates sequences of partitions for both graphs before calling algorithm *Match4* while conauto-v2 only generates one of them.

The point-line graphs of Desarguesian projective planes have proved to be resistant to all our efforts. It has been impossible to improve the results obtained with conauto-v0. Our only relief is that although our algorithm is slower than both nauty and vf2 (for this family of graphs), their behavior is only slightly better than that of conauto-v2. They are able to give an answer for the graphs of 182 vertices, while conauto-v2 stops at 146 vertices. None of them seems to work in polynomial time for this family of graphs. In a trace of algorithm conauto-v2, we have observed that the number of paths explored in the search process is asymptotically similar to of Miller's $O(n^{\log \log n + O(1)})$ bound for the isomorphism of projective planes [52]. Nevertheless, this

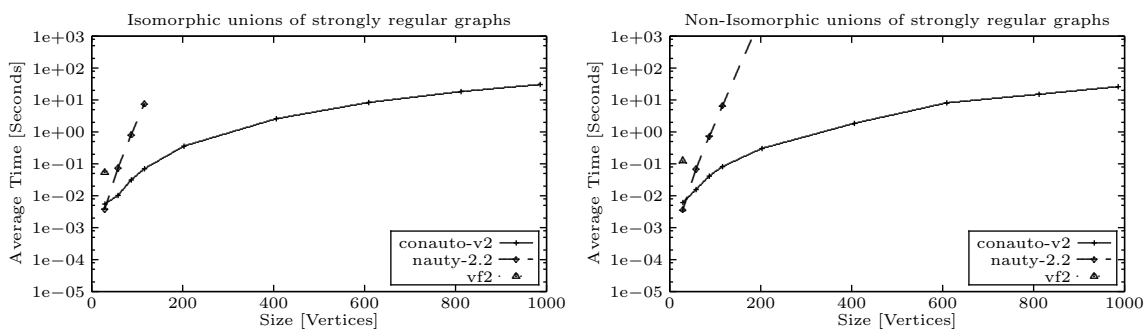


Figure 8.10: Performance of conauto-v2 with unions of strongly regular graphs.

is not necessarily a lower bound, so some heuristic or invariant might help improve the results in the future.

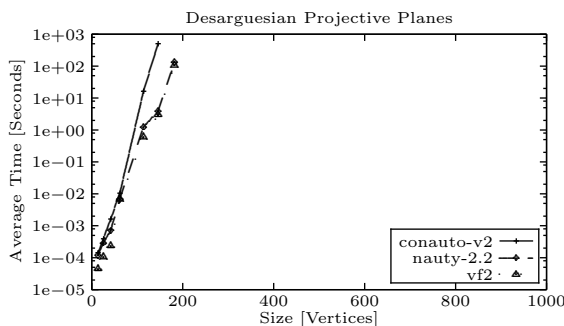


Figure 8.11: Performance of conauto-v2 with point-line graphs of Desarguesian projective planes.

Our algorithm has proved, in practice, to be efficient for a wide variety of families of graphs, even for families of graphs that are known to be hard for nauty (the reference package for graph isomorphism testing), such as Miyazaki's Fürer gadgets, and unions of regular graphs (in particular strongly regular graphs). The fact that vf2 runs for Desarguesian projective planes as fast as nauty makes us think that maybe it is possible to improve the performance of conauto for these graphs just by adding some new heuristic to the algorithm.

Chapter 9

Complexity Analysis

In this section, we study time and space complexity of our algorithm `conauto-v2`. First we focus on space complexity. Then, we prove that, with high probability, our algorithm takes polynomial time for random graphs $G \in \mathcal{G}(n, p)$.

9.1 Space Complexity

This section is devoted to studying the space complexity of algorithm `conauto-v2`. It will be shown that this space complexity is limited to $O(n^2)$, where n is the number of vertices of the graph. We assume words of $O(\log n)$ bits, since they need to store vertex identifiers. Among the data structures used by `conauto-v2`, they are the ones necessary to represent the adjacency matrices of the graphs being tested, which needs $O(n^2)$ words for a graph of n vertices.

Each partition may be represented using $O(n)$ words. The space needed for a sequence of partitions depends on the number of partitions in the sequence. At each step in the refinement procedure, a vertex or a set refinement is performed. That means that, at least one vertex is discarded (in the case of a vertex refinement), or at least one new cell is generated (in the case of a set refinement). Since the refinement process stops when there are no remaining vertices (in fact when there are no remaining vertices with links, because they would be discarded in the next step) or when all the cells in the partition are singleton, then in at most $2n - 1$ steps, the stop condition is reached. Note that if there are two vertices, in one set refinement both vertices would be in singleton cells, and in one vertex refinement, only one vertex would be left, so one refinement step is enough. Hence, a sequence of partitions has at most $2n$ partitions, what requires $O(n^2)$ words.

Each orbit partition may be represented using $O(n)$ words, so a sequence of orbit partitions needs at most $O(n^2)$ words. In fact, since a sequence of orbit partitions is computed only for graphs for which the original sequence of partitions has only one backtracking point remaining, only a few orbit partitions will be generated. In this process, also sequences of partitions are generated, but since they are discarded after being used, it is only necessary to store in memory one extra sequence at a time. Hence, the space needed to store the orbit partitions is also limited to $O(n^2)$ words. This yields a total amount of space required by our algorithm to be $O(n^2)$ words, or $O(n^2 \log n)$ bits.

9.2 Time Complexity

9.2.1 Generation of a Sequence of Partitions

Concerning time complexity, we first consider the two basic operations used in the process of partition refinement: vertex and set refinements. A vertex refinement requires testing a row or column of the adjacency matrix of the graph. That needs $O(n)$ time. Then the vertices in each cell must be classified in at most 4 different subcells, what may be done also in $O(n)$ steps using *counting sort* [18, pp. 168–170]. Thus, a vertex refinement may be done in $O(n)$ time. In a set refinement, all the vertices in the partition need to be tested with all the vertices in the pivot cell. In the worst case there would be only one cell in the partition, what requires comparing each vertex with all the others, clearly $O(n^2)$ comparisons.

The generation of a sequence of partitions starts with the degree partition of the graph. That means going through the whole adjacency matrix, what requires $O(n^2)$ operations, and ordering the vertices according to their degree, what may be done using *merge sort* [18, pp. 29–35] in $\Theta(n \log n)$ steps. Next, in the main loop of Algorithm 14 (*GenerateSequenceOfPartitions2*, lines 8–37) there are two different possibilities for each iteration. Let us analyze the time-complexity of each one:

1. If $|S_{p_l}^l| = 1$, i.e., the chosen pivot set has size 1, then a vertex refinement is performed, so this case has $O(n)$ cost.
2. Otherwise, set refinements are tried until one succeeds, or every possibility has been tried. In case there is only one cell in the partition, the cost of the unique set refinement in the loop would be in the worst case $O(n^2)$. In case there are various valid cells to be tried, let $S^l = (S_1^l, \dots, S_{r_l}^l)$ be the partition to be refined and let $V^l = \bigcup_{j=1}^{r_l} S_j^l$. The cost of trying cell S_i^l , $i \in \{1, \dots, r_l\}$ is $O(|S_i^l| |V^l|)$, and the cost of trying all the cells in the partition, $\sum_{i=1}^{r_l} O(|S_i^l| |V^l|)$. Since $\sum_{i=1}^{r_l} |S_i^l| = |V^l|$, then the cost would be $O(|V^l|^2)$. Hence, for the general case we can state that the cost of the while loop (lines 15–24) is in the worst case $O(n^2)$. If that fails, a vertex refinement takes place, but that has only a cost of $O(n)$, so the cost of this execution path remains $O(n^2)$.

If each vertex in the graph has a different degree, then the sequence of partitions has only one partition. In this case, the cost of generating its sequence of partitions would be $O(n^2)$ (the main loop of Algorithm 14 has no iteration). Otherwise, we know (Section 9.1) that the number of refinement steps in a sequence of partitions is $O(n)$. Thus, if the sequence of partitions only involves vertex refinements (Case 1 above), the generation of the sequence of partitions would require $O(n^2)$ operations. If set refinements are needed at some step in the generation of the sequence of partitions, then the generation of a complete sequence of partitions may need $O(n^3)$ steps. If a sequence of partitions has backtracking points, then for sure it has involved at least one set refinement, that that was performed unsuccessfully on the pivot cell subsequently used for the vertex refinement at the backtracking point. Hence, the total time required to generate a sequence of partitions (execution of Algorithm 14) is $O(n^3)$, and may be as low as $O(n^2)$ in some cases.

9.2.2 Finding a Match

The time complexity of Algorithm 17 (*Match₄*) is harder to bound. Since it is a backtracking procedure, its cost depends greatly on its input. The number of backtracking points conditions the time complexity, and the number of backtracking points is determined by the structure of the graphs. However, it is easy to estimate the time complexity of the algorithm in the case that there are no backtracking points. In this case, the algorithm tries to generate a sequence of partitions compatible with the original one in a straightforward fashion. Hence, the time complexity of generating this new sequence of partitions is the same needed to generate the first one: $O(n^2)$ or $O(n^3)$ depending on the necessity to perform set refinements or not (cases 1 and 2 above).

Let α be the number of backtracking points. Then the time complexity is $O(n^{\alpha+3})$. In case a family of graphs yields always sequences of partitions with a (bounded by a constant) number α of backtracking points, the cost of the algorithm would remain polynomial, though the exponent increases according to the number of existing backtracking points α . If the number of backtracking points depends on the size of the graph, then the time-complexity of the algorithm becomes superpolynomial. That seems to be the case of, for example, the point-line graphs of Desarguesian projective planes.

9.2.3 Time Complexity for Undirected Graphs

Several authors have investigated the isomorphism of random graphs [5, 10, 22]. Their approach is to describe an algorithm, usually very simple, for the graph isomorphism problem, that works for a certain family of graphs. Then, they show that the probability of a random graph belonging to that family tends to 1 when the number of vertices of the graph tends to infinite.

In [5], the algorithm proposed works in $O(n^2)$ and computes a canonical labeling of a graph, provided that it belongs to a family of graphs \mathcal{K} . The authors also prove that the probability that a random graph $G(n, 1/2)$ on n vertices belongs to this family is greater than $1 - \sqrt[7]{1/n}$ for sufficiently large n . Their algorithm works as follows for a graph G on n vertices:

1. Compute $r = \lceil 3 \log n / \log 2 \rceil$.
2. Compute the degree of each vertex of the graph.
3. Order the vertices by degree; call them $v(1), \dots, v(n)$. Denote by $d(i)$ the degree of $v(i)$.
4. If $d(i) = d(i+1)$ for some $i \in \{1, \dots, r-1\}$, then $G \notin \mathcal{K}$, FAIL.
5. Let $a(i, j) = 1$ if $v(i)$ and $v(j)$ are adjacent, and $a(i, j) = 0$ otherwise. Then, compute $f(v(i)) = \sum_{j=1}^r a(i, j)2^j$ for each $i \in \{r+1, \dots, n\}$.
6. Order the vertices $v(r+1), \dots, v(n)$ according to their f -value; call them $w(r+1), \dots, w(n)$.
7. If $f(w(i)) = f(w(i+1))$ for some $i \in \{r+1, \dots, n\}$, then $G \notin \mathcal{K}$, FAIL.
8. Label $v(i)$ by i for $i \in \{1, \dots, r\}$, and $w(j)$ by j for $j \in \{r+1, \dots, n\}$. This labeling will be canonical, and $G \in \mathcal{K}$. END.

Here, a graph G belongs to \mathcal{K} if its r vertices of highest degree may be distinguished by their degree, and the other vertices of the graph may be distinguished by their adjacencies with the r vertices of higher degree. In our algorithm, this is analogous to the case when the degree partition is able to distinguish at least a set Z of r vertices (put them in singleton cells), and then, applying successive vertex refinements using these as pivot vertices, all the other vertices are distinguished. In this case, our algorithm also generates the sequence of partitions in $O(n^2)$

time, and is not restricted to have in Z the r vertices of higher degree. Hence, our algorithm works in $O(n^2)$ time for more graphs than the one proposed in [5]. Note as well that the authors of [5] only consider undirected graphs. Consequently, the probability that our algorithm works in $O(n^2)$ time is at least the same of their algorithm for undirected graphs in $G(n, 1/2)$.

The algorithm proposed by Czajka and Pandurangan [22] covers a wider family of random graphs. It works with high probability for random graphs $G(n, p)$ for $p \in [\omega(\ln^4 n/n \ln \ln n), 1 - \omega(\ln^4 n/n \ln \ln n)]$. Like the algorithm of [5], this algorithm provides a canonical labeling for all graphs in a given family. The algorithm is based on the degree neighborhood of the vertices, which is a sorted list of the degrees of the vertex's neighbors. It works as follows:

1. Compute vertex degrees.
2. Compute the degree neighborhood for each vertex.
3. Sort the vertices by degree neighborhood in lexicographic order.
4. If the degree neighborhoods are not distinct for each vertex, FAIL.
5. Label the vertices in the sorted order.

If this algorithm does not fail, it computes a canonical labeling of the graph. The authors show that it can be computed in $O(|V| + |E|)$ (V is the vertex set and E the edge set). This algorithm does not have a trivial correspondence with the way Algorithm 14 computes a sequence of partitions for a graph. However, the classification performed by this algorithm is equivalent to a refinement where each vertex is classified according to the number of adjacent vertices it has of each degree. This may be accomplished starting from the degree partition, and then performing successive set refinements using a different cell in the original partition as the pivot set for each refinement. Since our algorithm keeps applying refinements until it finds a partition with singleton cells, or with no remaining links, it will be able to find a sequence of partitions without backtracking points for at least as many graphs as this algorithm. Our algorithm's time complexity will be $O(n^3)$ in this case, since we can not fix the number of set refinements needed.

Recall that our algorithm generates a sequence of partitions for each of the graphs considered, and then keeps the one with fewer backtracking points. The above argument shows that if a graph is labeled with the algorithm of [22], then its sequence of partitions has no backtracking points. Hence, we can apply the analysis in [22], and conclude that, with high probability, our algorithm has time complexity $O(n^3)$ if the graphs to be compared are undirected random graphs $G(n, p)$ for $p \in [\omega(\ln^4 n/n \ln \ln n), 1 - \omega(\ln^4 n/n \ln \ln n)]$.

Chapter 10

Conclusions and Further Improvements

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

10.1 Conclusions

We have developed an algorithm for graph isomorphism testing that meets the requirement to be fast in practice. We have compared its practical performance with the world's most acknowledged package for graph isomorphism testing, Brendan McKay's *nauty*. Our algorithm has outperformed it for a number of graph families, what shows that our new approach to the problem, not trying to compute the whole automorphism group of the graphs may be better, at least in some cases.

Since it is known that computing the automorphism group of a graph is harder than isomorphism testing, our approach may open a new research line that may yield even better results in future, finding a way to deal with the most hard cases, such as the point line graphs of Desarguesian projective planes.

We have also shown that our algorithm requires a limited amount of memory, $O(n^2)$, what is a satisfactory bound. We have also shown that it has polynomial time complexity with high probability (for random graphs), with a best case of $O(n^2)$ time complexity.

An early version of *conauto* has been recoded and included in the LEDA C++ class library of algorithms [65], commercialized by Algorithmic Solutions Software GmbH. As noted in [65], both implementations of *conauto* have a very uniform behavior. It is also worth noting that the LEDA implementation was found to be slower than ours.

10.2 Future extensions

One line to explore is the use of more sophisticated automorphism management. This should consider automorphisms that are known to exist but have not been considered in the latest version of the algorithm. (For instance, in Chapter 7 automorphic relations were identified

under certain conditions; but have not been stored nor used further.) Hence, something like the tools presented in [40, Chapter 6] should be added to the algorithm to store and operate with automorphisms, and what is more interesting, to determine which automorphisms are applicable at some step during the search for a matching.

In Chapter 8, the search for automorphisms was extended in the case the only backtracking points in the original sequence were the first two ones. Adding tools to manage automorphisms efficiently would allow a further extension of this search for automorphisms, what might improve the performance in some other cases apart from the Latin square graphs targeted there.

Another way to improve the performance of the algorithm would be to add new refinement techniques, i.e. new invariants that might help eliminate some backtracking points. The problem with this approach is that the new invariants should be fast to implement, and we would like the algorithm to decide when a certain invariant is useful without the intervention of the user. (This would be unlike the case of nauty, where many invariants may be used at the user's request to speed up the algorithm.)

One idea is to try additional new refinement techniques when we find that there are no valid cells, and a backtracking point would arise. Some possibilities we may consider are the following:

- Refine using the combination of two pivot cells. Consider the pivot cells P_i and P_j . For each vertex v in a cell S , we count the number of vertices in P_i that are accessible from v using the different paths of length 2 that go through cell P_j . In this case, we need to find some heuristics that help discarding pairs of pivot cells to avoid trying pairs of cells that we could determine in advance that are useless. Otherwise, the cost of this invariant might make it inefficient.
- Sometimes, a vertex refinement followed by one or more set refinements might help differentiate among the vertices in a cell. The process consists of choosing a cell to be refined. Then, for each vertex in this cell, a vertex refinement followed by as many set refinements as possible are performed. Finally, the vertices are classified according to the structure of the final partition obtained, and the refinement process followed. A simplified version would be to perform only one set refinement after the vertex refinement.

An idea that we have not exploited yet is acquiring knowledge of non-automorphisms, in addition to automorphisms. During the search for automorphisms, when at some backtracking point not all the vertices in the pivot cell are equivalent, we keep that backtracking point, and probably, not all the vertex equivalences discovered will be applicable, due to the partial information of automorphisms managed by the algorithm. One thing that could help is keeping knowledge of how many vertices in the pivot cell have been found to be equivalent. If there are n vertices in the pivot cell, and k of them are equivalent to the pivot vertex, since isomorphisms preserve automorphisms, in the search for a sequence of partitions compatible with the target, it would be necessary to try at most $n - k$ vertices prior to discarding this path in the search space. The cost for the algorithm is just counting the successful vertices at each backtracking point, which is very little.

Something a bit more sophisticated would be to take note of the level at which an incompatibility has been found during the tests for equivalence among the vertices in the pivot cells, and, at once, take note also of the exact point and cause of the discrepancy among the sequences of partitions. Perhaps, several vertices in the pivot cell may have the same discrepancy, but maybe also there can be several types of discrepancies, so counting all of them, it might be faster to find that the incompatibility found during the search for a sequence of partitions compatible

with the target is not among the possible discrepancies, or that the amount of vertices for a given discrepancy might be exceeded.

In Section 9.2.3, it was shown that our algorithm finishes in polynomial time with high probability for randomly chosen undirected graphs. A more thorough study is left for future work. Among the lines to follow, one would explore a similar result for directed graphs. A second line would try to bound the expected time complexity for random graphs. To do so, we could attempt to bound the amount of graphs for which the algorithm works in superpolynomial time, and to bound the worst-case complexity for these graphs.

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] M. Abdulrahim and M. Misra. A graph isomorphism algorithm for object recognition. *Pattern Analysis and Applications*, 1(3):189–201, 1998.
- [3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, Boston, MA, USA, 1974.
- [4] V. Arvind and Piyush P. Kurur. Graph isomorphism is in SPP. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 743–750, Washington, DC, USA, 2002. IEEE Computer Society.
- [5] L. Babai, P. Erdős, and M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, August 1980.
- [6] László Babai and Luděk Kučera. Canonical labeling of graphs in linear average time. In *FOCS '79: Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 39–46. IEEE Computer Society, 1979.
- [7] László Babai, D. Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 310–324, New York, NY, USA, 1982. ACM Press.
- [8] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183, New York, NY, USA, 1983. ACM Press.
- [9] A. T. Bertziss. A backtrack procedure for isomorphism of directed graphs. *Journal of the ACM*, 20(3):365–377, 1973.
- [10] B. Bollobás. *Random Graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, second edition, 2001.
- [11] R. Boppana, J. Hastad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25:27–32, 1987.
- [12] Joachim Braun, Ralf Gugisch, Adalbert Kerber, Reinhard Laue, Markus Meringer, and Christoph Rucker. MOLGEN-CID – A canonizer for molecules and graphs accessible through the internet. *Journal of Chemical Information and Computer Sciences*, 44(2):542–548, 2004.

- [13] 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.
- [14] 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.
- [15] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Subgraph transformations for the inexact matching of attributed relational graphs. In *Graph Based Representations in Pattern Recognition*, volume 12 of *Computing Supplementum*, pages 43–52, 1998.
- [16] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proceedings of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations*, pages 149–159, Ischia, Italy, May 2001.
- [17] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the VF graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing*, pages 1172–1177, Venice (Italy), September 1999. IEEE Computer Society.
- [18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
- [19] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, 1970.
- [20] Derek G. Corneil and David G. Kirkpatrick. A theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. Comput.*, 9(2):281–297, 1980.
- [21] 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/>.
- [22] Tomek Czajka and Gopal Pandurangan. Improved random graph isomorphism. *Journal of Discrete Algorithms*, 6(1):85–92, 2008.
- [23] Carl Ebeling. GeminiII: A second generation layout validation tool. In *Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD-88)*, pages 322–325. IEEE Computer Society Press, November 1988.
- [24] Carl Ebeling and Ofer Zajicek. Validating VLSI circuit layout by wirelist comparison. In *Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD-83)*, pages 172–173. IEEE Computer Society Press, September 1983.
- [25] 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.
- [26] I. S. Filotti and Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 236–243, New York, NY, USA, 1980. ACM Press.
- [27] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations*, pages 188–199, Ischia, Italy, May 2001.

- [28] Scott Fortin. The graph isomorphism problem. Technical Report TR 96-20, University of Alberta, Edmonton, Alberta, Canada, July 1996.
- [29] Martin Fürer. Graph isomorphism testing without numerics for graphs of bounded eigenvalue multiplicity. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 624–631, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.
- [30] C. D. Godsil and B. D. McKay. Constructing cospectral graphs. *Aequationes Mathematicae*, 25:257–268, 1983.
- [31] Mark Goldberg. The graph isomorphism problem. In Jonathan L. Gross and Jay Yellen, editors, *Handbook of graph theory*, Discrete Mathematics and its Applications, chapter 2.2, pages 68–78. CRC Press, 2003.
- [32] M. Gori, M. Maggini, and L. Sarti. Graph matching using random walks. In *Proceedings of the 17th International Conference on Pattern Recognition*, volume 3, pages 394–397, August 2004.
- [33] Marco Gori, Marco Maggini, and Lorenzo Sarti. The RW2 algorithm for exact graph matching. In Sameer Singh, Maneesha Singh, Chidanand Apté, and Petra Perner, editors, *Pattern Recognition and Data Mining, Third International Conference on Advances in Pattern Recognition, ICAPR 2005, Bath, UK, August 22-25, 2005, Part I*, volume 3686 of *Lecture Notes in Computer Science*, pages 81–88, 2005.
- [34] 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/>.
- [35] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184, New York, NY, USA, 1974. ACM Press.
- [36] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [37] Gow-Hsing King and Wen-Guey Tzeng. A new graph invariant for graph isomorphism: Probability propagation matrix. *Journal of Information Science and Engineering*, 15(3):337–352, 1999.
- [38] Johannes Köbler, Uwe Schöning, and Jacobo Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2(4):301–330, 1992.
- [39] William Kocay. On writing isomorphism programs. In W. Wallis, editor, *Computational and Constructive Design Theory*, pages 135–175, Dordrecht, The Netherlands, 1996. Kluwer.
- [40] 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.
- [41] Felix Lazebnik and Andrew Thomason. Orthomorphisms and the construction of projective planes. *Mathematics of Computation*, 73(247):1547–1557, 2004.
- [42] G. Levi. Graph isomorphism: A heuristic edge-partitioning-oriented algorithm. *Computing*, 12(4):291–313, 1974.

- [43] X. Liu and D. J. Klein. The graph isomorphism problem. *Journal of Computational Chemistry*, 12(10):1243–1251, 1991.
- [44] 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.
- [45] R. Mathon. Sample graphs for isomorphism testing. *Congressus Numerantium*, 21:499–517, 1978.
- [46] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979.
- [47] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.
- [48] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [49] Brendan D. McKay. *Nauty User’s Guide (version 2.2)*. Computer Science Department, Australian National University, 2002. <http://cs.anu.edu.au/~bdm/nauty/nug.pdf>.
- [50] Brendan D. McKay. The nauty page. Computer Science Department, Australian National University, 2004. <http://cs.anu.edu.au/~bdm/nauty/>.
- [51] Bruno T. Messmer and Horst Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.
- [52] Gary L. Miller. On the $n^{\log n}$ isomorphism technique (a preliminary report). In *STOC ’78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 51–58, New York, NY, USA, 1978. ACM Press.
- [53] Hari Ballabh Mittal. A fast backtrack algorithm for graph isomorphism. *Information Processing Letters*, 29(2):105–110, September 1988.
- [54] 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.
- [55] G. E. Moorhouse. Projective planes of small order. Department of Mathematics, University of Wyoming, 2005. <http://www.uwyo.edu/moorhouse/pub/planes/>.
- [56] Mikhail E. Muzychuk and Gottfried Tinhofer. Recognizing circulant graphs in polynomial time: An application of association schemes. *Electronic Journal of Combinatorics*, 8(1), 2001.
- [57] Greeshma Neglur, Robert L. Grossman, and Bing Liu. Assigning unique keys to chemical compounds for data integration: Some interesting counter examples. *Lecture Notes in Computer Science*, 3615:145–157, Aug 2005.
- [58] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. SubGemini: identifying subcircuits using a fast subgraph isomorphism algorithm. In *DAC ’93: Proceedings of the 30th international conference on Design automation*, pages 31–37, New York, NY, USA, June 1993. ACM Press.

- [59] Georg Pelz and Uli Roettcher. Circuit comparison by hierarchical pattern matching. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD-91)*, pages 290–293. IEEE Computer Society Press, 1991.
- [60] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.
- [61] Sven Reichard. Personal communication, April 2002.
- [62] 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.
- [63] Douglas C. Schmidt and Larry E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM*, 23(3):433–445, 1976.
- [64] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988.
- [65] Johannes Singler. Graph isomorphism implementation in LEDA 5.1. Technical report, Algorithmic Solutions Software GmbH, Dec. 2005.
- [66] 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/>.
- [67] 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.
- [68] 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>.
- [69] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
- [70] Stephen H. Unger. GIT – A heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM*, 7(1):26–34, 1964.
- [71] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1):59–68, 2003.
- [72] David Weininger, Arthur Weininger, and Joseph L. Weininger. SMILES 2. Algorithm for generation of unique SMILES notation. *Journal of Chemical Information and Computer Sciences*, 29(2):97–101, 1989.
- [73] Boris Weisfeiler, editor. *On construction and identification of graphs*. Number 558 in Lecture Notes in Mathematics. Springer, 1976.
- [74] Jin yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.
- [75] V. N. Zemlyachenko, N. M. Korneenko, and R. I. Tyshkevich. Graph isomorphism problem. *Journal of Mathematical Sciences (Historical Archive)*, 29(4):1426–1481, 1985. Translated

from Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V. A. Steklova AN SSSR, Vol. 118, pp. 83–158, 1982.

Resumen en Castellano

Antecedentes

El problema de, dados dos grafos, determinar si son isomorfos o no, ha sido estudiado durante décadas por científicos de diversos ámbitos, tanto por su considerable campo de aplicación práctica, como por su propio interés desde el punto de vista teórico.

El isomorfismo de grafos se ha usado para reconocimiento de formas en visión artificial, para minería de datos, o en química matemática, para obtener identificadores únicos de los compuestos químicos, algo imprescindible en catalogación. La obtención de un identificador canónico para un compuesto (o más genéricamente para un grafo) es un problema directamente derivado del de calcular el grupo de automorfismo de un grafo, y habitualmente se ha utilizado también para determinar el isomorfismo. Sin embargo, calcular el grupo de automorfismo de una pareja de grafos puede ser más difícil que comprobar si son isomorfos.

Desde el punto de vista de complejidad algorítmica, también es un problema interesante, por el hecho de que su complejidad no está clara. Aunque está claro que está en NP, no se sabe si es NP-completo. Además, hay muchos indicios que hacen sospechar que no es un problema NP-completo. Por ejemplo, contar el número de isomorfismos de un grafo tiene aproximadamente la misma complejidad que determinar la existencia de un isomorfismo. Sin embargo, la contabilización tiende a ser mucho más compleja que la decisión en todos los problemas NP-completos. De hecho, la complejidad exacta del problema es, aun hoy, un problema abierto.

A pesar de todo esto, se conocen muchos casos de familias de grafos para las que existe una solución del problema en tiempo polinómico. Por ejemplo, en el caso de que el grado de los vértices esté acotado. Todo esto ha hecho que muchos investigadores hayan buscado algoritmos eficientes para resolver el problema usando diversos enfoques.

Una posibilidad es tratar de buscar directamente un isomorfismo, utilizando un algoritmo de búsqueda clásico, y podando ramas mediante heurísticos y el uso de invariantes. Esta opción suele ser buena en el caso de que el grafo tenga muy pocos automorfismos, y en el caso de que los grafos evaluados sean isomorfos. Sin embargo, si los grafos no son isomorfos y tienen muchos automorfismos, el algoritmo puede recorrer muchas ramas automorfas del árbol de búsqueda antes de descubrir que no hay solución posible. Este es el caso del algoritmo *vf2*.

Otra posibilidad es encontrar un etiquetado canónico de cada grafo, y a continuación comparar directamente los etiquetados canónicos. Esto supone determinar el grupo de automorfismo de los grafos, lo que, como se ha comentado anteriormente, puede ser más complejo de lo estrictamente necesario. Sin embargo, da muy buenos resultados cuando los grafos tienen muchos automorfismos. El ejemplo por excelencia de este tipo de algoritmos es *nauty*.

Objetivos

El objetivo de esta tesis es proponer un algoritmo novedoso para resolver el problema del isomorfismo de grafos, que sea eficiente (tiempo polinómico) en la mayoría de los casos, y que sea completo, es decir que dé una respuesta positiva o negativa en todos los casos (hay algoritmos que son muy rápidos pero no son completos). Además de la eficiencia teórica de nuestro algoritmo, también nos interesa especialmente la eficiencia práctica. Para ello, se ha programado el algoritmo en C, y se ha comparado su tiempo de ejecución con los de `vf2` y `nauty` para diversas familias de grafos.

Además, otro requisito impuesto a nuestro algoritmo es que la cantidad de memoria requerida sea $O(n^2)$ donde n es el número de vértices del grafo. Este requisito es necesario para poder manejar grafos de gran tamaño. Téngase en cuenta que en el caso de $O(n^3)$, para grafos de miles de vértices, harían falta gigabits de memoria. Dado que la matriz de adyacencia de un grafo ya requiere $O(n^2)$ bits, no es posible reducir la complejidad asintótica por debajo de este valor.

Finalmente, nuestro algoritmo debe suponer una mejora frente a los algoritmos existentes. Esto es, en el caso de familias de grafos para las que los algoritmos existentes tienen una complejidad temporal aparentemente polinómica, el nuestro debería tener un comportamiento similar, y además, debería mejorar los resultados para algunas familias de grafos con las que los algoritmos actuales tienen un comportamiento exponencial. Además, queremos que nuestro algoritmo tenga un comportamiento lo más uniforme posible para grafos con características similares.

Batería de Pruebas

Con el fin de comparar la eficiencia práctica de nuestro algoritmo con otros existentes, hemos generado una batería de pruebas con familias de grafos de características diversas:

- **Grafos aleatorios.** Los grafos aleatorios son un ejemplo clásico de grafos, y además son los grafos más frecuentes, por lo que parece impensable una batería de prueba sin contar con ellos. Cualquier algoritmo para determinar el isomorfismo de grafos debe ser rápido con esta familia de grafos, si pretende tener utilidad práctica.
- **Mallas.** Hay un tipo especial de mallas regulares dirigidas que resultaban especialmente duras para la versión 2.0 del algoritmo `nauty`, mientras que el algoritmo `vf2` era muy rápido con ellas. Sin embargo, `nauty` era más rápido que `vf2` con la versión no dirigida de esos grafos. La versión 2.2 de `nauty` mejoró considerablemente su rendimiento con la versión dirigida, pero aún era más de un orden de magnitud más lento que con la versión no dirigida. Nosotros perseguíamos un algoritmo que tuviera un comportamiento similar con independencia de que los grafos fueran dirigidos o no.
- **Grafos de Miyazaki.** Basándose en una construcción de Martin Fürer, Miyazaki construyó una familia de grafos que, aún teniendo grado constante, hacía que `nauty` tuviera un comportamiento exponencial, incluso usando invariantes adicionales. Esto parecía deberse a la forma en la que `nauty` detectaba los automorfismos. Nosotros pretendíamos evitar este problema usando una técnica diferente para la búsqueda de automorfismos.
- **Grafos fuertemente regulares.** Los grafos fuertemente regulares son muy regulares, pero no necesariamente tienen un grupo de automorfismo muy grande. Esto puede hacer que algunos sean muy fáciles de procesar, mientras que otros resulten mucho más difíciles,

sobre todo en el caso de que los grafos a procesar no sean isomorfos y el algoritmo no se base en etiquetados canónicos.

- **Grafos unión.** Hemos considerado grafos unión en los que cada componente tiene sus vértices conectados con todos los vértices de todas las demás componentes del grafo, en lugar de hacer la unión disjunta. De este tipo, hemos considerado dos familias. Grafos unión de grafos fuertemente regulares de pequeño tamaño, y grafos unión de grafos tripartitos, de pequeño tamaño también, diseñados por nosotros mismos. Es conocido por el propio McKay que nauty es exponencial para este tipo de construcciones.
- **Planos proyectivos.** Se han considerado los grafos de incidencia punto-recta de los planos proyectivos finitos desarguesianos. Es conocido que este tipo de grafos es de los más duros para el isomorfismo.

Algoritmos

Para desarrollar nuestro algoritmo, hemos partido de un algoritmo relativamente sencillo, que aplica las ideas principales de nuestra tesis, y lo hemos ido enriqueciendo progresivamente. Nuestras líneas maestras son las siguientes:

- En lugar de considerar la matriz de adyacencias clásica M en la que un arco desde el vértice i hasta el vértice j se representa mediante un 1 en la posición M_{ij} , hemos usado una matriz con 4 valores posibles: 0, 1, 2, 3, de forma que 0 indica que i y j no son adyacentes, 1 indica que hay un arco que va de j a i , 2 indica que hay un arco que va de i a j , y 3 que hay un arco en cada sentido o una arista en un grafo no dirigido. Esto permite simplificar el procesado de grafos dirigidos.
- Usamos clasificación de vértices para construir particiones que faciliten la construcción de un isomorfismo entre los dos grafos. Se comienza con la partición trivial, en la que todos los vértices están en el mismo conjunto, y se van reclasificando los vértices de acuerdo el número de vértices de cada tipo que tienen respecto de un conjunto de la partición, denominado conjunto pivote. El proceso termina cuando todas las celdas de la partición tienen un único vértice. Si en algún momento no es posible dividir ninguna celda, se realiza una individualización, esto es, se elige una celda y de ella se extrae un vértice que se sitúa en una nueva celda, y se sigue intentando refinar la partición con esas dos nuevas celdas.
- La secuencia de particiones obtenida para un grafo, junto con los pivotes usados en cada refinamiento, identifican unívocamente al grafo. Una vez obtenida la secuencia de particiones para uno de los grafos, se intenta generar una secuencia de particiones para el otro grafo, que sea compatible con la del primer grafo. Si no se han realizado individualizaciones de vértices, la generación será directa (en caso de que sea posible). El problema radica en que si ha habido individualizaciones de vértices, entonces para el segundo grafo habrá que probar, en principio, individualizando todos los vértices del grupo pivote correspondiente, para encontrar aquel que corresponde con el usado para el primer grafo. Esto requiere un algoritmo de vuelta atrás, que puede tener un coste exponencial.

Al algoritmo básico se le han añadido nuevas funcionalidades que le permiten detectar automorfismos en los grafos (aunque no necesariamente todos), y usar estos automorfismos para podar drásticamente el árbol de búsqueda. Además, es capaz de detectar componentes en los grafos, y aislarlas de forma que su comportamiento mejora considerablemente con grafos unión.

Lo más importante de las sucesivas mejoras que se van introduciendo es que se hacen sin ocasionar una merma sustancial del rendimiento para los casos en los que el algoritmo anterior ya tenía un buen comportamiento. Cada nueva versión que se genera se compara con los otros algoritmos mediante la batería de pruebas, para comprobar que se va consiguiendo el objetivo perseguido.

Para terminar, se realiza un pequeño análisis de complejidad, tanto espacial como temporal, del algoritmo final.

Conclusiones

Hemos desarrollado un algoritmo para el isomorfismo de grafos que cumple el requisito de ser rápido en la práctica. Hemos comparado su rendimiento, mediante una batería de pruebas, con el algoritmo más reconocido internacionalmente, el nauty de Brendan McKay. Nuestro algoritmo ha mejorado sustancialmente sus tiempos para varias familias de grafos, lo que demuestra que nuestro enfoque del problema, sin intentar obtener el grupo de automorfismo completo de los grafos, puede ser una opción mejor, al menos en algunos casos.

Puesto que es sabido que calcular el grupo de automorfismo de un grafo puede ser más duro que el isomorfismo, nuestro enfoque puede abrir una nueva línea de investigación que puede dar mejores resultados en el futuro, encontrando un modo de tratar los casos más duros, como el de los grafos de adyacencia punto-recta de los planos proyectivos finitos desarguesianos.

Hemos demostrado que nuestro algoritmo tiene una complejidad espacial $O(n^2)$ donde n es el número de vértices de los grafos, con lo que se ha alcanzado el objetivo en cuanto al uso de memoria.

Demostramos también que nuestro algoritmo tiene una complejidad temporal, en el caso mejor, $O(n^2)$, es decir, cuadrático en el número de vértices del grafo. También demostramos que con alta probabilidad, nuestro algoritmo tiene una complejidad temporal $O(n^3)$ si consideramos grafos aleatorios $G(n, p)$ no dirigidos, para valores de $p \in [\omega(\ln^4 n/n \ln \ln n), 1 - \omega(\ln^4 n/n \ln \ln n)]$.

Una versión antigua de conauto fue recodificada e incluida en la biblioteca de clases C++ LEDA [65], comercializada por Algorithmic Solutions Software GmbH. Como se menciona en [65], tanto nuestra versión de conauto como la suya tienen un comportamiento muy uniforme. Además, debemos destacar que su versión es aproximadamente la mitad de rápida que la nuestra. Sin embargo, su recodificación del algoritmo vf2 es unas cuatro veces más rápida que la original.