AN INVESTIGATION INTO GRAPH ISOMORPHISM BASED ZERO-KNOWLEDGE PROOFS

Eric Ayeh

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2009

APPROVED:

Kamesh Namuduri, Major Professor
Murali Varanasi, Committee Member and Chair of the Department of Electrical Engineering
Parthasarathy Guturu, Committee Member
Shengli Fu, Graduate Program Coordinator
Costas Tsatsoulis, Dean of the College of Engineering
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Ayeh, Eric. An investigation into graph isomorphism based zero-knowledge proofs. Master of Science (Electrical Engineering), December 2009, 42 pp., 2 tables, 12 illustrations, bibliography, 30 titles.

Zero-knowledge proofs protocols are effective interactive methods to prove a node's identity without disclosing any additional information other than the veracity of the proof. They are implementable in several ways. In this thesis, I investigate the graph isomorphism based zero-knowledge proofs protocol. My experiments and analyses suggest that graph isomorphism can easily be solved for many types of graphs and hence is not an ideal solution for implementing ZKP.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Dr Kamesh Namuduri, Dr Murali Varanasi, and Dr Parthasarathy Guturu, who formed my advisory committee. The work accomplished in this thesis would not have been possible without the constant mentoring of Dr Kamesh Namuduri, the support and guidance of Mr. Oluwayomi Adamo. I would also like to thank Dr José Luis López Presa and Dr Takunari Miyazaki for their insight into the problem of graph isomorphism. My gratitude goes to the faculty and staff of the Electrical Engineering Department for their encouragement and motivation. Lastly, I am very grateful to my family and friends, especially my dad for being an inspiration.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF ILLUSTRATIONS

Page

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

With the exponential increase in technologies, our world is characterized by continuous exchange of data. These data are exchanged through different mediums and carry in some cases very sensitive information, which in the hands of malicious persons can have a considerable impact in our society. Imagine for instance, the impact it will have on your life if a person with bad intentions knows your social security number, the password to your bank account, the password to your house alarm system, or more importantly, if a terrorist can access some critical security information.

In fact, a common problem faced by traditional communication systems is impersonation. Defined as the act of imitating or copying someone's behavior, impersonation occurs when an eavesdropper or an intruder, by listening to a communication, gains enough information allowing him to act as either one of the two parties involved in the exchange of information.

The main question that arises from these situations is whether one could protect his/her (sensitive) information in a world where we are constantly using a password either to check an email, a bank account, or to log into a computer. In other words, how could one protect the secret key allowing access into a network?

There are several solutions available in the literature to serve this purpose. For example secret-key cryptosystems where two parties involved in an exchange of information have to agree on one or several secret keys prior to the actual exchange of data. Another approach is public-key cryptosystems where the sender and the receiver have a pair of cryptographic keys, one that is public and is available to everybody, while the other is private and kept secret.

Despite the fact that these systems help in protecting communications, they both require the communicating parties to have a certain knowledge of the secret key and are susceptible to problems such as the man-in-the middle-attack (MITM).

An alternative to the MITM attack and authentication problems in general is the use of zero-knowledge proof (ZKP) protocols. Introduced by Goldwasser, Micali and Rackoff, ZKP is a smart way to prove a node's identity without disclosing any information about the secret of that identity [1], [2]. In fact, ZKP is an interactive protocol where a prover can prove the veracity of a statement to a verifier without disclosing any other information, which could allow an eavesdropper or the verifier to impersonate him. Such a statement is usually a mathematical problem with complexity in the order of nondeterministic polynomial (NP) or NP-complete. During a ZKP interaction, the prover will try for example to convince the verifier that he/she knows the secret password to open a door without actually giving the password to the verifier. The verifier throughout the interactions will ask questions in the aim of verifying that the prover really knows the secret password. If the answer to a question is wrong, the communication is immediately terminated, or the access to the network is denied. On the

other hand, because the prover could be a malicious party trying to cheat, the verifier will continue the interaction until he/she is convinced of the identity of the prover even if the answer to a question is right. As shown in Figure 1.1 below and obtained from [20], is an example of the ZKP protocol which is based on the description in [18].

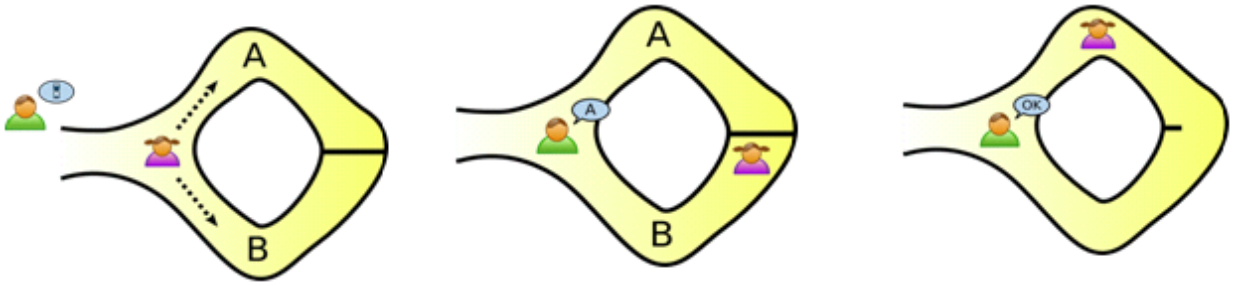
Here, Victor is the verifier and Peggy is the prover, who wants to prove Victor that she knows the secret password to open a door without disclosing any information about the password to Victor. A round of this interaction is summarized in the following three steps.

Step 1: Victor waits outside the entrance of the cave, while Peggy goes in and randomly enters through path A or B.

Step 2: Victor enters the cave and asks Peggy to come out through a path that he chooses randomly.

Step 3: If Peggy really knows the secret password, she opens the door if necessary and returns along the path requested by Victor.

Note that it is not necessary for Peggy to use the secret password if she enters in A and Victor asks her to come out through path A. Therefore, these three steps are repeated until Victor is completely convinced of the fact that Peggy really knows the secret password to open the door.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 1.1: Example of zero-knowledge proof protocol.

In this thesis, I am assessing the graph isomorphism based zero-knowledge proofs (GIZKP). The graph isomorphism problem has long been considered suitable for the zero-knowledge protocol [4], [5]; however, no work has been done toward the actual implementation of the GIZKP. Therefore, the question that I am trying to answer through this work is whether the graph isomorphism problem is a good choice for ZKP.

A zero-knowledge proof is said to be perfect, when regardless of the fact that a verifier has access to powerful computational resources, he/she will not gain any knowledge other than the veracity of the proof by participating in a round of the protocol.

Due to the fact that several practical algorithms for solving the graph isomorphism problem are available in the literature, I conjecture that in most cases, the graph isomorphism problem might not be suitable for the implementation of ZKP systems.

1.2 Organization of the Thesis

The rest of this thesis is organized as follows. Chapter 2 discusses existing literature. In Chapter 3, I first introduce the graph isomorphism problem, then provide the characteristics of the graph isomorphism based zero-knowledge proofs. Subsequently, I discuss available approaches in the literature to solve the graph isomorphism problem. Finally, I describe a program called *nauty*. Next, Chapter 4 discusses the potential problems faced during the implementation of the graph isomorphism based ZKP protocol and introduces the notion of hard graphs. Using some of these hard graphs I present in Chapter 5 the results of experiments done using the *nauty* program. Based on these results, I finally summarize the work done in this thesis in Chapter 6, and discuss possible future work.

CHAPTER 2

LITERATURE SURVEY

Since its introduction in 1985 by Goldwasser, Micali, and Rackoff [1], zero-knowledge proofs (ZKP) have evolved and attracted researchers. Used to convey solely the fact that an assertion is true, a zero-knowledge proof system has applications in cryptography and security protocols requiring authentication. Goldreich, Micali and Wigderson showed in [2] that for the implementation of interactive zero-knowledge proofs, the choice of the language is very important in order to avoid conveying additional knowledge other than the veracity of the proof. They showed also that all languages in NP have a zero-knowledge proof system [2]. In fact, quadratic residue, graph 3-colorability, prime factoring, Hamiltonian cycle for large graphs, and graph isomorphism are examples of problems that are considered to have a perfect zero-knowledge proof system [1], [2], [20]. As aforementioned, my focus in this thesis is on the graph isomorphism based ZKP.

The graph isomorphism problem which has been studied for several years by researchers in mathematics and computer science is the problem of determining if two dissimilar graphs are isomorphic or not.

Due to its various applications in fields such as image processing, pattern recognition, DNA matching, computer graphics, and organic chemistry [10], [11], [12], the graph isomorphism problem has drawn so much attention that in 1977 Read and

Corneil published a journal article that they named "The Graph Isomorphism Disease" [21].

Another factor that has drawn researchers' attention is the computational complexity of the problem. In fact, the graph isomorphism problem is known to belong to the complexity class nondeterministic polynomial (NP) time but not known to be solvable in polynomial time nor NP-complete for the general case (i.e. for any pair of graphs). This is proved in [22], where it is shown that the graph isomorphism problem is not NP-complete otherwise the hierarchy of the complexity classes collapses to its second level. Certain classes of graphs do exist that are solvable by most practical isomorphism testing algorithms in polynomial time [6], [7], [8], [9], [16], [19].

Most of the works available in the literature have been focused on finding practical isomorphism testing algorithms that solve the general problem in polynomial time. As a result, algorithms such as the *nauty* package of Brendan McKay have been developed [8], [9]. It has been regarded as the fastest isomorphism testing program until recently, when algorithms offering a better performance such as *bliss* of Tommi Juntilla and Petteri Kashi, *saucy* of Martin Kutz and Pascal Schweitzer, *sinauto* and *conauto* of José Luis López Presa, *Traces* of Aldofo Piperno, have been developed [6], [16], [19], and [24]. Nevertheless, few of the main reasons behind the impressive performance that *nauty* has been offering are: 1) the reduction of the graph isomorphism problem to the problem of finding a canonical label for each of the graph being compared, 2) the use of automorphisms found while searching for a certificate to prune the search tree, and 3) the use of invariants. In fact, algorithms such as Ullman, SD, VF and VF2, which are based

on a depth first backtrack strategy, are not comparable to *nauty* for the general case of the problem.

On the other hand, some researchers, including Fortin and Miyazaki, tried to find hard graphs for the isomorphism problem [7], [15]. Depending on the application of the graph isomorphism problem, one would like to find classes of graphs that make isomorphism testing more complex for most practical algorithms. These hard graphs range from the Miyazaki's constructions to projective planes [15], [19], [24].

CHAPTER 3

GRAPH ISOMORPHISM BASED ZERO-KNOWLEDGE PROOFS

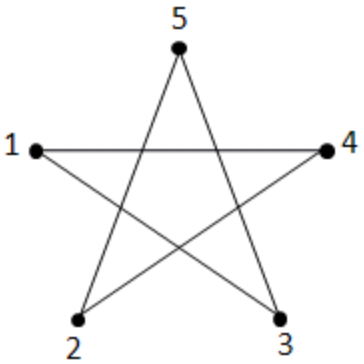
3.1 Graph

In this thesis, a graph is a set of nodes or vertices *V*, connected by a set of edges *E*. The sets of vertices and edges are finites. A graph with *n* vertices will have: *V* = {1, 2, 3,..., n} and *E* a 2-element subsets of *V*. Let u and v be two vertices of a graph. If $(u, v) \in E$, then u and v are said to be adjacent or neighbors.

A graph is represented by its adjacency matrix. For instance, a graph with *n* vertices, is represented by a $n \times n$ matrix $M = [m_{i,j}]$, where the entry $m_{i,j}$ is "1" if there is an edge linking the vertex *i* to the vertex *j*, and is "0" otherwise. For undirected graphs, the adjacency matrix is symmetric around the diagonal.

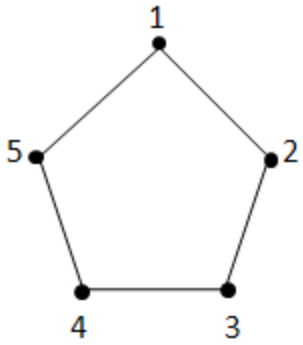
3.2 Graph Isomorphism

Two graphs G_1 and G_2 are said to be isomorphic, if a one-to-one permutation or mapping exists between the set of vertices of G_1 and the set of vertices of G_2, with the property that if two nodes of G_1 are adjacent, so are their images in G_2. The graph isomorphism problem is therefore the problem of determining whether two given graphs are isomorphic. In other words, it is the problem of determining if two graphs with different structures are the same. Figure 3.1 gives an example of isomorphic graphs, with their corresponding adjacency matrices. Notice that the entries of the matrices, where $m_{i,j} = 0$ are left blank.



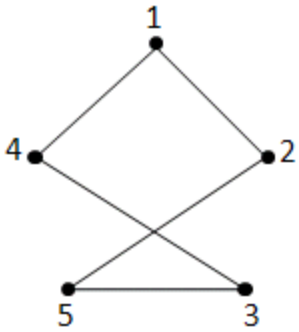
	1	2	3	4	5
1			1	1	
2				1	1
3	1				1
4	1	1			
5		1	1		

(a)



	1	2	3	4	5
1		1			1
2	1		1		
3		1		1	
4			1		1
5	1			1	

(b)



	1	2	3	4	5
1		1		1	
2	1				1
3				1	1
4	1		1		
5		1	1		

(c)

Figure 3.1: Example of isomorphic graphs with their corresponding adjacency matrices.

For instance, Graph (b) is obtained, by relabeling the vertices of Graph (a) according to the following permutation: (3, 5, 2, 4, 1). This means that Node 1 in Graph (a) becomes Node 3 in Graph (b), Node 5 becomes Node 1 and so on. Following in Figure 3.2 is an illustration of how the permutation is applied to Graph (a).

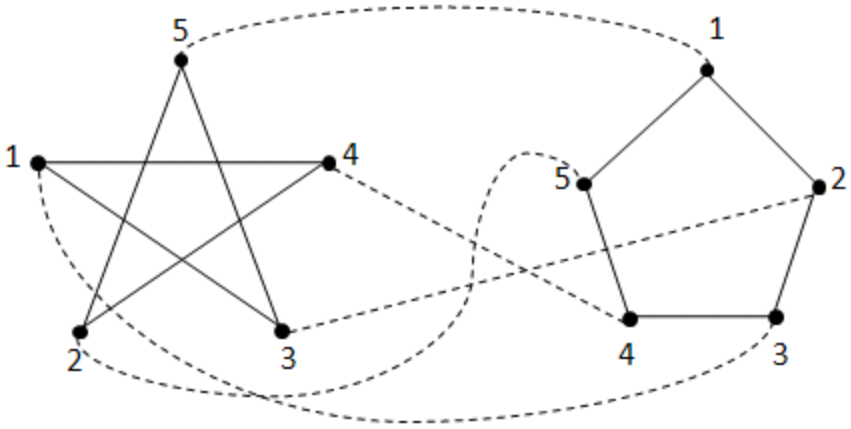


Figure 3.2: Relabeling the vertices of graph (a) using the permutation (3, 5, 2, 4, 1).

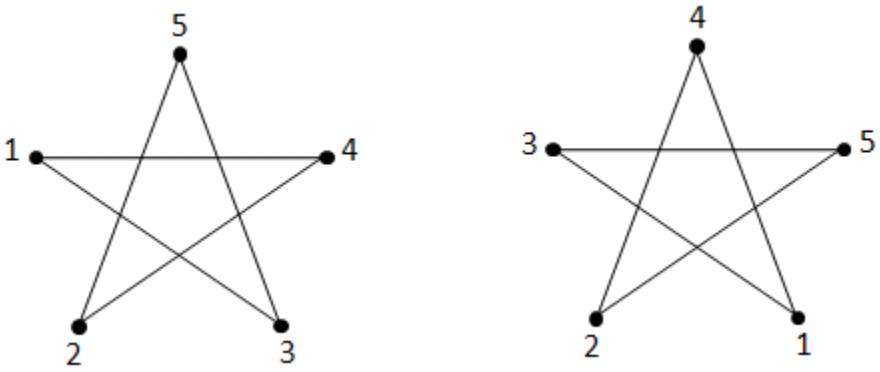
Even though it is not an easy task to devise a practical algorithm for the general case of isomorphism testing, there are some classes of graphs such as planar graphs and graphs with bounded genus for which efficient algorithms are known [13], [14]. On the other hand, for classes of graphs that are considered hard for the isomorphism problem, several practical graph isomorphism algorithms with exponential upper bound time complexity exist. However, the *nauty* package is known to be one of the most powerful algorithms currently available [6], [7], [8], [19]. Other algorithms, such as the ones presented in [6], [19], [24] that perform better than *nauty* for certain classes of graphs, are also being developed.

3.3 Graph Automorphism

An automorphism is an isomorphism of a graph into itself. For example *nauty* detects an automorphism, when after relabeling the vertices of a graph; two different leaf partitions (see definition in 3.6.2) provide the same adjacency matrix. The smallest of the

automorphs is used by *nauty* to compute the canonical label. Additionally, detecting the automorphs of a graph helps *nauty* in pruning the search tree.

As shown in Figure 3.3, the permutation (3, 2, 1, 5, 4) on Graph (a) induces an automorphism.



	1	**2**	**3**	**4**	**5**
1			1	1	
2				1	1
3	1				1
4	1	1			
5		1	1		

	3	**2**	**1**	**5**	**4**
3			1	1	
2				1	1
1	1				1
5	1	1			
4		1	1		

Figure 3.3: Example of automorphic graphs with their corresponding adjacency matrices.

3.4 Graph Isomorphism based Zero-Knowledge Proofs

Zero-knowledge proofs (ZKP) systems can be implemented in several ways. Yet, in this thesis, I am focusing on zero-knowledge proofs based on graph isomorphism, where the verifier is bounded to a certain number of interactions, after which he/she will grant or deny access to the prover.

Suppose there are two graphs G_1 and G_2, such that the graph G_2 is generated by relabeling the vertices of G_1 according to a secret permutation π while preserving the edges. The pair of graphs G_1 and G_2 forms the public key pair, and the permutation π serves as the private key. A third graph H, which is either obtained from G_1 or G_2 using another random permutation, say ρ is sent to the verifier who will in return challenge the prover to provide the permutation σ which can map H back to either G_1 or G_2.

For instance, if H is obtained from G_1 and the verifier challenges the prover to map H to G_1, then $\sigma = \rho^{-1}$. Similarly, if H is obtained from G_2 and the verifier challenges the prover to map H to G_2, then $\sigma = \rho^{-1}$. On the other hand, if H is obtained from G_1 and the verifier challenges the prover to provide the permutation that maps H to G_2, then $\sigma = \rho^{-1} \circ \pi$, which is a combination of ρ^{-1} and π. In fact, ρ^{-1} will be applied to H to obtain G_1 then the vertices of G_1 will be modified according to the secret permutation π to get G_2. Finally, if H is obtained from G_2 and the verifier challenges the prover to map H to G_1, then $\sigma = \rho^{-1} \circ \pi^{-1}$.

One can notice that in the first two cases, the secret permutation π is not even used. Therefore, a verifier could only be certain of a node's identity after many interactions. Moreover, we can also observe that during the whole interaction process, no clue was given about the secret itself, hence the name zero-knowledge proof.

Given G_1 and G_2 such that $G_2 = \pi(G_1)$, the interactions constituting a round of the graph isomorphism based ZKP protocol are illustrated as follows:

1. Prover chooses randomly $a \in \{1, 2\}$
2. Prover chooses a random permutation ρ, and generates $H = \rho(G_a)$.

3. Prover sends the adjacency matrix of *H* to the verifier.
4. Verifier sends $b \in \{1, 2\}$ to the prover and challenges for σ which maps *H* to G_b.
5. If $a = b$ the prover sends $\sigma = \rho^{-1}$ to the verifier.
6. If $a = 1$ and $b = 2$ the prover sends $\sigma = \rho^{-1} \circ \pi$ to the verifier.
7. If $a = 2$ and $b = 1$ the prover sends $\sigma = \rho^{-1} \circ \pi^{-1}$ to the verifier.
8. Verifier checks if $\sigma(H) = G_b$ and grants access to the prover accordingly.

Several rounds of these interactions are needed for the verifier to be completely convinced of the prover's identity, since the prover can be lucky and guess the value of *b* before sending *H*. However, the probability that this happens is $1/2^n$, with *n* being the number of rounds. Therefore, with several rounds, this probability is considerably low, and the level of confidence that the verifier will gain about the identity of the prover is $1 - 1/2^n$. For example after 10 rounds, the confidence level of the verifier is approximately 99.9 %.

Additionally, as a zero-knowledge protocol, the graph isomorphism based ZKP protocol must present the following properties: soundness, completeness and zero-knowledge. According to [4], a protocol is sound when the verifier follows the protocol and is always able to reject a proof if it is erroneous. On the other hand, a protocol is complete when both parties follow the protocol, and the prover is able to convince the verifier every time a proof is true. Finally, a protocol is zero-knowledge when the verifier does not learn anything but the validity of the proof provided by the prover.

3.5 Approaches to the Graph Isomorphism Problem

There are different approaches to the problem of finding isomorphisms of a graph, however most practical algorithms available in the literature are sub-divisible into two different categories. In fact, according to the author in [6], the algorithms in the first category proceed directly by taking the two graphs to be compared for isomorphism, and try to find a match between them. They proceed by using a depth-first backtrack algorithm and by using heuristics to reduce the size of the search tree.

On the other hand, the algorithms in the second category proceed by considering one graph at the time. They take a single graph, say G_1 and compute a function $C(G_1)$ which returns a certificate or a canonical label of the graph, such that for two graphs that are being compared (G_1 and G_2), $C(G_1) = C(G_2)$ if and only if G_1 and G_2 are isomorphic. After obtaining the canonical label for each graph the task of the algorithms is just to compare them.

These two classes of algorithms, even though they differ in the way they solve the isomorphism problem, make use of invariants.

3.5.1 Invariants

A graph in general has two types of invariant, which are: graph invariant and vertex invariant. A graph invariant is a function f such that, if applied to two graphs G_1 and G_2, that are isomorphic, $f(G_1) = f(G_2)$. However, $f(G_1) = f(G_2)$ does not necessarily mean that the graphs G_1 and G_2 are isomorphic. A graph invariant is therefore a necessary condition for isomorphism. Moreover, if an invariant is both necessary and sufficient for isomorphism, it is said to be complete [6]. Such a complete graph invariant

is called a certificate. Examples of graph invariants are the canonical label, the number of vertices, and the number of edges in a graph.

A vertex invariant on the other hand, is a function f on a vertex v, such that if there is an isomorphism between the vertices $v \in G_1$, and $v' \in G_2$, then $f(v) = f(v')$. A typical example of vertex invariant is the degree of a vertex (i.e. the number of adjacent vertices to that vertex). In fact, if a vertex v is isomorphic to a vertex v', they must have the same degree. However, if two vertices have the same degree, it is not necessarily true that they are isomorphic.

Furthermore, classifying the vertices of a graph is a method used by almost all graph isomorphism algorithms to prune the search space [8]. There are many more vertex invariants that have been proposed in the literature [6]. They are used either directly by the algorithm or in some cases, only at the request of the user. This is because some of the invariants require a longer computational time and using them might either increase or decrease the performance of the algorithm. As a matter of fact, *nauty* offers a number of invariants that can be applied, when requested by the user [9].

3.5.2 Canonical Label

Another way of defining a certificate for a graph is to consider its entire adjacency matrices, since each permutation of the *n* vertices of a graph defines a different adjacency matrix. The smallest *n* x *n*-bit number, obtained by concatenating the rows or columns of each possible adjacency matrix, is defined as the canonical label of the graph. However, computing such a certificate is known to be a NP-complete problem [6]. Therefore, in some cases, computing this certificate may even be harder than solving the isomorphism

problem. To avoid this problem, most isomorphism algorithms that use canonical labeling, try to generate only certain permutations, based on the structure of the graph.

3.5.3 Direct Backtracking Algorithms

Direct backtracking algorithms classify the nodes of the graphs according to some invariants, and use heuristic in order to prune the search tree. They mainly proceed, by exploring all the possible matching of the vertices of one graph against the vertices of the other graph and backtrack if a branch of the search tree does not provide a valid solution [6].

These algorithms are effective for some type of graphs but are more likely to have a high time complexity when the graphs being tested are highly symmetric. In fact these algorithms do not detect automorphisms (symmetries). Therefore, when the graphs are highly symmetric, they have to go around the whole search tree, exploring branches that are symmetrical to other branches that have already been explored, thus, increasing the time it takes to solve the isomorphism problem [6]. However, the advantage that they have is that once an isomorphism is found, they stop exploring the search tree.

3.5.4 Canonical Labeling Algorithms

Canonical labeling algorithms do not compare both graphs directly. Instead, they work independently on each graph. They generate the canonical labels of both graphs one at the time and then compare them directly. The most widely used of canonical labeling programs is *nauty* by Brendan McKay.

3.6 The *nauty* Program

Most canonical labeling programs such as *nauty* (*n*o *aut*omorphisms *y*es), use a backtracking algorithm that goes through the search tree in the aim of finding a canonical label, while building the automorphism group of the graph [6]. The *nauty* program for instance, starts with an initial partition, which is a classification of the vertices of the graph being tested based on their degree, or color for colored graphs. The initial partition, which is the root of the search tree, is then refined until a stable partition is reached. From the stable partition, which is a partition that cannot be refined further, a vertex individualization is done in order to generate a child partition and the refinement process restart. Once a discrete partition, where all the cells have size one is reached, a leaf or terminal coloring in the search tree is obtained. The leaf partitions induce different labeling of the graph, which in turn induce different candidate certificates. The smallest of such certificates, is the canonical label of the graph.

The nodes of the search tree, where a vertex has been individualized, are the backtracking points that will be used to find paths to other leaf partitions and therefore obtaining other canonical labels [6]. The first leaf partition is considered as inducing the best candidate for canonical labeling. From this leaf partition, *nauty* backtracks in order to find other terminal nodes, which will induce new labeling of the graph. If the new labeling leads to the same canonical label or the same adjacency matrix, an automorphism is detected and saved. This automorphism will be used later to eliminate part of the search tree from consideration. If the new labeling induces a smaller canonical label, it is chosen as the new candidate certificate. On the other hand, if a new branch is

known to induce a bigger certificate, it is discarded [6]. Finally, when all possible branches of the search tree have been fully examined or eliminated, the canonical label and the automorphism group are obtained.

The main advantage of this approach is the use of discovered automorphisms to prune the search tree. Therefore, for graphs having a large automorphism group, these classes of algorithms are very fast. However, computing the whole automorphism group might in some cases decrease their performance.

Several algorithms that have a better performance than *nauty* for certain classes of graphs are meanwhile being devised [6], [16], [19]. For example, the author in [6] devised an algorithm which is comparable to *nauty* in most cases and in some cases performs better. His algorithm uses the characteristics presented by the two classes of algorithms above described.

The next section discusses some of the characteristics that make this class of algorithms, especially the *nauty* program efficient for isomorphism testing. Understanding how *nauty* proceeds to solve the problem helps in the task of determining which graphs are hard for isomorphism testing.

3.6.1 *nauty*'s Invariants

The *nauty* program, in addition to using the degree of a vertex as an invariant, offers different type of vertex invariants such as two-paths, distances, and adjacencies, which can be applied at the user's request [8]. Two-paths are the number of vertices reachable along a path of length two, distances are the number of vertices reachable at

each distance, and adjacencies are to diminish the poor performance of *nauty* with digraphs [7], [8].

Depending on the types of graphs being tested, these invariants are used to prune the search tree. However, their usage, which is left to *nauty*'s users, requires a thorough understanding of how they are applied, the types of graphs that they are used for, and the levels of the search tree at which they can be used, without actually decreasing the performance of the program [19].

3.6.2 Partitions

A partition $\pi = V_1, \dots\dots, V_m,$ divides the vertices of a graph into non-empty subsets of V which are called cells. A discrete partition or leaf partition is one that has cells with only one vertice, namely trivial cells.

3.6.3 Operations on Partitions

In order to process to the refinement, *nauty* starts with an initial partition, where the vertices in each cell have the same degree. It then selects the first cell with more than one element namely the target cell, and computes a sequence a_v based on the adjacencies of the vertices in the target cell with the nodes in all the cells of the initial partition. Then, it uses the calculated sequence to split the target cell into a number of subsets, so that the vertices in each subset have the same value for a_v. This is illustrated by the following algorithm [7]:

1) Let $\pi = V_1, \ldots\ldots, V_m$ be the initial partition, where for all vertices $x, y \in V_i$ we have $d(x, V) = d(y, V)$.
2) Select a $V_i \in \pi$ such that V_i has more than one vertice.
3) For each vertice $v \in V_i$, compute a sequence $a_v = (d(v, V_1), \ldots, d(v, V_m))$.
4) Split V_i into a number of subsets, so that the vertices in each subset have the same value for a_v.

Another operation that *nauty* performs on partitions is individualization. It chooses the first cell, which has more than one element. Then, for each vertex v of that cell, it creates a child partition by splitting the chosen cell into two different cells, one containing only v and the other without v [7], [8].

3.6.4 Backtracking

In order to find the canonical label of a graph, *nauty* starts with the initial partition, which is based on the degrees or colors of the vertices and generates a tree namely search tree. The initial partition or root of the search tree is considered to be at level 0. Now, since the canonical label is generated from the smallest of the automorphs, *nauty* performs comparison on all possible leaf partitions of the search tree.

Starting from the initial partition, *nauty* determines the target cell which is most likely to contain the greatest number of vertices. Once such cell is selected, *nauty* process to individualization of each vertex of the cell in order to refine the partition. This is done until a leaf partition is reached. When a leaf partition is reached, its corresponding canonical label is computed and stored as a potential canonical label for the graph. The *nauty* program then backtracks to the parent of such partition at an upper level in the tree.

From this ancestor, another leaf partition is generated. The canonical label associated with the new leaf partition is computed and compared with the previous one. If the value is the same, an automorphism is discovered and saved. On the other hand, if this new value is greater than the previous, it is automatically discarded. If conversely the new value is better, it will substitute the old label as the new potential certificate.

Subsequently, the automorphism found is used to prune the search tree, since automorphs will produce the same canonical label. The whole process, which is illustrated in Figure 3.4, is repeated until a canonical label is obtained for the graph.

The example of Figure 3.4 is obtained from the *nauty* User's Guide (Version 2.2) [9]. In this example, the target cell is underlined, and the vertex being individualized is marked on the edges of the tree. The target cell here, which is on the left side of the tree, is {4, 5, 6}. Vertex 4 of the target cell is individualized to obtain the partition ({4}, {5, 6}, {2, 3}, {1}). The cell {5, 6} being the new target cell, Vertex 5 and Vertex 6 are individualized to produce two leaf partitions.

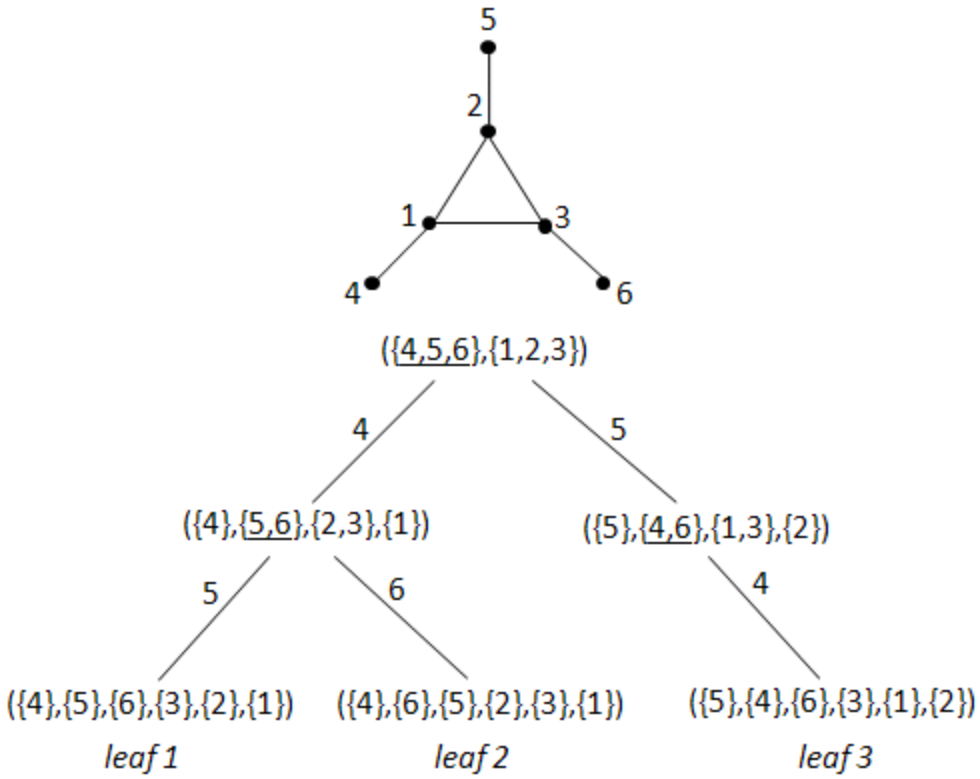
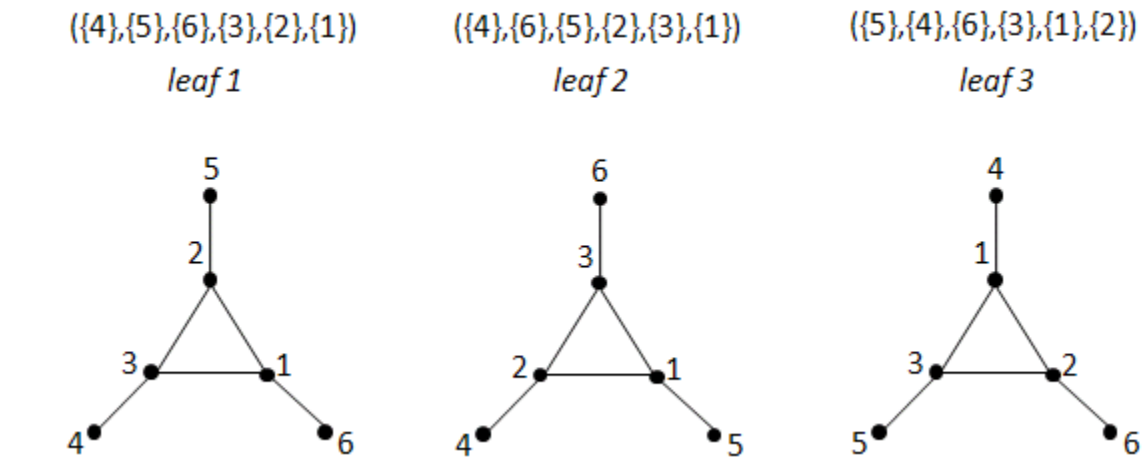


Figure 3.4: Example of search tree.

Comparing the canonical label associated with the two leaf partitions shows that an automorphism: (2, 3) (5, 6) is detected. This means that it is not necessary to individualized Vertex 6 of the target cell {4, 5, 6}, because no additional information is obtained by doing so. In other word, the same canonical label is obtained by individualizing Vertex 5 or 6. The *nauty* program has in this case been able to prune out part of the tree.

Another way to look at this process is to consider the adjacency matrices associated with each leaf partition. If the same adjacency matrix is obtained for different leaf partitions, we can assert that these partitions will induce the same canonical label. Thus using that information, *nauty* will be able to prune the search tree. Note that harder

the graph, larger is its search tree. Hence, from a performance standpoint, been able to eliminate parts of the search tree is very important. As shown in Figure 3.5, we can see that all the three leaf partitions are equivalent and the following automorphisms: (3, 1), (2, 3) (5, 6) and (3, 1, 2) (4, 5) have been detected.



	3	**2**	**1**	**4**	**5**	**6**
3		1	1	1		
2	1		1		1	
1	1	1				1
4	1					
5		1				
6			1			

	2	**3**	**1**	**4**	**6**	**5**
2		1	1	1		
3	1		1		1	
1	1	1				1
4	1					
6		1				
5			1			

	3	**1**	**2**	**5**	**4**	**6**
3		1	1	1		
1	1		1		1	
2	1	1				1
5	1					
4		1				
6			1			

Figure 3.5: Leaf partitions and their corresponding adjacency matrices.

CHAPTER 4

SUITABILITY OF GRAPH ISOMORPHISM FOR ZERO-KNOWLEDGE PROOFS

Before processing to the actual implementation of the graph isomorphism based zero-knowledge proofs (GIZKP) protocol, there are a number of questions that one could ask. Even though the graph isomorphism problem is not known to be solvable in polynomial time for the general case, there are several graphs for which polynomial algorithms exist. Therefore, a question that one might ask is: what types of graphs are suitable for the implementation of the GIZKP protocol?

On the other hand, since the GIZKP is an interactive protocol, one could be concerned with the problem of eavesdropping, cheating prover or verifier and the man-in-the-middle attack. In the following subsections, I discuss some of these problems and their plausible solutions.

4.1 Problems Faced by the GIZKP

In the implementation of the GIZKP protocol as aforementioned, the designer is faced with different problems. The first problem deals with cheating prover or verifier. Since the graphs G_1 and G_2 constitute the public keys, which all the parties involved in the interactions must have, envisage the case where the verifier or any other party possess a powerful algorithm such as the one used by the *nauty* program. He/she will then, in a considerable low time, depending on the type and the size of the graph, be able to detect the secret permutation π which has been use to generate G_2. Knowing such secret will allow him/her to impersonate the prover to the verifier or to a third party.

The second problem deals with the interception of graph *H*. During an interaction of the zero-knowledge protocol, the adjacency matrix of the randomly generated graph *H* is sent to the verifier by the prover. In the case an eavesdropper is listening to the conversation and is able to intercept the adjacency matrix of graph *H*, it would be easy for him to impersonate the prover if and only if, the graph used is not hard for the isomorphism problem or just as in the first case, he possess a powerful and practical algorithm.

In the above situations, the graph isomorphism based ZKP protocol cannot be considered as perfect. Therefore, even though the graph isomorphism problem is not known to be in P, for implementing the GIZKP protocol, I am interested in finding graphs that are very complex for the isomorphism problem. Since *nauty* is known to be one of the fastest practical and available algorithms, my goal is then to find graphs that are hard for *nauty* and evaluate their performance for the ZKP protocol.

Many researchers have worked on looking for efficient practical graph isomorphism algorithms. However, only few of them have focused their attention on finding hard graphs [7], [15], as this required a lot of knowledge in group theory. The search for hard graphs is indeed not an easy task [7], [16], but certain type of graphs such as trees, random graphs, planar graphs, graphs with bounded eigenvalue multiplicity, and graphs with bounded genus can be ruled out as finding their isomorphs is almost always easy [13], [14], [16].

4.2 Hard Graphs for Isomorphism Testing

A simple invariant that all practical graph isomorphism algorithms use when solving the isomorphism, or automorphism problem, is the degree of the vertices constituting the graphs. Therefore, in the search for hard graphs, one could start with regular graphs, which have the same degree for each of the vertices. In fact, according to the author in [16], [17], the performance of *nauty* or practical isomorphism tools in general starts degrading when dealing with graphs that have very few automorphisms, but a high degree of regularity.

In this thesis, I am not initiating the search for families of graphs, whose isomorphisms are hard to break. Instead, I am considering all the benchmarks families of graphs provided in the literature [6], [15], [16], [24]. In fact, these families of graphs include all the currently known hard instances for canonical labeling algorithms [19]. These graphs are as follows:

- Projective planes (PP)
- Cai-Fürer-Immerman construction
- Constraint satisfaction problems
- Hadamard matrices (Had)
- Miyazaki's constructions (Mz)
- Affine and projective geometries
- Random regular graphs (Rnd)
- Strongly regular graphs
- Grid graphs

Also, unions of graphs with similar structures are known to be very complex for the *nauty* program [6]. A simple description of most of the graphs above cited is given as follows:

4.3 Projective Planes

There are several types of projective planes in the literature. A finite projective plane of order k is defined as a set of $v = k^2 + k + 1$ nodes with the properties that: any two nodes determine a line, any two lines determine a node, every node has $k + 1$ lines on it, and finally every line contains $k + 1$ nodes [28]. The smallest finite projective plane, which is of order 2 is shown in Figure 4.1 below:

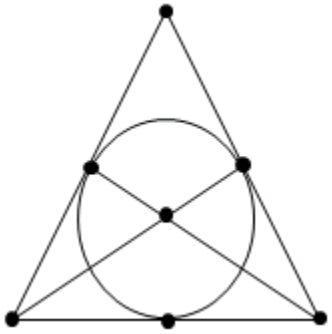


Figure 4.1: Finite projective plane of order 2.

However, the types of projective planes of interest are the ones known to be very complex for isomorphism testing. Some of these graphs, which are provided in [29] and used by the authors in [16] and [24], are the bipartite point-line incidence graphs of the known projective planes of order 16.

Additionally, the Desarguesian projective planes in [6] which were generated from the point-line incidence matrices, provided by Gordon Moorhouse [30], are also

known to be very hard instances for *nauty*. Figure 4.2 shows an example of pointline graph of the Desarguesian projective plane [6].

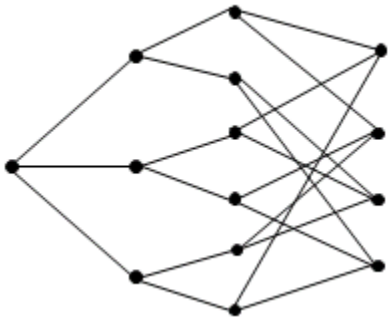


Figure 4.2: Point-line graph of the Desarguesian projective plane of order 2.

4.4 Random Regular Graphs

In general, according to the definition in [27], a random *r*-regular graph is a graph selected from the set $\mathcal{G}_{r-reg}$ of *r*-regular graphs. An *n* vertices *r*-regular graph is such that $3 \leq r < n$ and $nr = 2m$ is even. In this thesis, based on the benchmark presented in [24], only *n* vertices random 3-regular graphs constructed by rejection sampling are considered.

4.5 Strongly Regular Graphs

A regular graph with *n* vertices and degree *k* is said to be strongly regular if every two adjacent vertices have λ common neighbors and every two non-adjacent vertices have μ common neighbors. They are usually referred to as $srg(n, k, \lambda, \mu)$. In this thesis, I am only considering Paley graphs, triangular graphs, lattice graphs and Latin Square graphs due the fact that they are complex for most canonical labeling algorithms.

4.5.1Paley Graphs

Let q be an odd prime power such that: $q \equiv 1\ (mod\ 4)$. A Paley graph of order q is an undirected strongly regular graph with parameters $n = q,\ k = (q-1)/2,\ \lambda = (q-5)/4,$ and $\mu = (q-1)/4$. The vertices of a Paley graph belong to the finite field of order q ($\mathbb{F}_q$), and its edges are such that they connect pairs of vertices that differ in a quadratic residue. In fact, two vertices are adjacent when their difference is a square in the field [26]. For example the graph illustrated in Figure 4.3 is a Paley graph of order 5.

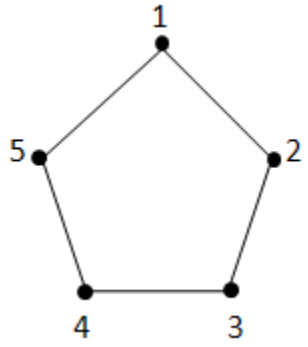


Figure 4.3: Paley graph (5, 2, 0, 1).

4.5.2 Triangular Graphs

A triangular graph T_q is a strongly regular graph with parameters $n = q(q-1)/2,\ k = 2(q-2),\ \lambda = q-2,$ and $\mu = 4$. Triangular graphs are vertex transitive and have large automorphism groups [6]. Figure 4.4 represents the triangular graph T_4.

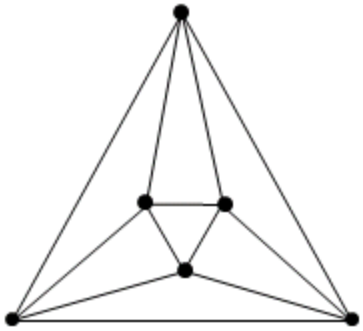


Figure 4.4: Triangular graph (6, 4, 2, 4).

4.5.3 Lattice Graphs

A lattice graph is considered as a type Latin square graph. Its vertices are the m^2 elements of a Latin square of order m and there is an edge between two vertices if and only if they are in the same row or column [6]. Lattice graphs are strongly regular graphs with parameters $n = m^2,\ k = 2(m-1),\ \lambda = m-2,$ and $\mu = 2$. They have a large automorphisms group, which make them complex for canonical labeling algorithms. An example of lattice graph where m = 3, can be seen in Figure 4.5 below.

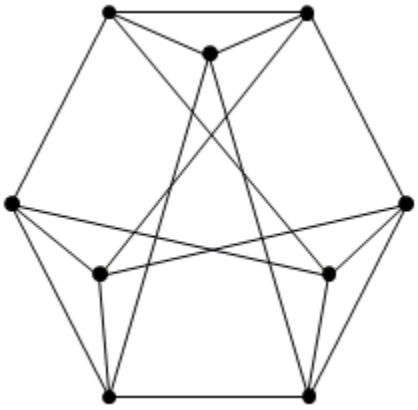


Figure 4.5: Lattice graph (9, 4, 1, 2).

4.5.4 Latin Square Graphs

As defined in [25], a Latin square of order *n* is an *n* x *n* matrix in which each of the n^2 cells contains a symbol from an alphabet of size *n*, such that each symbol in the alphabet occurs just once in each row and once in each column. The alphabet is completely arbitrary, but it is often convenient to take it to be the set {1, 2, …, n}.

4.6 Miyazaki's Constructions

In his paper "The complexity of McKay's canonical labeling algorithm," Miyazaki constructed regular graphs based on the Cai-Fürer-Innerman construction, which he proved to be very complex for *nauty*, therefore for canonical labeling algorithms in general. Miyazaki proved that a wrong choice of the target cell is responsible for the existence of intractable graphs for *nauty* [19]. Information on how these graphs are constructed can be found in [6] and [15]. Figure 4.6 shows an example of such graph.

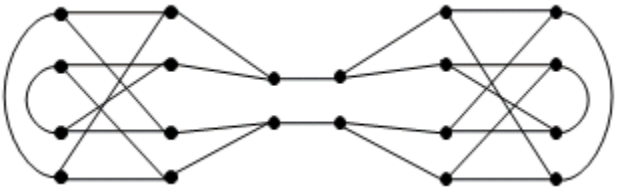


Figure 4.6: Example of Miyazaki's graph.

CHAPTER 5

EXPERIMENTS WITH SOME OF THE HARD GRAPHS

In this section, I conduct experiments with some of the hard graphs available in the literature in order to access their performance for the graph isomorphism based zero-knowledge proofs (GIZKP) protocol. Since the running time of canonical labeling of a graph is essentially related to the size of the associated search tree [15], [19], I present my results in term of the size of the search tree and the CPU time. Note that the harder the graph, the larger is its search tree.

These experiments were performed on a Dell Latitude XT2 with Intel(R) Core (TM) 2 Duo at 1.40 GHz with 3.45 GB of memory under Ubuntu 9.4. All the tests were performed using *nauty* (Version 2.2). The undirected graphs tested were generated using a program provided by Miyazaki [15]. The results of experiments on Miyazaki's Type B graphs are reported in Table 5.1, where we can see that up to 1000 nodes, *nauty* computed the canonical label in less than 4 seconds. However, starting from 1500 nodes graph, the computation time increases exponentially.

Using different permutations of the same graph, in [6], *nauty* was not able to handle a graph of 40 nodes in 10000 seconds. In fact, applying permutations on the vertices of a graph generates different versions of the graph that might be complex for the *nauty* program. The author in [7] for instance suggested that in order to increase the computational time of *nauty*, it suffices to swap the endpoints of two edges selected randomly.

This reduces the size of the automorphism group, therefore limiting the capability of *nauty* to prune the search tree.

Table 5.1: Experiments with Miyazaki's Type-B graphs.

Graph size	**Search tree size**	**CPU time (seconds)**
20	8	0.00
40	13	0.00
60	19	0.00
80	26	0.00
100	34	0.00
120	43	0.00
140	53	0.00
160	64	0.00
180	76	0.00
200	89	0.00
220	103	0.00
240	118	0.01
260	134	0.01
280	151	0.01
300	169	0.01
320	188	0.01
340	208	0.02
360	229	0.03
380	251	0.04
400	274	0.07
420	298	0.08
440	323	0.09
460	349	0.10
1000	1,429	3.64
1200	2,074	8.87
1500	4,000,698	37,135.31

On the other hand, comparing the results in Table 5.1 with the ones obtained by Miyazaki in [15], where he used a former version of *nauty* and a less powerful machine, I noticed a considerable gain in performance. For example a graph of 460 vertices took

nauty (Version 1.7) running on a Sun SPARCstation-1 computer, 994,427.10 seconds with 218,104,107 nodes for the search tree. The same graph took us 0.10 seconds with a search tree size of 349 nodes.

I have not conducted experiments on all the aforementioned hard graphs. However, results on the performance of *nauty* and other canonical labeling tools such as *bliss* and *Traces* for most of these graphs are available in [6], [16], [19], and [24]. Some of these results obtained in [19] are reported in Table 5.2 below. Mz-aug2 graphs are modified version of Miyazaki's constructions, Rnd-3-reg are 3 regular random graphs and Latin-sw is a modified version of a Latin graph.

These experiments clearly show that there are algorithms that perform better than *nauty* for some families of graphs. In fact, while in some cases, the *nauty* program took a large amount of time or has not been able to solve the problem in a predetermined amount of time, programs like *Traces* or *bliss* computed the automorphism group or the canonical label in few seconds. These results additionally prove that even for hard instances of isomorphism testing, there are algorithms that are capable of solving the problem in polynomial time.

I know that isomorphism testing algorithms available in the literature attempt to solve the general case of the problem in polynomial time, which is an even harder problem because the algorithms have to be able to test whether any pair of graphs is isomorphic. They may therefore be efficient for some families of graphs, while offering a poor performance for others.

Table 5.2: Experiments with hard graphs.

Graph	**Graph Size**	***nauty*(2.2)**	***bliss*(0.35)**	***Traces***
Had-52	208	10.73	0.50	0.15
Had-100	400	185.00	4.43	1.83
Had-184	736	8544.87	34.96	6.11
Had-232	928	29352.07	84.51	14.99
Had-236	944	Timed out	23150.37	103.36
Mz-18	360	0.20	0.01	0.64
Mz-50	1000	>24000	0.30	24.76
Mz-aug2-18	432	106.08	22.64	1.22
Mz-aug2-20	480	525.01	110.14	1.50
Mz-aug2-22	528	2500.61	439.06	2.06
Mz-aug2-30	720	Timed out	Timed out	5.47
Mz-aug2-50	1200	Timed out	Timed out	25.44
PP-16-2	546	42288.08	1868.28	2.51
PP-16-7	546	Timed out	4662.65	0.96
PP-16-9	546	15598.57	825.15	0.24
PP-16-21	546	Timed out	25871.22	2.79
PP-(flag-6)	1514	Timed out	26665.97	40.45
Rnd-3-reg-3000-1	3000	328.57	0.39	0.31
Rnd-3-reg-10000-1	10000	42821.01	5.60	4.24
Latin-sw-30-11	900	49.79	46.32	12.43

In contrast, devising specific polynomial time algorithms that solve the problem for specific families of graphs is a much easier task. It is in fact possible to develop algorithms that will exploit characteristics, such invariants and patterns of specific families of graphs in order to test for isomorphism in polynomial time.

Given these facts, and given the performances that practical algorithms are offering, it is evident that specific devised algorithms will perform better, which make the graph isomorphism not convenient for ZKP.

One could also argue that since the larger the graph, the longer a program will take to test for isomorphism, to opt for very large graphs in the implementation of the GIZKP protocol. This is not feasible, because in reality there is a limit on the size of the graphs, which is based on the capabilities of the infrastructures used. The memory requirement for each public key graph is for instance n^2 bits for n vertices graph. However, this memory requirement is reduced to $(n^2 - n)/2$ bits if testing undirected graphs. The private key in general requires n bytes.

Note also that even though these experiments were performed on average with graphs having only hundreds of nodes, testing graphs with thousands of vertices is possible in polynomial time.

CHAPTER 6
CONCLUSIONS

I have investigated the graph isomorphism based zero-knowledge proofs (ZKP) protocol. I surveyed the literature and reported the current methods for isomorphism testing. I enumerated some of the known hard graphs for the problem and tested some of them using one of the fastest isomorphism testing programs available, the *nauty* package. I also presented the results of tests performed by other researchers. The obtained results clearly show that most instances of the isomorphism problem are solvable in polynomial time. However, since these experiments were performed on tools devised for the general case of the problem, I conjecture that better results, obtainable in polynomial time for large graphs, are possible by using specific algorithms for specific families of graphs.

While there are major advantages to being able to test for isomorphism in polynomial time, this constitutes the main problem for the implementation of the graph isomorphism based ZKP protocol. Therefore, since it is possible to test for isomorphism in polynomial time, the graph isomorphism is not perfectly suitable for the implementation of a ZKP protocol.

Future works could include implementation of specific algorithms for each hard graph that will be able to solve the isomorphism problem in polynomial time. Another route to explore is to develop a timed graph isomorphism based ZKP, where the prover will be bound to a specific amount of time to provide the answers to the challenges of the verifier.

BIBLIOGRAPHY

[1] S. Goldwasser, S. Micali and C. Rackoff, "The knowledge complexity of interactive proof systems," in *Proc of STOC 1985*, pp. 291-304.

[2] O. Goldreich, S. Micali and O. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems", *Journal of the ACM*, vol. 38, no. 1, pp. 691–729, 1991.

[3] M. Bellare, S. Micali and R. Ostrovsky, "Perfect zero-knowledge in constant rounds," *Proc. 22nd STOC*, pp. 482-493, 1990.

[4] K. Namuduri, "An active trust model based on zero-knowledge proofs for airborne networks," *presented at the work shop on Cyber Security and Information Intelligence Research Workshop (CSIIR)*, Oak Ridge National Laboratory, April 13-15 2009.

[5] O. Goldreich and H. Krawczyk, "On the composition of zero-knowledge proof systems," *SIAM Journal on Computing*, vol. 25, pp.169-192, 1990.

[6] J. L. L. Presa, "Efficient algorithms for graph isomorphism testing," *Doctoral Thesis*, Madrid 2009.

[7] S. Fortin, "The graph isomorphism problem," *Technical Report TR 96-20*, July 1996.

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

[9] B. 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.

[10] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Graph matching applications in pattern recognition and image processing," in *IEEE International Conference on Image Processing*, vol. 2, pp. 21–24, September 2003.

[11] J-L. Faulon, "Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial–time for molecular graphs," *Journal of chemical information and computer science*, vol. 38, pp. 432–444, 1998.

[12] J. Buhler, P. Wocjan, "Quantum approaches to the graph isomorphism problem," Institute for Quantum Information, California Institute of technology, Pasadena, CA, 2006.

[13] J. E. Hopcroft and J. K. Wong, "Linear time algorithm for isomorphism of planar graphs (preliminary report)," in *Proc of the 6th Annual ACM Symposium on Theory of computing*, pp. 172–184, 1974.

[14] J. Chen,"A linear time algorithm for isomorphism of graphs of bounded average genus source," in *Proc of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, vol. 657, pp.103-113, 1992.

[15] T. Miyazaki, "The complexity of McKay's canonical labeling algorithm," *28, Amer. Math. Soc*, pp. 239-256, 1997.

[16] M. Kutz and Pascal Schweitzer, “ScrewBox: a randomized certifying graph-non-isomorphism algorithm,” ALENEX 2007.

[17] W. Kocay, “On writing isomorphism programs,” in *Computational and Constructive*. Design Theory, pp. 135–175, 1996.

[18] J-J. Quisquater, L. Guillou, T. Berson, “How to explain zero-knowledge protocols to your children,” *Advances in Cryptology - CRYPTO '89: Proceedings*, vol.435, pp. 628-631, 1990.

[19] A. Piperno, “Search space contraction in canonical labeling of graphs,” (Preliminary Version) CoRR abs/0804.4881, 2008.

[20] Zero-knowledge proof. Available: http://en.wikipedia.org/wiki/Zero-knowledge_proof

[21] R. Read and D. Corneil, “The graph isomorphism disease,” *Journal of Graph Theory*, vol.1, pp. 339–363, 1977.

[22] U. Schöning, “Graph isomorphism is in the low hierarchy,” in *Proc. of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 114–124, 1987.

[23] P. Foggia, C. Sansone, and M. Vento, “A performance comparison of five algorithms for graph isomorphism,” in *Proc. of the 3rd IAPR-TC15 Workshop on Graph-based Representations*, pp. 188–199, 2001.

[24] T. Junttila and P. Kaski, “Engineering an efficient canonical labeling tool for large and sparse graphs,” in *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX07)*, SIAM, 2007.

[25] Latin square, "The encyclopedia of design theory." Available: http://designtheory.org/library/encyc/topics/lsee.pdf.

[26] Paley graph. Available: http://www.win.tue.nl/~aeb/drg/graphs/Paley.html.

[27] Béla Bollobás, "Random graphs - section 2.4: Random Regular Graphs," 2nd ed. Cambridge University Press, 2001.

[28] Projective plane. Available: http://mathworld.wolfram.com/ProjectivePlane.html

[29] A library of projective planes of order 16 maintained by G. Royle. Available: http://www.csse.uwa.edu.au/~gordon/remote/planes16/index.html

[30] G. E. Moorhouse, "Projective planes of small order," Dept. of Mathematics, Univ. of Wyoming, 2005. Available: http://www.uwyo.edu/moorhouse/pub/planes/.