ALGEBRAIC APPROACH TO

GRAPH ISOMORPHISM

by

Shahzad Choudhry

B.Sc., University of Colorado at Denver, 2000

A-thesis submitted to the
University of Colorado at Denver and Health Sciences Center
in partial fulfillment
of the requirements for the degree of
Master of Science
Computer Science
2007
This thesis for the Masters of Science degree by Shahzad Choudhry has been approved by

Tom Altman

Boris Stilman

Hyunkeun Ra

4/23/07
Date
ABSTRACT

The subject of this thesis is graph isomorphism. There are many ways to draw the same graph. If the graph is complex, with a little imagination, one can see that deciding whether two drawings represent the same graph can become a challenging task. This, however, is not graph isomorphism. The word iso indicates similarity and the word morph indicates change. The graph isomorphism problem is one of deciding whether two different graphs are similar in the sense of structural equivalence. That is whether they have the same pattern of connections. To this day, no practical algorithm has been designed to solve this problem for all cases. A Graph Isomorphism $f$ from a graph $G$ to a graph $H$ is a one-to-one mapping for each node in $G$ to a node in $H$ such that for any two nodes $u$ and $v$ in $G$ there are the same number of edges between $f(u)$ and $f(v)$ as there are between $u$ and $v$ in $G$. Two graphs $G$ and $H$ are isomorphic, if and only if, there exists a vertex bijection $f: V_G \rightarrow V_H$ that
preserves edge connectivity. If you can re-label the nodes of one graph to match those of another graph, preserving edge multiplicity, you have two isomorphic graphs. This thesis provides a detailed specification of the graph isomorphism problem along with a discussion of algorithms for its solution. The major portion of the thesis work performed was the design and implementation of a computer program to check if two graphs are isomorphic. This report also provides the design documentation, testing, and pseudo-code for this program, which uses Altman's Marking Algorithm [3].

This abstract accurately represents the content of the candidate's thesis. I recommend its publication.

Signed

Tom Altman
CONTENTS

Figures................................................................................................................... vii

Tables........................................................................................................ viii

Chapter

1. Introduction........................................................................................................... 1
1.1 Graph Isomorphism Specification....................................................................... 1
1.2 Graph Isomorphism Algorithms......................................................................... 7
1.2.1 Vertex Filtering (VF) Algorithm.................................................................... 12
1.2.2 McKay's Nauty Algorithm.............................................................................. 15
1.2.3 The Marking Algorithm.................................................................................. 16
1.2.4 The Projection Algorithm.............................................................................. 18
2. Developer Requirements....................................................................................... 21
3. High-Level Design............................................................................................... 25
4. Detailed Design..................................................................................................... 30
4.1 Program Hardware and Software Specifications............................................ 30
4.2 Code Files, Classes, and Data Structures....................................................... 31
4.3 The Marking Algorithm..................................................................................... 34
4.4 The LR-Cholesky Algorithm............................................................................. 38
5. Testing and Analysis............................................................................................. 41
6. Conclusion ........................................................................................................ 48

References ........................................................................................................... 51
LIST OF FIGURES

Figure

1.1 Example Graphs ........................................................................................................ 1
1.2 Example Graphs ...................................................................................................... 4
1.3 Example Graphs ...................................................................................................... 5
3.1 Example Graph Input File ....................................................................................... 29
LIST OF TABLES

Table

5.1 Test Data ........................................................................................................................................42
1. Introduction

This introduction will provide a detailed specification of Graph Isomorphism (GI) followed by a discussion of algorithms to solve the problem.

1.1 Graph Isomorphism Specification

It is not difficult to see that there are an infinite number of ways to draw the same graph. If the graph is a little complex, with a little imagination, one can see that deciding whether two drawings represent the same graph can become a challenging task. This, however, is not graph isomorphism. The word *iso* indicates similarity and the word *morph* indicates change. The graph isomorphism problem is one of deciding whether two different graphs are similar in the sense of structural equivalence. That is whether they have the same pattern of connections. To this day, no practical algorithm has been designed to solve this problem for all cases. With that said, let us move on to further specification and explanation of the problem.

Consider the following two graphs, G and H.

![Graphs G and H](image)

**Fig 1.1 Example Graphs**
It is easy to see, because of the way the graph is drawn, that by renaming the vertices, from graph H, we can get graph G. That is 9→1, 2→4, 4→2, and 5→3 transforms graph H to graph G. Note that this renaming or transformation in turn, causes the edges to change endpoints, that is edge (9, 2) in H now becomes edge (1, 4), as in G. Each renaming, of edges or nodes, is known as a bijection. If a set of bijections exists to transform a graph to another graph, the two graphs are isomorphic. However, finding these bijections can become very difficult when dealing with more complex graphs. Consider the following graph specifications:

**Graph G** has 4 nodes with edges (4, 1), (4, 2), and (3, 4).

**Graph H** has 4 nodes with edges (9, 2), (4, 2), and (2, 5).

Yes, these are the same graphs G and H as shown in the diagrams before. Since the graphs are undirected, the edge \((x, y)\) is the same as edge \((y, x)\). Now, without the aid of the diagram, we can see that it is not that straightforward to come up with the bijections to transform G to H or H to G.

In the graphs above, the edge bijections are a direct result of node bijections. The node bijections are sufficient to show the transformation, and thus show that graphs G and H are isomorphic. The graphs are undirected, but what if the graphs were digraphs, where \((x, y)\) and \((y, x)\) are not the same edge? Then we can simply take the direction of the edges into account when looking for isomorphism. What if the edges were labeled and/or assigned weights, as in a weighted graph or weighted digraph? In this case, we simply say that the weight on an edge is not really part of
the structure of the graph and is generally ignored in graph isomorphism. However, in multi-graphs and multi-digraphs, if edge multiplicity is not maintained, edge bijections are needed. That is, if an isomorphism, \( f \), on a graph \( H \), maps a node \( v \) to node \( f(v) \) and a node \( u \) to node \( f(u) \); \( f(v) \) and \( f(u) \) must have the same number of edges between each other as do the original nodes \( v \) and \( u \) in \( H \). If this is not the case, then we are changing the structure of the original graph \( H \), which is defined by its edges and which edges connect which nodes. In a naïve algorithm, for the general case, it may require \( n! \) vertex bijections to show that one graph can be transformed into another, where \( n \) is the number of nodes.

Now, let us formalize a definition of graph isomorphism. A graph isomorphism \( f \) from a graph \( G \) to a graph \( H \) is a one-to-one mapping for each node in \( G \) to a node in \( H \) such that for any two nodes \( u \) and \( v \) in \( G \) there are the same number of edges between \( f(u) \) and \( f(v) \) as there are between \( u \) and \( v \) in \( G \). Two graphs \( G \) and \( H \) are isomorphic, if and only if, there exists a vertex bijection \( f: V_G \rightarrow V_H \) that preserves edge multiplicity. If you can re-label the nodes of one graph to match those of another graph, preserving edge multiplicity, you have two isomorphic graphs. In other words, \( G \) and \( H \) are isomorphic if their vertices can be labeled or numbered so that adjacency matrices are equal. With regard to matrices, the problem for graphs \( G \) and \( H \) can be boiled down to: does there exist a permutation matrix \( P \) such that \( P^T G P = H \)? Of course, adjacency must be preserved, to avoid changing the structure of the graph. A vertex bijection \( f: V_G \rightarrow V_H \) preserves adjacency if \( f(u) \) is adjacent to \( f(v) \) if
and only if \( u \) is adjacent to \( v \), for all \( u, v \) in \( G \). For simple graphs, which do not include multi-graphs or multi-digraphs, preserving adjacency is the same as preserving edge multiplicity (as given in the definition), since there is, at most, only one edge between any pair of vertices. In the case of digraphs, the edge direction must also be preserved.

Now that we know what graph isomorphism is, we can see that in order for two graphs to be isomorphic, they must have the same number of nodes. This is true, because each node in one graph has to be mapped by the isomorphism to a node in the other graph and because this is a one-to-one mapping. What about the number of edges? If the number of nodes and the number of edges are the same amongst two simple non-multi graphs, are the graphs isomorphic? See the example below.

![Example Graphs](image)

**Fig 1.2 Example Graphs**

In this example, you can easily see that the structure of the graphs is different even though they have the same number of nodes and edges. The graphs are said to be of different *isomorphism type*. Graphs of the same isomorphism type are isomorphic. Isomorphic type, however, only seems to make sense if we are talking about graphs with a fixed number of nodes. On the other hand, we could abstract graphs to isomorphic types, in which case, all graphs with the same isomorphic type can be
considered the same graph. Thus, the number of nodes and the number of edges being the same for two graphs $G$ and $H$, is necessary, but not sufficient, to show isomorphism. Similar is the case for edge multiplicity preservation in an isomorphism. If $f$ is an isomorphism from graph $G$ to graph $H$, then node $v$ in $G$ for all nodes in $G$ and $f(v)$ in $H$ must have the same degree. Otherwise, the structure of the graphs $G$ and $H$ is different and they cannot be isomorphic. See the following example.

![Example Graphs](image)

Fig 1.3 Example Graphs

The two graphs have the same number of nodes, edges, and the same degree sequence of $<1, 1, 1, 2, 2, 3>$. However, they are not isomorphic because of the structural difference. Such properties of graphs are known as isomorphism invariants. These invariants cannot be different between two isomorphic graphs. Thus, they can be used to show non-isomorphism, but are not sufficient in themselves to show isomorphism.
A common goal with graph isomorphism is to create a practical general algorithm to decide graph isomorphism, or to show that no such algorithm exists. In a brute-force approach there are $n!$ vertex bijections to check and each check would require examining all vertex pairs, where $n$ is the number of nodes. There are many uses of graphs. Similarly, as isomorphism is a property of graphs, it also has importance for the same uses. In particular, there are uses in linguistics with context-free grammars, in circuits, in pattern recognition, in low-level image representations, in chemistry with chemical compounds, etc.

As mentioned earlier, no practical algorithm is known to exist that solves the general graph isomorphism problem in polynomial time. In fact, it is not even known which classification the problem falls into, whether it is NP-Complete, or whether it falls into the $P$ class and can be decided in polynomial time. The problem is, however, in NP, due to the ties to the permutation of vertices. There are also other types of classifications, but it is still not known whether graph isomorphism falls into them or not. In either case, it is considered a difficult problem to solve efficiently. However, by placing certain restrictions on the type of graphs, polynomial time algorithms have been devised. Such restrictions include bounded Eigen-value multiplicity, planar graphs, graphs satisfying a fixed degree bound, interval graphs, permutation graphs, convex graphs, and more. A more general problem, the subgraph isomorphism problem, is known to be NP-Complete. A new class, GI, the set of problems with a polynomial-time Turing reduction to the graph isomorphism
problem, was also created. The problem is trivially a complete problem for GI [14].

The rest of this thesis is dedicated to algorithms for graph isomorphism. Graph isomorphism is perhaps the most widely studied natural problem whose complexity is not classified as complete for some natural class. In 1852, Francis Guthrie posed the Four Color Problem for coloring a map. This problem asks if it is possible to color, using only four colors, any map of countries in such a way as to prevent two bordering countries from having the same color [11]. This problem was solved in 1976 by Kenneth Appel and Wolfgang Haken. This can be considered the birth of graph theory. While trying to solve it, mathematicians invented many fundamental graph theoretic terms and concepts, and graph isomorphism was born [11].

1.2 Graph Isomorphism Algorithms

Efficient graph matching algorithms suited for matching large graphs have been a subject of research for approximately the last three and a half decades. As mentioned in the previous section, most algorithms reduce the complexity by imposing restrictions on the input graphs. An alternative approach to reducing matching complexity is that of using an adequate representation of the searching process and pruning unprofitable paths in the search space. This is a technique most common in Artificial Intelligence. In this way, no restrictions must be imposed on the structure of the input graphs and the obtained algorithms can be generally applied. However, this process can become a difficult problem to solve.
One of the pioneer papers ascribed to this area of study, was published in 1970 by D.G. Corneil, C.C. Gotlieb [7]. It illustrates an isomorphism algorithm, which performs suitable transformations on the input graphs, in order to find a different representation for which the matching is computationally more convenient. However, it has been shown that the conjecture on which this method is based, is not always true.

A procedure that significantly reduces the size of the search space is the backtracking algorithm proposed by Ullmann in his 1976 paper [8]. This algorithm is devised for both graph isomorphism and sub-graph isomorphism, and is still one of the most commonly used today, for exact graph matching. Compared with other algorithms, it is more convenient in terms of matching time, in case of one-to-one matching.

Another backtracking algorithm, SD, was also presented in 1976 by D.C. Schmidt and L.E. Druffel [9]. It uses the information contained in the distance matrix representation of a graph to establish an initial partition of the graph nodes. This distance matrix information is then used in a backtracking procedure to reduce the search tree of possible mappings.

A more recent algorithm, known as VF for Vector Filtering, is based on a depth-first search strategy, with a set of rules to efficiently prune the search tree. This algorithm can be found in the 1999 paper by L. P. Cordella, P. Foggia, C. Sansone,
and M. Vento [10]. In 2000, an improvement was made to this VF algorithm, by the same authors, giving rise to the VF2, which always outperforms the VF [11].

The McKay's Nauty algorithm was developed in 1981 by B.D. McKay [12]. Even if Nauty is considered to be one of the fastest graph isomorphism algorithms available, it has been shown that there are categories of graphs for which it employs exponential time in order to find an isomorphism. It is based on a set of transformations that reduce the graphs to be matched to a canonical form for which it is very simple to test for the presence of an isomorphism.

Another possible approach to the isomorphism problem is the one presented in 1995 by H. Bunke and B.T. Messmer [13]. Instead of reducing the complexity of matching two graphs, the authors attempt to reduce the overall computational cost when matching a sample graph against a large set of prototypes. The method performs the matching in quadratic time with the size of the input graph and independently of the number of prototypes. It is obviously convenient in applications requiring matching a graph against a database, but the memory required to store the pre-processed database grows exponentially with the size of the graphs, making the method suitable only for small graphs. So, in case of one-to-one matching, other algorithms, in particular Ullmann's, are more suitable.

All the above cited algorithms are exact ones, meaning they find the exact solution to the matching problem, if any. Other techniques, like those based on non-deterministic paradigms, able to find approximate solutions to the matching problem
have been proposed, especially in the recent past. In most cases, they are very powerful in reducing the complexity from exponential to polynomial, but they do not guarantee finding an exact and optimal solution. As mentioned before, another category of algorithms for this problem places restrictions on the input graphs. Such restrictions include bounded Eigen-value multiplicity, planar graphs, graphs satisfying a fixed degree bound, interval graphs, permutation graphs, convex graphs, and more. These algorithms are also polynomial in running time. However, unlike approximation algorithms, these algorithms are important and practical, as given problem definitions may very well meet, or can be modified to meet the restrictions with little consequence.

Altman's paper [3] gives two heuristic backtracking algorithms with polynomial running times, if no backtracking is applied. The Marking Algorithm is based on the Principal Axis Theorem and a specific marking of diagonal elements of the adjacency matrices. The Projection Algorithm modifies the other, incorporating a heuristic that uses at most $n$ steepest descent projections. Both algorithms rely on the calculation of Eigen-values of the adjacency matrices, which are forced to be symmetric, positive definite matrices to ensure that the Eigen-value algorithm(s) converge.

Thus far, we have discussed different classifications of algorithms, and given a brief overview of several effective and well-known algorithms. For each algorithm, the source papers were referenced, and can be studied for more detailed specifications of these algorithms. However, we will now discuss four of these algorithms in more
detail, along with their performance. The four algorithms that will be discussed below are Vertex Filtering (VF) algorithm, McKay's Nauty algorithm, Altman's Marking algorithm, and Altman's Projection algorithm.

Through research, one useful paper comparing the performance of the five general GI algorithms discussed above was found. The paper was published fairly recently, by P. Foggia, C. Sansone, and M. Vento [4]. As it could be expected, the research in this paper showed that none of the algorithms is definitively better than the others. In particular, for randomly connected graphs, the Nauty algorithm is better if the graphs are dense and/or of large size. On the contrary, for smaller, sparse graphs, VF2 performs better. On more regular graphs, i.e. on 2D meshes, VF2 is definitely the best algorithm, since the Nauty algorithm is not even able to find a solution for graphs bigger than a few dozens of nodes. In the case of bounded valence graphs, if the valence is small, VF2 is always the best algorithm; while for bigger values of the valence, if the size of the graphs is small, the Nauty algorithm is more convenient. Finally, it is also worth noting that SD, VF, and VF2, are the only algorithms that were always able to find a solution to the isomorphism problem, independent of the size and kind of graphs to be matched.

It seems that it is a tie between the performance of the VF2 (based on the VF) and the Nauty algorithm. However, depending on the type of graphs and valence, one would be better suited to a particular situation than the other. As these two algorithms have proven themselves, let us take a closer look at them.
1.2.1 Vertex Filtering (VF) Algorithm

There are actually two main VF algorithms for graph and sub-graph isomorphism common today, the VF and VF2. Performance evaluations done by the authors of VF show major increases in speed compared to the algorithm written by Ullmann [8]. The VF2 algorithm reduces memory usage from $O(n^2)$ to $O(n)$ and has been found to always outperform its older version, the VF algorithm. The VF2 algorithm was published by the same authors in 2000, about one year after its predecessor [10] [11].

Both VF algorithms are implemented as backtracking match functions that apply recursive function calls for every added mapping between the pattern and the target graph. The recursive programming style makes it possible to backtrack simply by returning from the current invocation of the matching function. In order to keep track of which nodes have been mapped in each iteration, between the two input graphs, a state object is created during each iteration. The state object is cloned from the previous state and extended with a new mapping during each recursion. This means that every state object contains all the mappings found in the previous state object, in addition to the currently added one. This way the state object always includes a partial, valid isomorphism. The recursion continues until all the nodes and edges of the pattern graph have been matched. If at some point there are no more possible nodes in the target to match, even though the whole pattern graph has not yet
been matched, it simply returns to the previous level and tries the next pair. This continues until the whole search space has been covered. The VF algorithms do not have an exact strategy for choosing the search plan, instead it is dynamically determined during the search. This is done so that the next node is always selected from the adjacent nodes of the most recently added node. Choosing the path in this manner makes it a depth-first search. When the search finally reaches a node from which there are no adjacent nodes, then the unmapped nodes, which are adjacent to previously mapped nodes, will be matched. Below you will find the VF2 algorithm.

M is said to be a graph/sub-graph isomorphism if and only if M is an isomorphism between G2 and a sub-graph of G1. We will assume that the graphs involved are directed graphs, i.e. an arc \((i, j)\) is to be considered different from \((j, i)\). However, the extension of the algorithm to undirected graphs is trivial. The matching process can be suitably described by means of a State Space Representation (SSR). Each state \(s\) of the matching process can be associated to a partial mapping solution \(M(s)\), which contains only a subset of the components of the mapping function \(M\). A partial mapping solution univocally identifies two sub-graphs of \(G1\) and \(G2\), say \(G1(s)\) and \(G2(s)\), obtained by selecting from \(G1\) and \(G2\) only the nodes included in the components of \(M(s)\), and the branches connecting them. In the following, we will denote by \(M1(s)\) and \(M2(s)\) the projection of \(M(s)\) onto \(N1\) and \(N2\), respectively, while the sets of the branches of \(G1(s)\) and \(G2(s)\) will be denoted by \(B1(s)\) and \(B2(s)\), respectively.
A high-level description of the VF2 matching algorithm can be outlined at this point:

**PROCEDURE** Match(s)

**INPUT:** an intermediate state s; the initial state s₀ has M(s₀) = NULL

**OUTPUT:** the mappings between the two graphs

IF M(s) covers all the nodes of G₂ THEN

OUTPUT M(s)

ELSE

Compute set P(s) of the pairs candidate for inclusion in M(s)

FOREACH (n, m) member of P(s)

IF F(s, n, m) THEN

Compute state s ’ by adding (n, m) to M(s)

CALL Match(s ’)

END IF

END FOREACH

Restore data structures

END IF

END PROCEDURE

Here F(s, n, m) is a boolean function (called feasibility function) that is used to prune the search tree. If its value is true, it is guaranteed that the state s’ obtained by adding (n, m) to s, is a partial isomorphism if s is; hence the final state is either an isomorphism between G₁ and G₂, or a graph/sub-graph isomorphism between a sub-graph of G₁ and G₂. Moreover, F will also prune some states that, albeit corresponding to an isomorphism between G₁(s) and G₂(s), would not lead to a complete matching solution.

Once again, we reiterate that there is no efficient general graph isomorphism algorithm. In the worst cases, this algorithm performs the same as the brute-force
naïve algorithm. That is, it has a $O(n^m)$ exponential running time, where $n$ and $m$ are the number of nodes in the input graphs. However, in most cases it outperforms any other available algorithm, and requires close to $O(n)$ space; and thus it can handle larger graphs more easily than most other algorithms.

1.2.1 McKay's Nauty Algorithm

Despite the fact that this algorithm is advertised as the fastest general graph GI algorithm, it also has a worst-case $O(n^m)$ exponential running time, like the VF algorithm. However, in performance testing it was found that this algorithm outperforms any others when dealing with dense, small to medium size graphs. Except in the case of graphs known as 2-D meshes, where the algorithm was not able to find a solution, but this may just be due to an implementation error.

As mentioned before, the algorithm is based on a set of transformations that reduce the graphs to be matched, to a canonical form, for which it is very simple to test the presence of an isomorphism. There is no document which explains in detail how Nauty works. However, most of the important ideas were published in Brendan D. McKay's paper [12]. The algorithm finds the automorphism group of a vertex-colored graph. Nauty is also able to produce a canonically-labeled isomorph of the graph, to assist in isomorphism testing. An automorphism of a graph is a graph isomorphism with itself, i.e., a mapping from the vertices of the given graph, $G$, back to vertices of $G$, such that the resulting graph is isomorphic with $G$. The set of automorphisms for $G$ make up its automorphism group, which the algorithm uses as
an abstraction to look for isomorphism. You can also say that the algorithm is based on group theory. Some data structures, including trees, are used in the process. Coloring or partitioning is used, automorphisms of the graph are found by noticing that two labelings give the same labeled graph. The canonical labeling map corresponds to one of these labelings, chosen according to a complicated scheme.

This is just a high level view of some of the features of the algorithm. Despite my sincerest efforts, I was unable to find a good description or explanation of the algorithm. McKay's paper, which can be found online, contains a very long and difficult to understand description of the algorithm, which may also involve concept learning.

1.2.3 The Marking Algorithm

As mentioned earlier, this algorithm is based on the Principal Axis Theorem and a specific marking of the diagonal elements of the adjacency matrices. For every symmetric matrix $A$, there exists an orthogonal matrix $Z_A$ which transforms $A$ into a diagonal form $D_A = Z_A^T A Z_A$. The diagonal elements of $D_A$ are the Eigen-values of $A$. This is the Principal Axis Theorem, and it tells us that every symmetric matrix with $n$ rows and $n$ columns, has $n$ Eigen-values. Computation of Eigen-values is part of the algorithm. To ensure that the algorithm for computation of these values converges, it is necessary that only symmetric, positive definite matrices are used to represent the graphs. Matrices for undirected graphs are symmetric, and the algorithm can be used for simple undirected graphs. For the matrices to be positive definite, it is sufficient
that the matrices be *strongly diagonally dominant*, i.e., the sum of the absolute value of the non-diagonal entries must be less than the absolute value of the diagonal entry, for each diagonal entry in each row. This can be accomplished by setting the diagonal entries to \( n \), which will be zero for simple graphs with no self-loops. This will not change the structure of the permutation matrix \( P \), whose existence will show that the input graphs \( A \) and \( B \) are isomorphic.

The LR-Cholesky method for computing Eigen-values is recommended, since it is guaranteed to converge for the type of matrices described above with sufficient accuracy in polynomial time. The algorithm attempts to iteratively change or mark the diagonal entries of the adjacency matrices so that they remain isospectral or orthogonally congruent, while converging to an Eigen-value multiplicity of one. If successful, the matrices will be permutation similar, and thus isomorphic. With certain isospectral graphs, this algorithm will be forced to backtrack and try a different marking sequence, causing it to run in exponential time. Pairs of Isospectral Non-isomorphic Graphs or *PINGs*, where the graphs are orthogonally congruent, but not permutation similar, will cause this algorithm to backtrack. If the graphs are isomorphic, the algorithm marks their adjacency matrices in such a way as to wind up with orthogonally similar matrices, that is, they will have the same Eigen-values. However, whether the graphs are found to be isomorphic or not, the algorithm hopes that the graphs remain isospectral during the marking, otherwise backtracking will be
required. If the graphs are isomorphic and no backtracking is involved, the running
time of the algorithm is $O(n^4)$. Below is the pseudo-code for the algorithm.

```
Initialize ilast = jlast = 1
If Not SameEigenValues(A, B) then return "NOT ISOMORPHIC"
Else
    For i from ilast to n
        A[i, i] = 2in
        For j from jlast to n
            If B[j, j] = n then
                B[j, j] = 2in
                If SameEigenValues(A, B) then Exit j-loop
                Else If j = n then BACKTRACK if possible,
                       else return "NON ISOMORPHIC"
            Else
                B[j, j] = n
        End For
    End For
End If
```

1.2.4 The Projection Algorithm

The observation of how fast the previous algorithm converges toward the final
permutation matrix $P$, as the diagonal elements of the graphs are cumulatively
marked, led to the projection algorithm. There are two disadvantages to the marking
algorithm. First, since the algorithm begins marking immediately, the presence of
PINGs will force backtracking. Second, the algorithm must fully mark the pair
before a permutation matrix can be determined. The projection algorithm addresses
both of these issues in a two-phase approach. In the first phase, the vertex sets of the
two graphs are partitioned into equivalence classes based on Eigen-value invariance
with a single marking. This avoids the first shortcoming of the marking algorithm.
In the second phase, the marking algorithm is performed only on the sub-matrices
whose corresponding vertices belong to the same equivalence class from phase one. This refines the original vertex partition, and at each stage of refinement, the similarity transformation matrix $Q$ is examined and a candidate permutation matrix $P$ is computed by solving an assignment problem that maximizes the sum of the absolute values of $Q$'s entries (a one per-row, per-column choice). This avoids having to mark the entire matrices before a possible permutation matrix is found. We will now discuss each of the phases of the algorithm individually.

The first phase will determine equivalence classes of a graph matrix by separately marking each of its diagonal entries and checking to see if the Eigen-values are the same. The process will be done for both graphs and can complete in $O(n^4)$ steps. A Possible Mapping Tree (PMT) is constructed from these equivalence classes, where at level $i$, all possible mappings between the $i^{th}$ vertex equivalence classes of $G_A$ and $G_B$ are represented. Note that for some graphs this tree may just contain a root node and leaf nodes. The PMT gives us a vertex partitioning, for which all diagonals for each vertex in a partition are marked with a value sufficiently distinct from the other partitions, to allow for refined future partitioning. If the newly marked matrices are isospectral, then the algorithm moves on to the second phase described below.

Now, a refined marking is executed on vertex pairs in the same equivalence class. At each stage of refinement, $Q$ is computed, which converges towards some $P$. Any $Q$ defines a weighted matching, whose cost is bounded by $n$, by some $P$ that
maximizes the overall weighted matching cost between the vertices of $G_A$ and $G_B$.

The best choice for the next proper marking, at each step, is the one that maximizes the overall cost from $Q$. The running time for this choice computation is $O(n^4)$.

If no backtracking is needed, since the maximum number of markings is $n - 1$, the running time of this algorithm is $O(n^5)$. It is possible that the algorithm may need to backtrack after marking some nodes, when it is unable to proceed to a permutation matrix $P$. In this case, the running time again may become exponential as in the marking algorithm. However, if PINGs are present, non-isomorphism will be detected early, unlike in the marking algorithm. In some cases, if backtracking is not required, $P$ may be found earlier than in the marking algorithm as well.
2. Developer Requirements

This section describes the requirements in paragraph format for the Graph Isomorphism Program from a developer’s point of view. Some of these requirements were verbally discussed and some were left to be decided by the developer.

This program will determine if two simple, undirected, un-weighted graphs with no self-loops or multiple edges are isomorphic or not. The program will have a Graphical User Interface (GUI) and must be able to run on Microsoft Windows XP. Once the program is launched, the user will be allowed to input two graphs to check if they are isomorphic or not.

The program will allow for four types of graph input. Both graphs must be specified using one of these four input formats. Two of the input types will have a limit of one-hundred nodes. The user will have to select which format he/she would like to use, and the program will provide a method for this selection. The user may choose to supply the graphs by using GUI controls designed to represent adjacency matrices for the graphs. There will be two of these Adjacency Matrix Graph Representation (AMGR) GUI controls, side-by-side, to allow for graph input and modification. The AMGRs will allow the user to input graphs with up to one-hundred nodes.
The user will be able to specify files for the input graphs. The format of such files will be explained in a ‘ReadMe.doc’ file accompanying the program. The number of nodes and edges for the input graphs will be unlimited when using this type of input.

The third type of graph input will be a Graph Diagram Builder (GDB), which will allow the user to build a regular graph diagram by adding nodes and edges to this builder. This GDB will also allow for deletion of nodes and edges, and a mechanism for re-labeling of nodes. There will need to be one GDB for each input graph, and it should not allow any self-loops or multiple edges, and no more than one-hundred nodes.

The fourth and last type of graph input will be a Random Graph Generator (RGG), which will allow the user to generate random input graphs. The user will need to specify the number of nodes and edges, with no graph-size limit imposed. The user will be able to create both graphs randomly, or create the one random graph and another random graph isomorphic to the first.

The program should treat the AMGRs as the main input format to read from prior to performing the isomorphism check, as discussed below.

A button will be provided for the user to click on once the input graphs have been provided in one of the three formats. When this button is clicked, the program will first determine which format the user chose to provide the input graphs. If the user chose to provide the input graphs from files, the program will check to see if the
files exist and that they are in the right format. The format will be checked as the program reads in the graphs, from the files, into the AMGRs. The program will also, at this time, check to see that the graphs do not contain any self-loops or multiple edges. If the user chose to provide the input graphs using the GDBs, the AMGRs will be updated from them. Similarly, if the user chose to provide the input graphs from the RGGs, the AMGRs will be updated from them.

After checking the input and updating the AMGRs, the program will check to see if the graphs have the same number of nodes and edges. Then finally, the main part of the program is executed to see if the graphs are isomorphic or not. The program will check for isomorphism according to The Marking Algorithm presented in Altman’s paper [3]. This algorithm will not be discussed in this section, but will be discussed in the detailed design for this program along with some of its implementation details.

Once the program has checked for isomorphism and the results have been given to the user, the user-provided inputs the will not be cleared. Another button will be provided to clear the form, if the user desires. This will allow the user to modify the graph inputs and check for isomorphism again using the same button. The user can either modify the AMGRs, or modify the input graphs using the same format used to provide the input graphs originally. The user will have to select whichever format he/she will modify for both graphs, as in the previous run. Then the user can
make the modifications, and the program will then use the input format selected, to obtain the input graphs before checking them for isomorphism.

When using the RGGs for the input graphs, if the user chooses, the program will generate the second graph (B), isomorphic to the first (A), by performing random permutations. A random permutation matrix (P) will be created and \( P^TAP \) will produce B. The program will store P, also while running the algorithm to check for isomorphism when using RGGs, the program will also store the permutation matrix produced by the algorithm. A button will be provided to compare these two permutation matrices, after checking for isomorphism.

Finally, the program will also provide a progress bar and status text-indicator to show the progress while checking for isomorphism. The time taken to check for isomorphism will also be reported by the program.
3. High-Level Design

This section is for the Graph Isomorphism Program. It will discuss in paragraph format, how the program will accept input and give output. In other words, it will describe the Graphical User Interface (GUI) for the program. These designs follow directly from the requirements for the program.

The program will need some way of allowing the user to choose the way in which the input graphs will be provided. That is, either from adjacency matrices, files, graph diagrams, or random graphs. This can be accomplished by radio buttons across the top left of the form. The first one on the left will be “Adjacency Matrices” and will be selected by default. Then we will have the radio button for “Files,” then one for the “Graph Diagram Builder,” and the last one on the right, will be for the “Random Graph Generator.” Above these radio buttons will be the following label: “Select the method for providing the input graphs.”

Below the radio buttons will be two text fields labeled “Graph A” and “Graph B” for the full paths to the input files. There will also be “Browse” buttons to the right of each of these text fields to allow the user to navigate to the files. All four of these controls will be disabled unless the user selects the “Files” radio button. Below these controls, there will be three tabs: one for the “Adjacency Matrix Graph Representation” (AMGR) controls, one for the “Graph Diagram Builder” (GDB) controls, and one for the “Random Graph Generator” (RGG) controls.
The AMGR controls will be two side-by-side check-box grid controls with labels “0” to “100” for both rows and columns. All check-boxes in each of the AMGRs will be unchecked and the user will have to simply check a box to represent an edge. Both of these controls will be disabled until the user selects the number of nodes from a list box, labeled “Number of nodes,” in the middle of the two controls. When the user selects the number of nodes, the controls will be enabled and the grids resized to match the number of nodes selected. These grid controls will need vertical and horizontal scrollbars to allow the user to scroll through the rows and columns. If the user selects a check-box on the diagonal of either grid, to represent a self-loop, a message will be displayed that the selection will be ignored, as self-loops are not allowed.

The GDB controls, will reside on another tab. Each of these side-by-side controls will allow the user to build a diagram of a graph using nodes and edges. The details of these controls cannot yet be determined, as no such standard controls exist. Most probably, some third party components will be used to build the GDB controls. These controls will not allow, or the program will ignore, self-loops and multiple edges. These controls will not allow the user to attempt to build a diagram with more than 100 nodes. These controls will allow the user to change the labels of nodes, delete edges, and delete nodes.

The last and third tab will be for the RGG controls. At the top of this tab, there will be a label, “Specify the number of nodes.” Below this label will be a text
box to allow the user to enter the number of nodes. Below this, there will be another label, “Specify the number of edges.” Below this label will be another text box to allow the user to enter the number of edges. Below and indented to the right, will be a button to “Generate Random Graph A” followed below by another to “Generate Random Graph B.” Then below these buttons, will be the label “OR,” followed below by another button to “Generate Random Graph B Isomorphic to Graph A.” Finally, below this button and aligned to the bottom right of the tab area, will be a button to “Check if Permutation Matrices Equal.” This button will only be enabled if the user chooses to generate graph B isomorphic to A, and after the program has checked for isomorphism and found the graphs to be isomorphic. If the program works correctly, the latter will always be true if the former is true.

Underneath these tabs to the right there will be two buttons, one for “Clear Form” and to the right of that, one for “Check for Isomorphism.” The form clear button will clear all the input the user has provided and reset the form, variables, and data structures. The other button, when clicked, will first check all the input for errors, then store the graphs in an internal object representation along with other variables, and then find out if the graphs are isomorphic. The program will check for isomorphism according to the Marking Algorithm presented in Altman’s paper [3]. This algorithm will not be discussed here but will be discussed in the detailed design for this program along with other implementation details.
On the opposite side of the form, from the buttons discussed in the above paragraph, to the left, will be a status indicator, below which will be a progress bar. As the program checks for isomorphism, the different stages will be indicated by the status indicator. Suitable status indication texts will be decided and implemented when the program is written. Similarly, the progress bar will progress as computations are completed by the program, and these computations and progressions will have to be decided at the time the program is written. This is due to details that have not yet been designed and are beyond the scope of this section.

Once the program has found the graphs to be isomorphic to one another, or not, this result will be indicated by the status indicator and the progress bar will be reset. A message box will also accompany, to indicate the result. After the user clicks the “OK” button on this message box, the status indicator will also be reset, but the form will not be cleared. This is so that the user can modify the input and check for isomorphism again.

Lastly, we will discuss the format of the input files, if files are used for the input graphs. The files must be simple text files and could be created in Notepad, which is available in Microsoft Windows XP. The first line of the file will be the number of nodes. The node numbering or labeling will begin at 1, not 0. All other lines will simply give two numbers separated by a comma, to represent an edge. There can only be one edge on each line. Empty lines or lines with only white-space
characters will not be accepted anywhere in the files. An example of a simple file would look like the following.

```
6
4,5
6,2
1,3
4,3
1,2
```

Fig 3.1 Example Graph Input File
4. Detailed Design

This section will address in paragraph format, the implementation design details for the program. These designs follow directly from the requirements and high-level design for the program. The previous two sections should be reviewed prior to reading this section. This detailed design will be the final documentation leading to the actual implementation of the program.

First, we will discuss the hardware and software requirements for running the program and for the computer that will be used to build the program, along with the development environment and programming language. Second, we will discuss the main code files, classes, and data structures that will be part of the program. Third, we will discuss the specification of the main algorithms that will be used by the program for Graph Isomorphism (GI). The GDB or Graph Diagram Builder controls, that allow the user to provide input graphs by creating graph diagrams, will need to be created using some third party component(s), which will be determined during implementation.

4.1 Program Hardware and Software Specifications

The minimum hardware and software requirements for a machine to be able to run this program are as follows.

- Intel Pentium 4 - 1.6GHz or compatible CPU
- 256 MB of RAM
- 100 MB of free hard-disk space
Video Card that supports 32-Bit color quality and 1152x864 screen resolution
17-inch color monitor
Microsoft Windows XP operating system
Microsoft .NET Framework version 1.1.x

The hardware and software specifications for the machine that will be used to build this program are as follows.

- Intel Pentium 4 - 2.4GHz CPU
- 256 MB of RAM
- 24 GB of free hard-disk space
- NVIDIA GeForce4 MX 440 with AGP8x video card
- 17-inch color monitor
- Microsoft Windows XP Professional with Service Pack 2 operating system
- Microsoft .NET Framework version 1.1.4322

The development environment chosen for the program is Microsoft Visual Studio .NET Architect 2003, and the programming language is Visual Basic .NET or VB.NET. This is an object-oriented language, released in 2003, developed by Microsoft as a very high-level easy-to-use programming language. The Object Oriented Paradigm will be used in building the program.

4.2 Code Files, Classes, and Data Structures

The program will have a file named “Main.vb” which will contain all the auto-generated code from the development environment’s GUI builder for the single GUI form of the program. This file will contain any code required to allow user interaction with the GUI. It will contain the code that allows the GUI to interact with the rest of the program’s functionality. All the functionality in this file will be contained in a single auto-generated class called “Main.” This Main class will
contain the main code that will allow all of the objects of the program to interact with each other as well. Furthermore, this class will contain functions to update the status indicator and progress bar of the GUI form and will have global member object variables for the classes below. A timing mechanism will also be implemented for final analysis.

Another file called “Graph.vb” will contain a single class named “Graph” to implement the graph data structure. This data structure will be used to store internally and interact with simple graphs throughout the processing of the program. This class will contain three, main global member variables to represent a graph: one for the number of nodes in a graph, another for the number of edges, and a double dynamic integer array to store the adjacency matrix of a graph. The class will contain input functions for adding a node and adding an edge to a graph for building graphs. These functions will not allow multi-graphs, multi-digraphs, or graphs with self-loops to be created. The class will also contain accessor functions to get the number of nodes, number of edges, and any valid entry from the adjacency matrix.

A file called “GraphEigenValues.vb” will contain a single class named “GraphEigenValues” to provide the program with the functionality to find the Eigen-values of a graph. The constructor for this method will take a Graph object for which the Eigen-values need to be found. Thus, each object of this class will deal with a single graph. A third-party component will be used for the calculations, more details can be found in the “Algorithms” section. One global member variable will be
needed to represent a matrix in the format the third party component needs. The constructor will put the supplied graph’s adjacency matrix into this format. One main, read-only property function called “GetEigenValues” will use the third-party component to return the Eigen-values of the graph as an integer array.

Another code file for this program will be named “Isomorphic.vb” and will contain a single class called “Isomorphic.” The Isomorphic class will use objects of both the Graph and GraphEigenvalues classes, discussed above, to find out if the two input graphs are isomorphic to one another. This class will check for isomorphism according to the Marking Algorithm presented in Altman’s paper [3]. This paper also mentions the use of the LR-Cholesky method for finding Eigen-values of a graph. This class will implement functions for the Marking Algorithm as described in the next section. This class will also have a function for comparing Eigen-values of two graphs to see if they are the same or not, using objects of the GraphEigenvalues class.

The algorithm used by the class above will require a fifth file named “IntPairStack.vb,” which will contain a structure called “IntPair,” and a class called “IntPairStack.” The IntPair structure will contain two public member integer variables called “FirstInt” and “SecondInt.” The IntPairStack class will have one private member variable, which will be an array of IntPair structures. Later in this section, we will discuss how this stack will be used by the algorithm. The stack will be a first in last out or last in first out stack with push and pop functions that take and
return an IntPair, respectively. The constructor will take two integers, to push onto the stack.

4.3 The Marking Algorithm

This algorithm is based on the Principal Axis Theorem and a specific marking of the diagonal elements of the adjacency matrix. Computation of Eigen-values is part of the algorithm. To ensure that the algorithm for computation of these converges, it is necessary that only symmetric, positive definite matrices are used to represent the graphs. Matrices for undirected graphs are symmetric and the algorithm can be used for simple undirected graphs. For the matrices to be positive definite, it is sufficient that the matrices be strongly diagonally dominant, i.e., the sum of the absolute values of the non-diagonal entries must be less than the absolute value of the diagonal entry, for each diagonal entry in each row. This can be accomplished by setting the diagonal entries to $n$. This will not change the structure of the permutation matrix $P$, whose existence will show that the input graphs $A$ and $B$ are isomorphic.

The LR-Cholesky method for computing Eigen-values is recommended, since it is guaranteed to converge for the type of matrices described above with sufficient accuracy in polynomial time. The algorithm attempts to iteratively change or mark the diagonal entries of the adjacency matrices so that they remain isospectral or orthogonally congruent or have the same Eigen-values, while converging to an Eigen-value multiplicity of one, or having distinct Eigen-values. If successful, the matrices will be permutation similar, and thus isomorphic. The marking maps diagonal
elements of one matrix to the other, essentially creating the node permutations, and ultimately finding out if the graphs are isomorphic. With certain isospectral graphs, this algorithm will be forced to backtrack and try a different marking sequence, causing it to run in exponential time. Pairs of Isospectral Non-isomorphic Graphs or PINGs, where the graphs are orthogonally congruent, but not permutation similar, will cause this algorithm to backtrack. If the graphs are isomorphic, the algorithm marks their adjacency matrices in such a way as to produce orthogonally similar matrices, that is, they have the same Eigen-values. However, whether the graphs are found to be isomorphic or not, the algorithm hopes they remain isospectral during the marking, otherwise backtracking will be required. If the graphs are isomorphic and no backtracking is involved, the running time of the algorithm is $O(n^4)$.

Below is the pseudo-code for the algorithm. This pseudo-code needs to be refined in order to implement the backtracking functionality.

Initialize $i_{last} = j_{last} = 1$
If Not SameEigenValues($A$, $B$) then return “NOT ISOMORPHIC”
Else
   For $i$ from $i_{last}$ to $n$
      $A[i, i] = 2n$
   End For
   For $j$ from $j_{last}$ to $n$
      If $B[j, j] = n$ then
         $B[j, j] = 2n$
         If SameEigenValues($A$, $B$) then Exit j-loop
         Else If $j = n$ then BACKTRACK if possible,
         else return “NON ISOMORPHIC”
      End If
      Else $B[j, j] = n$
      End If
   End For
End If
In order to implement the backtracking functionality, this algorithm will be divided into two functions, a parent and a child. The parent function can then call the child function in a loop. The public parent function will be called “CheckIfGraphsIsomorphicByMarking” and the private child function will be called “MarkGraphs.” The Stack data structure, as described in the previous section, will be used here and the algorithms are more detailed. The pseudo-code for the parent is given below.

```
Initialize ilast = 1
Initialize jlast = 0
Initialize BackTrack = False
Initialize Stack(ilast, jlast)

Set all diagonal entries in A and B to n

While (True)
    If MarkGraphs(A, B, Stack, BackTrack) Then
        Return ISOMORPHIC
    Else
        If BackTrack Then
            If Stack is Empty Then
                Return NOT ISOMORPHIC
            Else
                Return NOT ISOMORPHIC
        EndIf
EndIf
```

If the marking is successful, the child returns true and we have isomorphic graphs. Otherwise, the child function will set the BackTrack variable, letting us know if we need to attempt to backtrack or not. If not, then the graphs are not isomorphic; otherwise, we attempt to backtrack. However, if the stack is empty, we cannot backtrack, meaning the graphs are not isomorphic. The pseudo-code for the child is
given below. It is almost the same as the original pseudo-code with some slight changes to work with the parent function and add more detail. Note that both parameters of this child function are reference parameters, that is, if the child function modifies the passed arguments, these changes will be made to the actual objects declared in the parent function. The first two are for the graphs, one is the IntPairStack and the other is the backtracking variable.

Initialize BackTrack = False
Initialize IntPair = Stack.Pop

If Not SameEigenValues(A, B) then
    return FALSE
Else
    For i from IntPair.FirstInt (ilast) to n
        A[i, i] = 2in
        For j from 1 to n
            If BO,j = n AND j != IntPair.SecondInt (jlast) then
                B[j, j] = 2in
                If SameEigenValues(A, B) then
                    jLast = j
                    Stack.Push(i, j)
                    Exit j-loop
                Else
                    B[j, j] = n
                    If j = n then
                        BACKTRACK = TRUE
                        Return FALSE
                    EndIf
                EndIf
            EndIf
        End For
    End For
    Return TRUE
EndIf
4.4 The LR-Cholesky Algorithm

Now it is time to discuss the LR-Cholesky algorithm for computing the Eigen-values of a graph. I have been researching this algorithm using resources online and have found the problem of calculating Eigen-values to be a very difficult one, in the field of Linear Algebra. In mathematics, and in particular in Linear Algebra, an important tool for describing Eigen-values of square matrices is the characteristic polynomial. Saying that $\lambda$ is an Eigen-value of $A$ is equivalent to stating that the system of linear equations $(A - \lambda I)v = 0$ (where $I$ is the identity matrix) has a non-zero solution $v$ (namely an Eigen-vector), and so it is equivalent to the determinant $\det(A - \lambda I)$ being zero. The function $p(\lambda) = \det(A - \lambda I)$ is a polynomial in $\lambda$ since determinants are defined as sums of products. This is the characteristic polynomial of $A$, and the Eigen-values of a matrix are the zeros of its characteristic polynomial. If the matrix is small, we can compute them symbolically using the characteristic polynomial. However, this is often impossible for larger matrices, in which case we must use a numerical method. In fact, for any square matrix larger than $3 \times 3$, the problem becomes difficult to solve. An analytic solution for the Eigen-values of $2 \times 2$ matrices can be obtained directly from the quadratic formula. If

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then the characteristic polynomial is
\[
\det\begin{bmatrix}
    a - \lambda & b \\
    c & d - \lambda
\end{bmatrix} = (a - \lambda)(d - \lambda) - bc = \lambda^2 - (a - d)\lambda + (ad - bc).
\]

Notice that the coefficients are the trace \( \text{TR}(A) = a + d \) and the determinant \( \det(A) = ad - bc \), so the solutions are

\[
\lambda = \frac{a + d}{2} \pm \sqrt{\frac{(a + d)^2}{4} + bc - ad} = \frac{a + d}{2} \pm \frac{\sqrt{4bc + (a - d)^2}}{2}.
\]

A formula for the Eigen-values of a 3x3 matrix or 4x4 matrix could be derived in an analogous way, using the formulae for the roots of a cubic or quartic equation.

The LR-Cholesky algorithm relies on reducing the matrix \( A \) to tri-diagonal form. It uses Laguerre's shifts for the symmetric tri-diagonal matrices. The general algorithm is as follows.

Let \( A \) be a s.p.d. matrix.

\[
\begin{align*}
A_0 &= A \\
k &= 0, 1, 2, \ldots \\
\text{choose shift } \sigma_k \\
A_k - \sigma_k I &= V_k V_k^T \quad \text{(Cholesky decomposition, } V_k \text{ is lower-triangular)} \\
A_{k+1} &= V_k^T V_k + \sigma_k I
\end{align*}
\]

Two steps of the zero shift Cholesky LR are equivalent to one step of the zero shift QR. The shift \( \sigma_k \) should be such that \( A_k - \sigma_k I \) is positive definite. When applied to a symmetric positive definite diagonal-plus-semi-separable matrix, \( D + S \), where \( D \) is the diagonal, the shift can be included into the diagonal part.

Algorithms for the Laguerre's Shift, \( V^T V \) Product, and the Cholesky Decomposition are also needed, along with the equations for their computation.
Implementing this algorithm becomes a more complex problem than implementing the algorithm for GI, discussed in the previous section. The main purpose of this program and thesis work is to solve the GI problem in Computer Science. Since this Eigen-value problem is mainly a study in Linear Algebra, and because of the complexities of its implementation, it seems unnecessary to spend the time and effort in implementing this algorithm. Also, code for this algorithm is not readily available over the internet either, it seems that only companies have implemented similar algorithms and are charging for their components and/or software.

After a significant amount of research, a .NET component that will work with VB.NET called .NET Matrix Library 2.5 made by BlueBit Software, was found. The advanced version 2.5 can calculate the Eigen-values of a matrix efficiently using the Cholesky Decomposition, for symmetric real matrices. Of course, the matrix may have to be supplied in a form that is accepted by the functions provided in the .NET component for calculating Eigen-values. In addition, learning to use the library will also be another task, but the process should be simple and documentation should be available on the internet or with the developer install. This component may need to be purchased since the trial version only handles 6×6 size matrices. Using this component will allow focus on the GI problem for the thesis work rather than worrying about the Linear Algebra problem of calculating Eigen-values.
5. Testing and Analysis

The material in this section will serve as a road-map for thoroughly testing the program. The test cases will be organized according to the number of nodes, increasing from lower boundary values. Since the program accepts input graphs in four different formats, that is, adjacency matrices, files, diagrams, and randomly generated graphs, some tests will be run four times, once for each format. It will also be necessary to have tests for a particular format that do not apply to the other formats. Similarly, for some tests it will only make sense to use one type of format.

Test data has been organized into the table below. Three of the graph input formats will be abbreviated as AMGR for Adjacency Matrix Graph Representation; GDB for Graph Diagram Builder; and RGG for Random Graph Generator. The first column is the test number for each distinct number of nodes and edges, including a letter for the different input formats. The input format used to provide the graphs is given in the “Input Type” column. The “Program State” column specifies whether, before running the test, the program was re-launched, the form was cleared, or the input modified. The “Iso” column specifies whether the test was for isomorphic graphs or not. Before testing, in all cases, we should know whether the graphs are isomorphic or not. For each test, if the program correctly identifies the graphs as isomorphic or not, the “Passed” column will contain “Yes,” to show that the program
passed the test. The “Time Taken” column gives the actual running time rounded to the nearest millisecond, as reported by the program in the status indicator.

### Table 5.1 Test Data

<table>
<thead>
<tr>
<th>Test</th>
<th>Nodes</th>
<th>Edges</th>
<th>Input Type</th>
<th>Program State</th>
<th>Iso</th>
<th>Time Taken</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>0</td>
<td>0</td>
<td>Files</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1b</td>
<td>0</td>
<td>0</td>
<td>GDB</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2a</td>
<td>1</td>
<td>0</td>
<td>AMGR</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2b</td>
<td>1</td>
<td>0</td>
<td>Files</td>
<td>Form Cleared</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2c</td>
<td>1</td>
<td>0</td>
<td>GDB</td>
<td>Form Cleared</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>3a</td>
<td>2</td>
<td>0</td>
<td>AMGR</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>3b</td>
<td>2</td>
<td>0</td>
<td>Files</td>
<td>Input Modified</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>3c</td>
<td>2</td>
<td>0</td>
<td>GDB</td>
<td>Input Modified</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4a</td>
<td>2</td>
<td>1</td>
<td>AMGR</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4b</td>
<td>2</td>
<td>1</td>
<td>Files</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4c</td>
<td>2</td>
<td>1</td>
<td>GDB</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4d</td>
<td>2</td>
<td>1</td>
<td>RGG</td>
<td>Re-launched</td>
<td>Y</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>5a</td>
<td>3</td>
<td>1</td>
<td>AMGR</td>
<td>Re-launched</td>
<td>Y</td>
<td>750</td>
<td>Yes</td>
</tr>
<tr>
<td>5b</td>
<td>3</td>
<td>1</td>
<td>Files</td>
<td>Form Cleared</td>
<td>Y</td>
<td>109</td>
<td>Yes</td>
</tr>
<tr>
<td>5c</td>
<td>3</td>
<td>1</td>
<td>GDB</td>
<td>Form Cleared</td>
<td>Y</td>
<td>94</td>
<td>Yes</td>
</tr>
<tr>
<td>5d</td>
<td>3</td>
<td>1</td>
<td>RGG</td>
<td>Form Cleared</td>
<td>Y</td>
<td>78</td>
<td>Yes</td>
</tr>
<tr>
<td>6a</td>
<td>3</td>
<td>2</td>
<td>AMGR</td>
<td>Re-launched</td>
<td>Y</td>
<td>313</td>
<td>Yes</td>
</tr>
<tr>
<td>6b</td>
<td>3</td>
<td>2</td>
<td>Files</td>
<td>Input Modified</td>
<td>Y</td>
<td>125</td>
<td>Yes</td>
</tr>
<tr>
<td>6c</td>
<td>3</td>
<td>2</td>
<td>GDB</td>
<td>Input Modified</td>
<td>Y</td>
<td>78</td>
<td>Yes</td>
</tr>
<tr>
<td>6d</td>
<td>3</td>
<td>2</td>
<td>RGG</td>
<td>Input Modified</td>
<td>Y</td>
<td>94</td>
<td>Yes</td>
</tr>
<tr>
<td>7a</td>
<td>3</td>
<td>3</td>
<td>AMGR</td>
<td>Form Cleared</td>
<td>Y</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>7b</td>
<td>3</td>
<td>3</td>
<td>Files</td>
<td>Re-launched</td>
<td>Y</td>
<td>406</td>
<td>Yes</td>
</tr>
<tr>
<td>7c</td>
<td>3</td>
<td>3</td>
<td>GDB</td>
<td>Re-launched</td>
<td>Y</td>
<td>406</td>
<td>Yes</td>
</tr>
<tr>
<td>7d</td>
<td>3</td>
<td>3</td>
<td>RGG</td>
<td>Re-launched</td>
<td>Y</td>
<td>109</td>
<td>Yes</td>
</tr>
<tr>
<td>8a</td>
<td>4</td>
<td>2</td>
<td>AMGR</td>
<td>Form Cleared</td>
<td>Y</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>8b</td>
<td>4</td>
<td>2</td>
<td>Files</td>
<td>Form Cleared</td>
<td>Y</td>
<td>125</td>
<td>Yes</td>
</tr>
<tr>
<td>8c</td>
<td>4</td>
<td>2</td>
<td>GDB</td>
<td>Form Cleared</td>
<td>Y</td>
<td>94</td>
<td>Yes</td>
</tr>
<tr>
<td>8d</td>
<td>4</td>
<td>2</td>
<td>RGG</td>
<td>Form Cleared</td>
<td>Y</td>
<td>109</td>
<td>Yes</td>
</tr>
<tr>
<td>9a</td>
<td>4</td>
<td>2</td>
<td>AMGR</td>
<td>Form Cleared</td>
<td>N</td>
<td>16</td>
<td>Yes</td>
</tr>
<tr>
<td>9b</td>
<td>4</td>
<td>2</td>
<td>Files</td>
<td>Input Modified</td>
<td>N</td>
<td>125</td>
<td>Yes</td>
</tr>
<tr>
<td>9c</td>
<td>4</td>
<td>2</td>
<td>GDB</td>
<td>Input Modified</td>
<td>N</td>
<td>78</td>
<td>Yes</td>
</tr>
<tr>
<td>9d</td>
<td>4</td>
<td>2</td>
<td>RGG</td>
<td>Input Modified</td>
<td>N</td>
<td>78</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Now that we have a set of test cases and their results, we can analyze the test data.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10a</td>
<td>4</td>
<td>3</td>
<td>AMGR</td>
<td>Input Modified</td>
</tr>
<tr>
<td>10b</td>
<td>4</td>
<td>3</td>
<td>Files</td>
<td>Re-launched</td>
</tr>
<tr>
<td>10c</td>
<td>4</td>
<td>3</td>
<td>GDB</td>
<td>Re-launched</td>
</tr>
<tr>
<td>10d</td>
<td>4</td>
<td>3</td>
<td>RGG</td>
<td>Re-launched</td>
</tr>
<tr>
<td>11a</td>
<td>4</td>
<td>3</td>
<td>AMGR</td>
<td>Input Modified</td>
</tr>
<tr>
<td>11b</td>
<td>4</td>
<td>3</td>
<td>Files</td>
<td>Form Cleared</td>
</tr>
<tr>
<td>11c</td>
<td>4</td>
<td>3</td>
<td>GDB</td>
<td>Form Cleared</td>
</tr>
<tr>
<td>11d</td>
<td>4</td>
<td>3</td>
<td>RGG</td>
<td>Form Cleared</td>
</tr>
<tr>
<td>12a</td>
<td>5</td>
<td>4</td>
<td>AMGR</td>
<td>Input Modified</td>
</tr>
<tr>
<td>12b</td>
<td>5</td>
<td>4</td>
<td>Files</td>
<td>Input Modified</td>
</tr>
<tr>
<td>12c</td>
<td>5</td>
<td>4</td>
<td>GDB</td>
<td>Input Modified</td>
</tr>
<tr>
<td>12d</td>
<td>5</td>
<td>4</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>13a</td>
<td>5</td>
<td>4</td>
<td>AMGR</td>
<td>Re-launched</td>
</tr>
<tr>
<td>13b</td>
<td>5</td>
<td>4</td>
<td>Files</td>
<td>Input Modified</td>
</tr>
<tr>
<td>13c</td>
<td>5</td>
<td>4</td>
<td>GDB</td>
<td>Input Modified</td>
</tr>
<tr>
<td>13d</td>
<td>5</td>
<td>4</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>14a</td>
<td>6</td>
<td>9</td>
<td>AMGR</td>
<td>Form Cleared</td>
</tr>
<tr>
<td>14b</td>
<td>6</td>
<td>9</td>
<td>Files</td>
<td>Input Modified</td>
</tr>
<tr>
<td>14c</td>
<td>6</td>
<td>9</td>
<td>GDB</td>
<td>Input Modified</td>
</tr>
<tr>
<td>14d</td>
<td>6</td>
<td>9</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>15a</td>
<td>6</td>
<td>9</td>
<td>AMGR</td>
<td>Form Cleared</td>
</tr>
<tr>
<td>15b</td>
<td>6</td>
<td>9</td>
<td>Files</td>
<td>Input Modified</td>
</tr>
<tr>
<td>15c</td>
<td>6</td>
<td>9</td>
<td>GDB</td>
<td>Input Modified</td>
</tr>
<tr>
<td>15d</td>
<td>6</td>
<td>9</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>16</td>
<td>RGG</td>
<td>Re-launched</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>16</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
<td>45</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>45</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>50</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>21</td>
<td>50</td>
<td>50</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>22</td>
<td>52</td>
<td>100</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>23</td>
<td>52</td>
<td>100</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>120</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>120</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>26</td>
<td>105</td>
<td>330</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
<tr>
<td>27</td>
<td>105</td>
<td>330</td>
<td>RGG</td>
<td>Input Modified</td>
</tr>
</tbody>
</table>

Now that we have a set of test cases and their results, we can analyze the test data.
The program result was as expected; for all test cases, meaning, in each case, the program passed the test. For the first tests, up to and including 4d, the program did not report any running times. This is due to the fact that the program did not run the algorithm to check for isomorphism. This is the expected behavior in these tests, since the nodes were less than three and the number of edges could not be more than one.

From test 5a through 15d, the program state is changed to see if the program gets the correct result in different situations, which it did. The running times in these tests are so small due to the small number of nodes and edges, that they are more dependant on the machine and operating system than the number of nodes and/or edges. However, in most cases, if the user needs to run multiple checks, the user would simply modify the input rather than clearing the form or re-launching the program. Therefore, in the tests where the input was modified, checks for isomorphism were performed a few times to make sure the running times were not significantly different. This is important if we want to analyze the running time. The program code does not store any data for future calculations or processing between checks for isomorphism. With these re-checks, we now know that this is also the case for the third party component used for Eigen-value calculations, the operating system, the .NET framework, and the machine itself.

From test 5a through 15d, the different input types are also tested, to see if the program gets the correct result no matter how the input graphs are provided, which it
did. These tests also show that the type of input format used to provide the graphs effects the running time. Therefore, we should use the same input type for analyzing running times.

The last tests, from test 16 through 27 all use the RGG input type to randomly create input graphs, both isomorphic and not. Generally speaking, when the program randomly creates both input graphs, they will not be isomorphic. If the user chooses to create graph B isomorphic to A, then they will always be isomorphic. In general, these tests show that the running times are significantly higher when the graphs are isomorphic. This makes sense since the randomly generated graphs probably do not have the same Eigen-values before marking, even when the diagonal elements are all the same in both graphs. These tests also show that the running times are much more dependant on the number of nodes and not the number of edges. This is because increasing the number of edges, with the same number of nodes, to make the graphs more dense, does not significantly change the running time.

For isomorphic tests, from test 16 through 27, the program option to randomly create graph B isomorphic to graph A, was used. When using this option, the program randomly generates a permutation matrix to create graph B isomorphic to A, which is stored. The program then also has another option to see if this permutation matrix is the same as the one generated by the algorithm, after checking for isomorphism. This check was performed in all of these tests and showed that there
can be more than one permutation matrix. Furthermore, it showed that it is more likely to have the same permutation matrices for denser graphs.

The program also has other options for the GDB input format. You can copy the diagram from graph A to B and then modify it to save time. This option was used for all GDB input format tests. Other options allow you to swap two node labels in the diagrams and randomly re-label the nodes in the diagram for graph B. These two options were tested independently from the tests given in the table above, and worked correctly. Similarly, many other tests, not in the table above, were performed to make sure the program works correctly. For example, there were many tests performed to ensure the program worked correctly and displayed the correct messages when bad input was provided by the user.

Now it is time to do the final running time analysis, that is, to show that the running time is actually $O(n^4)$, as is for the algorithm, and that the program code correctly implements the algorithm. The running time of $O(n^4)$ is based on the number of nodes only, and the program test results show that denser graphs do not significantly change the running time, if the nodes are kept constant. This running time is the highest order term of a quatric equation,

$$An^4 + Bn^3 + Cn^2 + Dn = \text{Actual Running Time}.$$  

Here A, B, C, and D are all constants and $n$ is the number of nodes. From tests 21, 23, 25, and 27, we can get the numbers of nodes and actual running times, and solve
the system of equations that result. The system of equations that needs to be solved is,

\[
\begin{align*}
6250000A + 125000B + 2500C + 50D &= 1234 \\
7311616A + 140608B + 2704C + 52D &= 2016 \\
10000000A + 1000000B + 100000C + 100D &= 28781 \\
121550625A + 1157625B + 11025C + 105D &= 38641.
\end{align*}
\]

The solutions for A, B, C, and D were found using a computer program available on the internet [20]. This gives us an equation for the actual running time for the program,

\[
\frac{93323757}{21221200000} n^4 - \frac{58917728297}{63663600000} n^3 + \frac{142439935447}{2122120000} n^2 - \frac{19958154731}{12732720} n.
\]

Using this equation, now we can predict the running time for, say, 75 nodes, which comes to 8717 milliseconds or approximately 9 seconds. Running the program using randomly generated isomorphic graphs with 75 nodes and 150 edges, a running time of 9345 milliseconds or approximately 9 seconds was reported. While not exactly equal, the two running times are equal when rounding to the nearest second.

However, this will not always be the case, because the algorithm has backtracking built into it, and depending on the graphs, it may run in exponential time.

Furthermore, the test data used to create the system of equations above may have contained tests in which backtracking was needed. To some extent however, we have shown that the running time of the implemented algorithm is indeed \(O(n^4)\).

Therefore, the actual running time may be less or more than the running time calculated using the equation above.
6. Conclusion

The Vertex Filtering (VF) and McKay's Nauty were two algorithms researched in more depth as they were advertised as the best available. However, there is no efficient general Graph Isomorphism (GI) algorithm. In the worst cases, these algorithms perform the same as the brute-force naïve algorithm. That is, with a $O(t^s)$ exponential running time, where $t$ and $s$ are the number of nodes in the input graphs. However, in most cases, they outperform any other available algorithms. It seems the VF2 and the Nauty algorithm can perform equally well, depending on the type of graphs and valence. The running times for these algorithms are high-order polynomials.

Also studied in depth, were the Marking Algorithm and the Projection Algorithm, both given in Altman's paper [3]. These are not solutions to the general problem, nor are they approximation algorithms; they basically restrict the input to simple undirected graphs. For many applications, they can be very useful. The Marking algorithm has a running time of $O(n^4)$ and the Projection Algorithm has a running time of $O(n^5)$. In the worst case, both will still run in exponential time, in the Projection Algorithm the likelihood of this happening is decreased.

The GI program designed, written, and tested; was discussed with Altman and implements the Marking Algorithm [3]. Developer Requirements, High-Level Design, and Detailed Design documents were created for the program. This program
uses the Marking Algorithm to see if two simple undirected graphs are isomorphic.

While writing these documents, a significant amount of research was done on calculating Eigen-values, using the LR-Cholesky method, and no useful pseudo-code or actual code was found for this daunting task. Since the algorithm relied on this, a suitable third-party component was found. The same is true for the graph diagram builder that is part of the program and allows the user to enter the two input graphs by drawing them. However, this functionality required much more coding to get it to work correctly, and even more research and testing to find a suitable third-party component.

Writing the program took up most of the time spent on this thesis, and modifications to the design were required throughout the process, but in the end, it was successful. A Testing and Analysis document was also written and used as a road map for thoroughly testing the program. The program created is a user-friendly GUI program with robust error checking and reporting for bad input and system exceptions. Four different input formats can be used to specify the graphs; these include adjacency matrices, files, diagrams, and random generation. The adjacency matrices are provided by simply selecting the number of nodes from a list, and then selecting check-boxes for edges. The diagram builder controls required the most custom programming and allow the user to copy the diagram for graph A to graph B. Options are also available to swap labels of two nodes and for graph B, to randomly re-label the nodes. Graph B can be randomly generated along with A or isomorphic
to A, using a randomly generated permutation matrix. This matrix can then be tested for equality against the one found by the algorithm, after it discovers the graphs are isomorphic. The program also includes an option to clear and reset the form, and a status indicator along with a progress bar. The program has been thoroughly tested and is an easy-to-use, reliable tool for checking if two graphs are isomorphic.

When randomly generating both graphs, for larger numbers of nodes, they were generally found to be non-isomorphic. When randomly generating graph B isomorphic to A, the random permutation matrix used to create graph B was not always the same as the one found by the algorithm. This shows that there can be multiple permutation matrices for pairs of isomorphic graphs. It was also found that this is less likely for denser graphs, and in such cases, the two permutation matrices were the same. The program’s running time was dominated by the number of nodes, as is the case for the algorithm. In fact, in the Testing and Analysis section of this thesis, to some extent, it is shown that the running time of the program is $O(n^4)$, as given for the algorithm [3].
REFERENCES


