Hierarchy of Graph Isomorphism Testing

Wen-Chi Chen

Computer Science
California Institute of Technology

5140:TR:84

Hierarchy of Graph Isomorphism Testing

by

Wen-Chi Chen

5140:TR:84

Department of Computer Science
California Institute of Technology
Pasadena, California

Submitted to the Computer Science Department in Partial Fulfillment of the Requirements for the Degree of Master of Science.

The research was sponsored in part by the Silicon Structure Project and in part by IBM Corporation.

Table of Contents

1. Introduction

As of late, Graph isomorphism algorithms are of great interest in VLSI circuit design. However, their usefulness is not limited to this particular topic. For example, they are used almost exclusively in all Artificial Intelligence applications where a matching problem between graphs and relational structures must be solved. Similar kinds of matching problems appear in pattern recognition[Ts79], scene analysis [Am73] and fingerprint checking [Me83]. In Structural Chemistry, this problem shows connection with the classification of chemical compounds [Ra74] and manipulation of a large number of chemical structures when used in a chemical information retrieval system [Su64]. In VLSI, the results from isomorphism testing assist debugging when a specified circuit is compared with a circuit extracted from a lower intermediate form.

The definition of graph isomorphism is as follows: Graph G(V,E) is a graph with vertex set V and edge set E. A graph G1(V1, E1) is said to be isomorphic to another graph G2(V2,E2) if there is a one to one mapping λ from V1 to V2 and λ preserves the adjacency relationship between E1 and E2, namely, if and only if (x,y) belongs to E1, (λ(x), λ(y)) belongs to E2. The dilemma of deciding if two given graphs are isomorphic to each other is called the graph isomorphism problem. This problem has received so much attention in the last decade that it is now sometimes referred to as the "graph isomorphism disease" [Rc77] since it has been so infectious in both the theoretical and practical computer science literature.

Recent theoretical attention of graph isomorphism emerges mainly from the study of NP-complete (nondeterministic polynomial) problems [Ga79]. A problem becomes a "decision problem" when it has only two possible solutions, (yes or no), and is said to be in the NP class if it can be solved in polynomial time using a nondeterministic Turing machine. Similarly, a problem is defined to be in the P class if and only if it can be solved in polynomial time using a deterministic Turing machine. We call a problem NP-complete if it is one of the hardest problems in NP. Since every problem in NP can be transformed to completeness in polynomial time, An NP-complete problem is not in the P class unless all the NP problems are in P.

The study of NP is of particular interest to computer scientists since they occre quite frequently in practical problems. More than a thousand commonly encountered problems from mathematics, computer science and operations research have been shown to be NP-complete. Consequently, it appears that P and NP provide a natural hierarchy for classifying computational complexity.

We still do not know whether or not P and NP are equal. While it is generally believed that they are not equal, it has yet to be proven. The question of whether or not P is equal to NP has become one of the most challenging unsolved problems in computer science and mathematics. And although a great deal of research have been put into this subject, there has yet been no major breakthrough.

To circumvent this seeming quagmire of difficulty, one can alternatively examine the existence of NPI, an intermediate class of problems between P and NP-complete. If it can be shown that NPI exists, P and NP cannot be the same. Conversely, if P is not equal to NP, then the set NPI is not empty. (This follows from a more general result proved by Ladner [La75].) There are several major candidates which are in NP, but it is still unknown whether they reside in P, NP-complete or neither. Graph isomorphism is currently the most distinguished open problem for the NPI search. If a particular problem is of similar complexity to graph isomorphism, some researchers dub it as being "isomorphism complete".

Several subproblems of graph isomorphism testing have been proved to be in the P class. A simple example of this is restricting a graph to be a rooted tree. Isomorphism of a rooted tree can be solved in linear time in the number of the vertices of the tree [CB81]. A general tree isomorphism can be easily rooted since a tree has at most two centers and these centers can be identified in linear time. If there is only one center, it can be labeled as the root. If there are two centers, we check both possibilities. Hence a general tree isomorphism problem turns out to be linear too. Some other interesting subproblems of graph isomorphism are also linear such as: isomorphism of planar , interval, and outplanar graphs. [HW74],[LB79],[BJM79].

For more complicated subproblems, some significant subcases have recently been shown to be solvable in polynomial time: graphs embeddable in the projective plane [Li80]; graphs embeddable in a surface of genus bounded by k [Mi80]; graphs with bounded degrees[Lu80], and graphs with bounded eigenvalue multiplicity [BGM82].

In addition, there are many other interesting extensions of the isomorphism problem. For example both 'subgraph isomorphism' and 'largest common subgraph' involve generalizing graph isomorphism in a way which allows more freedom in mapping one graph to another. On the other hand, isomorphism with restrictions can also generalize the original problem. They restrict the mapping from the initial graph to the target graph without intersecting with the attached set. Both kinds of generalizations give NP-complete problems. Indeed, graph isomorphism is along the boundary of P and NP-complete.

There are also some other problems along this border. A problem is isomorphism complete if it is polynomial transformable to the graph isomorphism problem. Almost all the known isomorphism complete problems are closely related to the isomorphism problem, such as: isomorphism of directed graphs; regular graphs; bipartite graphs; labeled graphs; line graphs; and chordal graphs along with isomorphism of semigroups and finite automata [Mi79,RC77]. The graph automorphism partitioning problem is also isomorphism complete. All of the above results can be proved by using graph transformations.

Graph isomorphism testing is a common problem to many fields. Although there are many nice theoretical results about this problem, most of them are not quite practically useful for general applications. Hence, a lot of heuristic algorithms have been proposed. In this paper we present two new heuristic graph isomorphism algorithms, GITestA and GITestB, as well as an isomorphism testing hierarchy. GITestA and GITestB are very powerful and can be used to test most graph isomorphism problems occuring in practical cases. GITestB is especially useful since its capability is similar to currently available general graph isomorphism testing algorithms. Also its time and space complexity is much lower. Using the isomorphism testing hierarchy, any graph isomorphism problem can be solved while maintaining an almost optimal average complexity.

In the remainer of this section, basic terminologies and definitions are given for easy reference. Since graph invariants are often used in heuristic graph isomorphism algorithms, a set of vertex invariants and their characteristics are discussed in section 2. These vertex invariants are used to form a graph invariant which is the basis of our graph isomorphism algorithms. In section 3, this algorithm is presented as well as analyzed. Several observations about this older algorithm are shown in section 4, when a new algorithm is proposed. It has similar power and much improved time as well as space complexity. Directed graph isomorphism testing are also described.

Strongly regular graphs and BIBD graphs will be introduced in section 5. Unfortunately, some of them are immune to our isomorphism testing algorithms. However, graph transforms can be used to make these graphs more receptive to testing. A hierarchy of graph isomorphism testing algorithms with a graph transform with our algorithms embedded is introduced. Finally, in section 6, we remark on our approach, and discuss ideas for further improvements.

1.2. terminology and definitions

Throughout this report, G(V,E) will refer to a graph G with vertex set V (of cardinality n) and edge set E (of cardinality m). A graph is said to be directed when its edges have orientation, i.e., (u,v) is not the same as (v,u) in general. Unless otherwise stated, all graphs are assumed to be undirected and without loop, that is, no edge joining a vertex to itself.

G1(V1,E1) is a subgraph of G(V,E) if and only if V1 is a subset of V , E1 is a subset of both E and V1 $\times$ V1. G(V,E) is a complete graph if and only if E=V $\times$ V. A graph G(V,E) is called bipartite if there are two vertex sets V1 and V2 such that V1 $\bigcup$ V2 = V, V1 $\bigcap$ V2 = $\emptyset$, and any edge (u,v) of E belongs either to V1 $\times$ V2 or V2 $\times$ V1 which are the same for undirected graph.

Two vertexes u and v are called neighbors if there is an edge (u,v) or (v,u) . A vertex V is of degree r if it has r neighbors. A graph is called a regular graph if its vertexes are all of the same degree. K_n denotes a regular graph with n vertices while $K_{p,q}$ denotes a bipartite graph when the vertices of V1 and V2 are of the same degree p and q respectively.

A path is a sequence of vertices $v_1, v_2, ..., v_n$ where (v_i, v_{i+1}) and $1 \leq i \leq n-1$ is an edge. The distance d(u,v) between two vertices u and v in G is defined as the length of the shortest path joining u and v. The shortest u-v path is often called a geodesic. The vertex diameter vd(v) is the longest geodesic out from v. The diameter of a graph itself is the largest vertex diameter of its vertices.

A graph G(V,E) is a complement of another graph G2(V2,E2) if V2=V and E $\cup$ E2= V $\times$ V. A vertex v is at distance d from another vertex u if there is a path from V to U with length d. The adjacency matrix A of G(V,E) is an n by n 0, 1 matrix with entry (u,v) to be 1 if and only if (u,v) belongs to E. A graph is connected if every pair of points are joined by a path. A maximal connected subgraph of G is called a connected component or simply a component of G.

2. Graph Invariants and Vertex Invariants

A *graph invariant* (or *g-invariant*) λ is a unary function mapping from the set of all graphs $\mathcal{G}$ to a certain set R such that any two isomorphic graphs G1, G2 $\in \mathcal{G}$ must have the same images, *i.e.*, $\lambda(G1) = \lambda(G2)$. The image of a g-invariant is called a *g-invariant value*. Given a g-invariant λ, two graphs are λ-*distinguishable* if they have different images under the mapping λ, otherwise they are λ-*similar*. Throughout this paper, two graphs are called *distinguishable* or *similar* instead of being called λ-*distinguishable* or λ-*similar* whenever the g-invariant λ is clearly referred. A g-invariant λ is *complete* if any two graphs are either isomorphic or λ- *distinguishable*. Otherwise it is *incomplete*.

Most graph isomorphism algorithms are based on g-invariants. Commonly used g-invariants are: *V*(G), the number of vertices in G; *E*(G), the number of edges in G; total-degree(G), the sum of the degrees of the vertices in G; and degree-sequence(G), the ascending sequence of the degrees of the vertices in G. An adjacency matrix *A* is not a g-invariant. However, the determinant $|A-\sigma I|$, where I is an identity matrix of dimension *V*(G), does not depend on how G is labelled. Therefore, it is a g-invariant. Consequently, the characteristic polynomial of *A* as well as the spectrum (the set of eigenvalues) of G are also g-invariants. All of the above g-invariants have been shown to be incomplete, namely, there are graphs, nonisomorphic yet *similar* under these g-invariants.

There are indeed some complete g-invariants. For example, an *upper triangle number* S of an m by n, 0,1 matrix A is defined as,

$$S = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{i,j} 2^{im+j}$$

where $A_{i,j}$ is the matrix element at ith row and jth column.

If one were to simultaneously permute the rows and columns of the adjacency matrix of a graph G, the ***maximum adjacency matrix*** of G could be obtained. (It is the one with the largest upper triangle mumber.) The next theorem will demonstrate that the maximum-adjacency-matrix-function M, (the function that assigns to G the upper triangle number of its maximum-adjacency-matrix), is a complete g-invariant.
Unfortunately, given a graph G, there is no known polynomial algorithm to get this g-invariant M(G).

Theorem 2.1: Maximum-adjacency-matrix-function M(G) is a complete g-invariant.
Proof:
A simple proof can be shown based on the fact that there is a one to one relationship between a graph and an image of M(G). First we prove that M(G) is a g-invariant.
Assume G is isomorphic to G$'$. A and A$'$ are the adjacency matrices of G and G$'$ respectively. M=M(G) and M$'$=M(G$'$). By the definition of graph isomorphism, there exists permutation matrix P such that A = $P \times$ A$' \times P^t$, where P^t is the transpose matrix of P. Moreover there is a permutation matrix Q such that M(G) = $Q \times$ A $\times Q^t$. Therefore, M(G) = $Q \times P \times$ A$' \times P^t \times Q^t = R \times R^t$, where $R = P \times Q$, is a permutation matrix. So M(G$'$) $\geq$ M(G) by the definition of maximum adjacency matrix. Similarly, we can prove M(G) $\geq$ M(G$'$). Hence , M(G) = M(G$'$) and M(G) is a g-invariant.

Next we prove that M(G) is complete.
If M(G) = M(G$'$). Since M(G) is also a symmetric 0,1 matrix with all the diagonal elements being zero, M(G) is an adjacency matrix of a certain graph.

Let H be the graph associated with M(G). Thus, H is isomorphic to G because there is a permutation matrix P such that $M(G) = P \times A \times P^t$. Similarly, H is isomorphic to G′. Therefore G is isomorphic to G′ and M(G) is a complete g-invariant. ∎

A ***vertex invariant*** or ***v-invariant*** ν is a binary function mapping from a set of 2-tuples (G,v), here G is a graph and v is a vertex of G, to a certain set P, such that vertices of the same automorphism partition of G will have the same images under the mapping ν. The image of a v-invariant is called a ***v-invariant value***. Definitions of ***similar*** and ***distinguishable*** vertices as well as ***complete*** v-invariants are analogous to those of g-invariants. Examples of v-invariants include degree(G,v): the degree of v in G, triangle(G,v): the number of triangles in G containing v.

In this paper, four classes of v-invariants, ν_1, ν_2, ν_3 and ν_4, will be used in testing graph isomorphism. A ***k-layer graph*** $LG^k(V^k, E^k)$ of the vertex v of graph G(V,E) is a subgraph of G such that V^k is the set of vertices k-distance away from v and E′ is the intersection of E and $V^k \times V^k$. Let $LG^k(V^k, G^k)$ be the k-layer graph of v in G, $LG^{k+1}(V^{k+1}$, and let $G^{k+1})$ be the $k+1$-layer graph of v in G. The definitions of these v-invariants are given below:

ν_1^k(G,v) : total degree of V^k, namely, the sum of the degrees of the vertices in V^k;

ν_2^k(G,v) : the number of edges between LG^k and LG^{k+1}, i.e., the size of the set $E \bigcap V^k \times V^{k+1}$.

ν_3^k(G,v) : the number of vertices in the (k+1)-layer graph of v.

$\nu_4^k(G,v)$: the number of connected components in G^k.

Example 2.1: In figure 2.1, there are two automorphism groups in graph G. Vertices 1,2,3,4 belong to the same group. They all have vertex diameter 2 and the same ν values listed as follows.
$\nu_1^1 = 14, \nu_2^1 = 4, \nu_3^1 = 3, \nu_4^1 = 2, \nu_1^2 = 8, \nu_2^2 = 0, \nu_3^2 = 0, \nu_4^2 = 1;$
Vertex 5,6,7,8 belong to another group with vertex diameter 4 and their ν values are:
$\nu_1^1 = 8, \nu_2^1 = 4, \nu_3^1 = 4, \nu_4^1 = 1, \nu_1^2 = 12, \nu_2^2 = 2, \nu_3^2 = 1, \nu_4^2 = 1, \nu_1^3 = 2, \nu_2^3 = 0, \nu_3^3 = 0, \nu_4^3 = 0.$

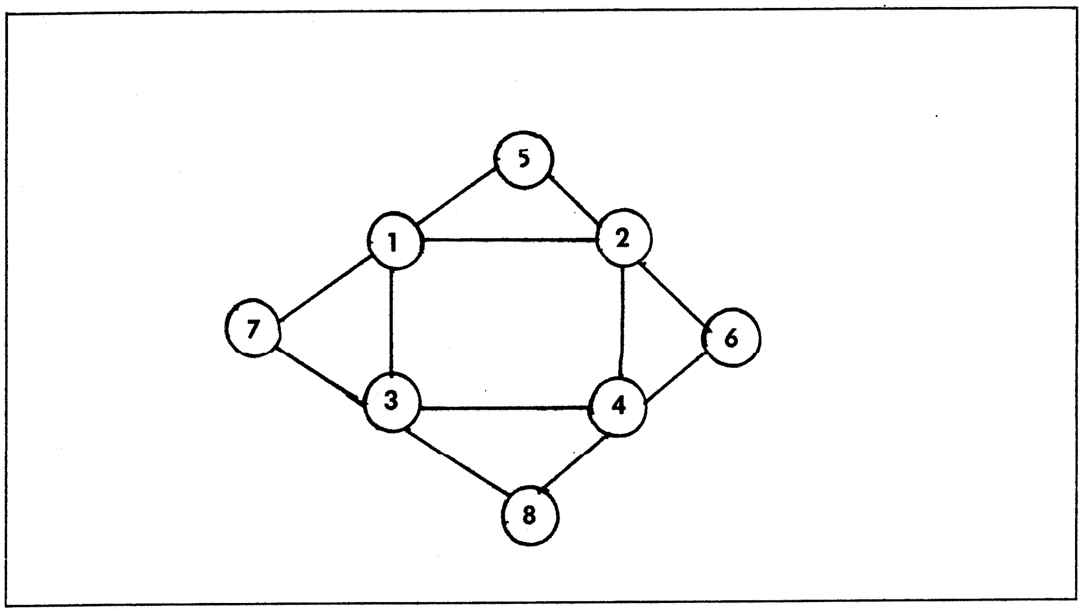

Figure 1.

Theorem 2.2: For all k, $\nu_1^k, \nu_2^k, \nu_3^k$ and ν_4^k are vertex invariants.

Proof:

Let vertex u and v be of the same automorphism partition in a graph G(V,E). Assume k is an arbitrary constant. By relabelling the vertices of G, we can get a graph G′(V′,E′) isomorphic to G and a vertex v′ isomorphic to v. Therefore v′ is also isomorphic to u.

So for $1 \leq i \leq< 4$, $\nu_i^k(G,u) = \nu_i^k(G',v') = \nu_i^k(G,v)$. ▯

Theorem 2.3: If $\rho_1, \rho_2, ..., \rho_n$ are v-invariants, and $\nu(G,v)$ is defined to be a sequence, $(\rho_1(G,v), \rho_2(G,v), ... , \rho_n(G,v))$. Then ν is also a v-invariant.

Proof:

Let u and v be of the same automorphism partition of graph G(V,E).

Thus, for $1 \leq i \leq n$, $\rho_i(G,u) = \rho_i(G,v)$.

Therefore $(\rho_1(G,v), \rho_2(G,v), ... ,\rho_n(G,v)) = (\rho_1(G,u), \rho_2(G,u), ... ,\rho_n(G,u))$, hence it is a v-invariant. ▯

Corollary: For any k,$\nu^k = (\nu_1^k, \nu_2^k, \nu_3^k, \nu_4^k)$ is a v-invariant. ▯

Let a v-invariant ν be formed by a sequence of v-invariants $\rho_1, \rho_2, ..., \rho_n$. ν is considered to be *redundant* if for any graph G, there exist two vertices u and v such that u and v are ν-distinguishable . They are then distinguishable under another v-invariant formed by a sequence of a proper subset of { $\rho_1, ..., \rho_n$.}

Theorem 2.4: $\nu = (\nu_1, \nu_2, \nu_3, \nu_4)$ is not redundant, where $\nu_i = (\nu_i^1, \nu_i^2, ..., \nu_i^m)$ and m is the vertex diameter of v in G.

Proof:

Assume ν is redundant, then there exists a ν_i of ν that is redundant, $1 \leq i \leq 4$.

Let $\nu 1 = (\nu 2, \nu 3, \nu 4)$. In figure 2.1,

$\nu 1$(G,u) = (8,6,2,4,2,2,0,0,1) = $\nu 1$((G,v).

However, ν_1(G,u) = (21,20,6) $\neq$ ν_1(G,v) = (19,22,6).

Therefore ν_1 is not redundant.

Let $\nu 2 = (\nu 1, \nu 3, \nu 4)$. In figure 2.2.

$\nu 2$(G,u) = (21,7,2,26,3,2,10,0,1) = $\nu 2$((G,v).

However, ν_2(G,u) = (8,6,0) $\neq$ ν_2(G,v) = (10,4,0).

Therefore ν_2 is not redundant.

Let $\nu 3 = (\nu 1, \nu 2, \nu 4)$. In figure 2.3.

$\nu 3$(G,u) = (19,8,2,24,4,2,10,0,1) = $\nu 3$((G,v).

However, ν_3(G,u) = (8,3,0) $\neq$ ν_3(G,v) = (7,4,0).

Therefore ν_3 is not redundant.

Let $\nu 4 = (\nu 1, \nu 2, \nu 3)$. In figure 2.4.

$\nu 4$(G,u) = (19,8,8,24,4,4,10,0,0) = $\nu 4$((G,v).

However, ν_4(G,u) = (2,3,1) $\neq$ ν_4(G,v) = (2,2,2).

Therefore ν_4 is not redundant.

Since none of $\nu_1, \nu_2, \nu_3, \nu_4$ is redundant, ν is not redundant. ∎

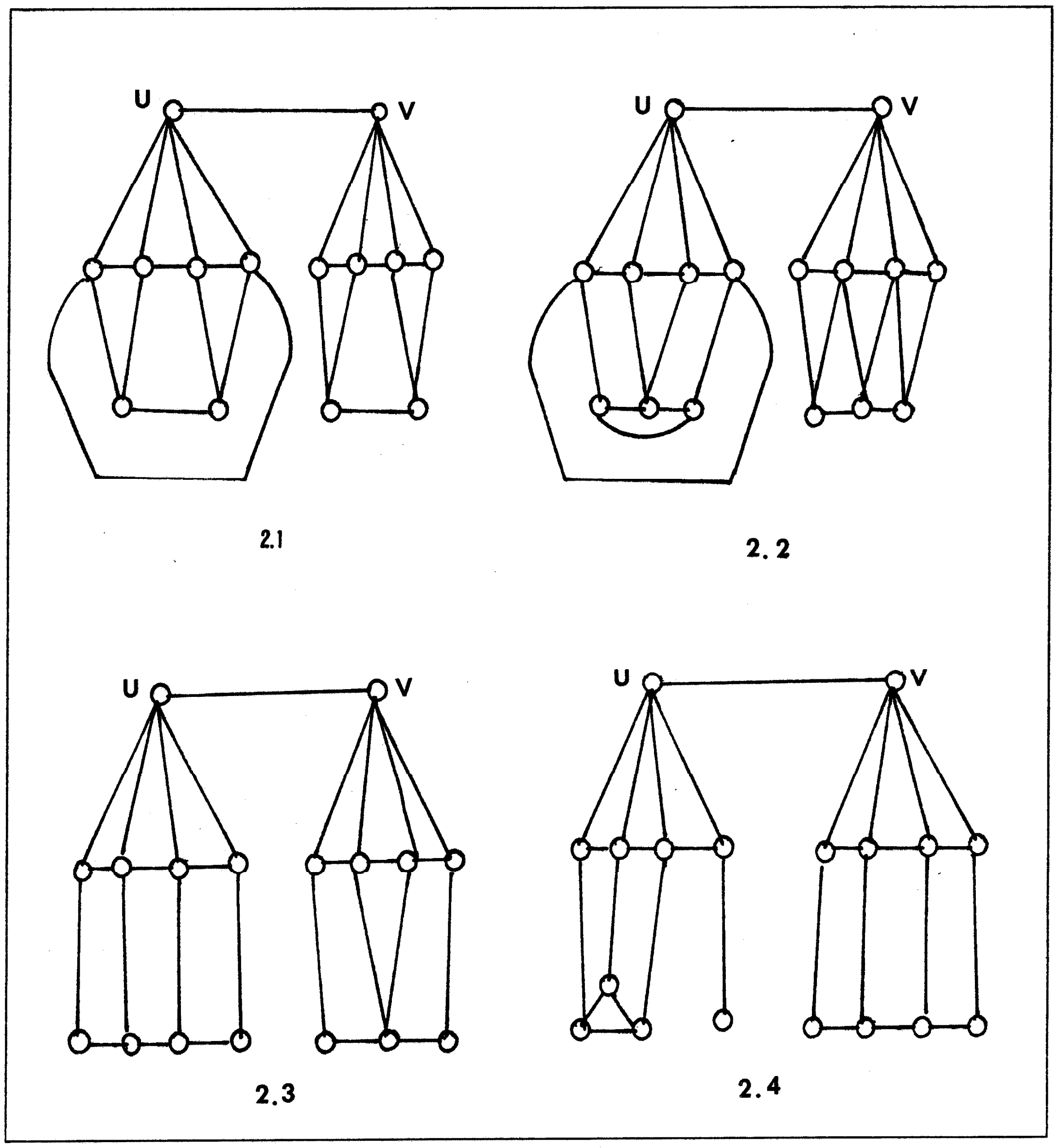

Figure 2.

V-invariants are often used for testing graph automorphism which is closely related to the problem of testing graph isomorphism. Unfortunately, they are not readily available for testing graph isomorphism. In general, putting the v-invariant values in a sequence will not give a g-invariant since it depends on how a graph is labelled. However, v-invariants in an ascending sequence are one of the few exceptions. For a graph G with vertex $v_1, v_2, ..., v_n$ and a v-invariant ν, an ascending sequence of v-invariant ν is a sequence of v-invariant values $\nu(v_1), \nu(v_2), ..., \nu(v_n)$ sorted in ascending order.

Theorem 2.5:An ascending sequence of a v-invariant is a g-invariant.

proof:

Let ν be a vertex invariants. Graph G(V_1, E_1) is isomorphic to graph G$'(V_2, E_2)$. $|V_1| = |V_2| = n$. vertex $u_1, u_2, ...u_n \in V_1$, $v_1, v_2, ..., v_n \in V_2$, and if $i \leq j$, then ν(G,v_i) $\leq$ ν(G,v_j), ν(G,u_i) $\leq$ ν(G,u_j).

Hence, g-invariant $\mathcal{I}$(G) $= (\nu(G, u_1), \nu(G, u_2), ..., \nu(G, u_n))$ and $\mathcal{I}$(G$'$) $= (\nu$(G$'$, v_1),ν(G$'$, v_2),...,ν(G$'$, v_n)). Assume $\mathcal{I}$(G) $\neq$ $\mathcal{I}$(G$'$). Thus there exist some mismatch between $\mathcal{I}$(G) and $\mathcal{I}$(G$'$). Let k be the smallest number i such that ν(G,u_i) $\neq$ ν(G$'$,v_i). Without loss of generality, assume ν(G,u_k) $<$ ν(G$'$,v_k). Since G is isomorphic to G$'$, there is a smallest $j < k$ such that ν(G,u_k) $=$ ν(G$'$,v_j) $=$ ν(G$'$,u_j). Consequently, there are $(k - j)$ vertices in G with v-invariants equal to ν(G,u_j), at the same time there are fewer vertices in G$'$ with v-invariants equal to this value. This violates that G and G$'$ are isomorphic. Therefore, this ascending sequence is a g-invariant. ▯

3 Graph Coding and Isomorphism Testing

For a vertex v in graph G, let ν(G,v) = (degree(v),ν^1, ν^2, ..., ν^m), where degree(v) is the degree of v, ν^ks as defined in section 2, and m is the vertex diameter VD(G,v). Thus, ν(G,v) is also a v-invariant due to theorem 2.3. From theorem 2.5, we can then form a g-invariant $\mathcal{I}$ by a sorted sequence of ν(G,v) which are sorted in an lexicographically ascending order. $\mathcal{I}$ and ν(G,v) are used in our graph isomorphism testing algorithm.

Example 3.1: For graph G in fig. 3, its v-invariants are:
$\nu(G,v_1)$: (22, 9, 5, 2, 24, 3, 1, 1, 3, 0, 0, 1)
$\nu(G,v_2)$: (21, 9, 6, 1, 25, 4, 1, 2, 4, 0, 0, 1)
$\nu(G,v_3)$: (28, 10, 5, 1, 20, 0, 0, 1)
$\nu(G,v_4)$: (25, 12, 6, 2, 24, 0, 0, 1)
$\nu(G,v_5)$: (20, 10, 5, 1, 22, 6, 2, 2, 8, 0, 0, 1)
$\nu(G,v_6)$: (21, 12, 5, 3, 24, 4, 1, 1, 4, 0, 0, 1)
$\nu(G,v_7)$: (27, 11, 5, 1, 21, 0, 0, 1)
$\nu(G,v_8)$: (21, 9, 6, 1, 26, 3, 1, 1, 3, 0, 0, 1)
$\nu(G,v_9)$: (23, 12, 6, 2, 26, 0, 0, 1)
$\nu(G,v_{10})$: (15, 8, 6, 1, 27, 7, 2, 2, 9, 0, 0, 1)
$\nu(G,v_{11})$: (17, 9, 6, 2, 29, 4, 1, 1, 4, 0, 0, 1)
$\nu(G,v_{12})$: (14, 11, 7, 3, 33, 4, 1, 1, 4, 0, 0, 1)

The g-invariant $\mathcal{I}$(G) is: (3, $\nu(G,v_{12})$, 3, $\nu(G,v_{10})$, 4 ,$\nu(G,v_{11})$, 4, $\nu(G,v_5)$, 4, $\nu(G,v_2)$, 4, $\nu(G,v_8)$,
5, $\nu(G,v_6)$, 5, $\nu(G,v_1)$, 5, $\nu(G,v_9)$, 5, $\nu(G,v_4)$, 6, $\nu(G,v_7)$, 6, $\nu(G,v_3)$)

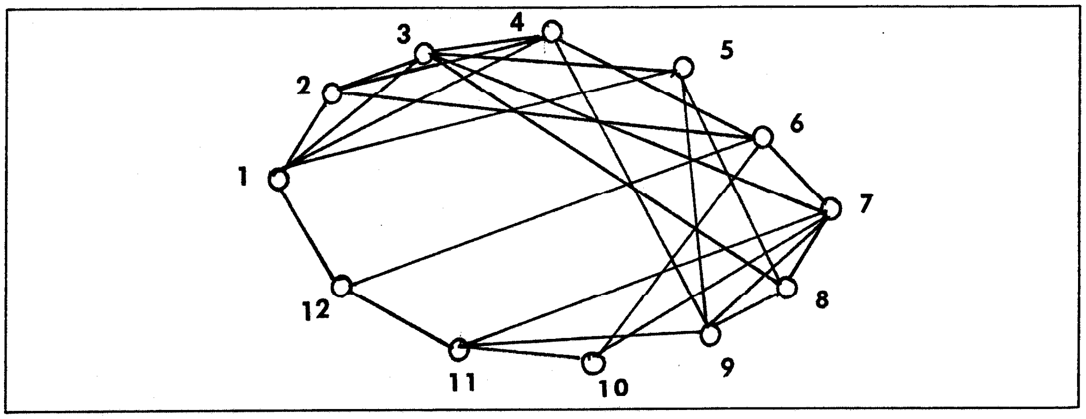

Figure 3.

ALGORITHM GITestA:

INPUT: Two graphs G1(V1,E1) and G2(V2,E2), their vertex adjacency list.

OUTPUT: Partitions P1 of V1 and P2 of V2. Two vertices of a graph are in the same partition if they have same v-invariants and similar connections. Also, an answer either *YES*, when G1 and G2 are isomorphic to each other; *NO*, when they are not; or *SIMILAR* when GITestA can not tell.

GIA1: Craph coding and vertex partitioning

Use algorithm GC and VCA to get v-invariants, g-invariants and partition vertices by their v-invariants.

GIA2: Comparing codes

Compare g-invariants of both graphs and the v-invariants as well as size of each partition, if they fail to match, answer *NO* and terminate.

GIA3: Refining partitions

Refine the partitions obtained in GIA2 by the connections of each vertex. Check if the refining of these two graphs are the same or not. Answer *NO* and terminate if they are not the same.

GIA4: Matching graphs

Try to match graph G1 and G2. If there were some partitions containing more than one vertex, a random match of a pair of vertices (u,v) will be selected with u from G and v from the corresponding partition in G2.

ALGORITHM VCA (Vertex Coding A):

INPUT: vertex adjacency list of graph $G(V,E)$ and a vertex v in V.

OUTPUT: vertex code $\nu(G,v)$.

```
VC1:    $\nu(1) \leftarrow$ degree(v);
          i $\leftarrow$ 2;
VC2:    'current layer set' C $\leftarrow \emptyset$;
          'next layer set N' $\leftarrow \emptyset$;
          'scanned  set' S $\leftarrow \emptyset$;
VC3:    'layer number' k $\leftarrow$ 0;
VC4:    mark(v) $\leftarrow$ 1;
          C $\leftarrow$ { v };
          N $\leftarrow$ { v's neighbors }
          S $\leftarrow$ C;
VC5:    while N not empty do
VC6:      begin
              k $\leftarrow$ k+1;
```

```
            $\nu_1^k \leftarrow 0$;
            $\nu_2^k \leftarrow 0$;
            $\nu_3^k \leftarrow 0$;
            $\nu_4^k \leftarrow 0$;
VC7:        C ← N;
            N ← Ø;
            S ← S $\bigcup$ C;
            for vertex v in C do mark(v) ← 0;
VC8:        while there is a vertex u in C and mark(u)=0 do
VC9:          begin
                C′ ← {u};
VC10:             while there is a vertex u in C′ and mark(u)=0 do

VC11:             begin
                    mark(u) ← 1;
VC12:                   $\nu_1^k \leftarrow \nu_1^k$+degree(u);
VC13:                   N′ ← neighbors(u) $\bigcap$ (V - S);
VC14:                   C′ ← C′ $\bigcup$ (neighbors(u) $\bigcap$ {unmarked
                    vertices in S});
VC15:                   $\nu_2^k \leftarrow \nu_2^k + |N'|$;
VC16:                   N ← N $\bigcup$ N′;
VC17:               end
VC18:             $\nu_4^k \leftarrow \nu_4^k + 1$;
VC19:       end
VC20:       $\nu_3^k \leftarrow |N|$;
VC21:       $\nu(i) \leftarrow \nu_1^k$
          $\nu(i+1) \leftarrow \nu_2^k$;
          $\nu(i+2) \leftarrow \nu_3^k$;
          $\nu(i+3) \leftarrow \nu_4^k$;
```

```
VC22:         $i \leftarrow i + 4$;
VC23:     end.
```

Algorithm GC gets the g-invariant for the input graph and partitions the vertices of the input graph according to their v-invariants. The lexicographical sorting at step GC6 using the algorithm 3.2 is from [AHU74]. Let l_i be the length of A[i]=($a_{i1}, a_{i2}, ..., a_{il_i}$), and let l_{max} be the largest of the l_i's. Each component of A[i] is in the range of 0 to |E|.

```
ALGORITHM GC (Graph Coding and Vertex Partition):
INPUT:vertex adjacency list of a graph $G(V,E)$.
OUTPUT: g-invariant  $\mathcal{I}$(G,V).
GC1:     clear array A, pointer k $\leftarrow$ 0;
GC2:     while V is not empty do
         begin
GC2:         get a vertex v from V;
GC3:         k $\leftarrow$ k+1;
GC4:         A(k) $\leftarrow$ $\nu(G,v)$ by employing algorithm VCA;
GC5:     end;
GC6:     (lexicographically sort A in ascending sequence.)
         clear queue Q.
         for $j$=0 to |E|-1 do clear bucket B[$j$];
         for $l$=$l_max$ step -1 to 1 do
         begin
```

```
          concatenate C[$l$] to the beginning of Q;
          (C[$l$] consists of all elements A[$i$] of length l)
          while Q not empty do
          begin
               let A[$i$] be the first element in Q;
               move A[$i$] from Q to B[$a_{il}$];
          end;
          for each $j$ on NONEMPTY[$l$] do
          (NONEMPTY[$l$] lists buckets which are occupied at pass $l$)
          begin
               concatenate B[$j$] to the end of Q;
               clear B[$j$];
          end;
     end;
GC7: i ← 1;
     while Q not empty do
     begin
          let PAR[i] be the first element in Q;
          if i=1 then
          begin
               N(i) ← 1;
               i ← i+1;
          end else
          begin
               if PAR[i]=PAR[i-1] then N(i) ← N(i)+1;
               else
               begin
                    i ← i+1;
                    N(i) ← 1;
```

```
            end;
        end;
    end;
    TOTAL ← i-1;
    (for vertices with same v-invariants, merge them into a partition.
     TOTAL is the total number of partitions and p(i) is the number
     of elements in partition i.)
```

```
ALGORITHM CC (Comparing Codes)
INPUT:  two g-invariants $\mathcal{I}(G_1)$ and $\mathcal{I}(G_2)$, and partitions
        $P_1$=(PAR1[1],PAR1[2],...,PAR1[$TOTAL_1$]);
        $P_2$=(PAR2[1],PAR2[2],...,PAR2[$TOTAL_1$]);
        N1(j) is the number of elements in partition j of graph $G_1$,
        N2(j) is the number of elements in partition j of graph $G_2$.
        $TOTAL_1$ is the number of partitions of $G_1$,
        $TOTAL_2$ is the number of partitions of $G_2$.
OUTPUT: NO: if they are not the same.
        YES: if they are the same.
CC1:    if $TOTAL_1 \neq TOTAL_2$ then answer NO and terminate;
        ( G1 is not isomorphic to G2 )
CC2:      for i ← 1 to $TOTAL_1$ do begin
CC3:          if PAR1(i) $\neq$ PAR2(i) or N1(i) $\neq$ N2(i) then
CC4:          answer NO and terminate;
CC5:      end;
CC6:      answer YES.
```

The next algorithm, RP(refining partitions), is a modification of an algorithm proposed by Hopcroft[Ho71] for minimizing states in a finite machine. Kubo, et. al.[Ku79] has applied this algorithm for automorphism partitions and here we modify it to serve our graph isomorphism testing.

```
ALGORITHM RP (Refining Partitions):
INPUT: two graphs G1(V1,E1),G2(V2,E2) represented by their vertex
       adjacency list A1($v_1$) and A2($v_2$) for all $v_1$ in G1 and
       $v_2$ in G2, and initial partitions P1, P2.
       P1={PAR1[1],PAR1[2],...,PAR1[TOTAL$_1$]}.
       P2={PAR2[1],PAR2[2],...,PAR2[TOTAL$_2$]}.
OUTPUT: Refining partition P1 and P2 according to A1 and A2. After
        each refining step, check if G1 and G2 have the same refining.
        If not, they are not isomorphic, answer NO and terminate.
        Otherwise, refine these partitions as much as possible.
RP1:    t1 $\leftarrow$ TOTAL$_1$;
        t2 $\leftarrow$ TOTAL$_2$;
        put P1 in waiting list W1=(WAIT1[1],WAIT1[2],...,WAIT1[t1]);
        put P2 in waiting list W2=(WAIT2[1],WAIT2[2],...,WAIT2[t1]);
RP2:    while W1 is not empty do
        begin
RP3:        get and delete the first vertex set WAIT1[1] from W1;
            get and delete the first vertex set WAIT2[1] from W2;
RP4:        N1 $\leftarrow$ neighbors of WAIT1[1];
RP5:        N2 $\leftarrow$ neighbors of WAIT2[1];
RP6:        for each vertex u in N1 do
```

```
            adj(u) ← the number of u's neighbors in WAIT1[1];
        for each vertex u in N2 do
            adj(u) ← the number of u's neighbors in WAIT2[1];
RP7:    for each j such that WAIT1[j] $\bigcap$ N1 $\neq \emptyset$ do
        begin
RP8:        clear all the buckets B1[i];
RP9:        for each u $\in$ WAIT1[j] do
            begin
                if adj(u)$\neq$0 then put u in bucket B1[adj(u)]
                else put u in B1[0];
            end;
RP10:       clear all the buckets B2[i];
RP11:       for each u $\in$ WAIT2[j] do
            begin
                if adj(u)$\neq$0 then put u in bucket B2[adj(u)]
                else put u in B2[0];
            end;
RP12:       LARGEST ← 1;
            LSIZE ← |B1[1]|;
            for i=1 to |V1| do
            begin
                if |B1[i]| $\neq$ |B2[i]| then
                G1 not isomorphic to G2, terminate;
                if |B1[i]| > LSIZE then
                begin
                    LARGEST ← i;
                    LSIZE ← |B1[i]|;
                end;
            end;
```

```
RP13:             WAIT1[j] ← B1[LARGEST];
                  clear B1[LARGEST];
                  WAIT2[j] ← B2[LARGEST];
                  clear B2[LARGEST];
                  for i=1 to |V1| do
                  begin
                      if |B1[i]| ≠ 0 then
                      begin
                          t1 ← t1+1;
                          WAIT1[t1] ← B1[i];
                      end;
                  end;
                  for i=1 to |V2| do
                  begin
                      if |B2[i]| ≠ 0 then
                      begin
                          t2 ← t2+1;
                          WAIT1[t2] ← B2[i];
                      end;
                  end;
RP14:         end;
RP15:     end;
```

AlGORITHM MG (Matching Graphs):

INPUT: partitions $V^1, V^2, ...V^p$ of G and corresponding partition

```
        $U^1, U^2, ...., U^p$ of G2.
OUTPUT: YES: if matching succeeded.
        NO:  if matching failed and p=n.
        SIMILAR: if matching failed and $p \neq n$.
MG1:  while there are vertices not matched do begin
MG2:      i ← 1;
          q ← p;
MG3:      if $U^i$ and $V^i$ are single element partitions, match the
          corresponding vertices contained in them else begin
MG4:          randomly select and match a vertex v from $V^i$ and
              a vertex u from $U^i$;
MG5:          refine partition $U^i$ to $U^q$ as much as possible
              by separating u from $U^i$;
MG6:          refine partition $V^i$ to $V^q$ as much as possible
              by separating v from $V^i$;
MG7:          modify q;
MG8:      end;
MG9:  end
MG10: check the match of G1 and G2 according to the connections of
      each vertex. If this checking succeeded then answer YES else if
      p=n then answer SIMILAR else answer NO;
```

Next we analyze the above algorithms.

Theorem 3.1: Algorithm VCA output the v-invariant $\nu(,G,v)$ for vertex v of graph G with time complexity O(|E|) and space complexity O(|V|).

Proof:

We prove VCA output the correct ν(G,v) by induction on the layer number k. First, we will prove that for any k at step VC5, a)C contains all the vertices of the current layer, b)N contains all the vertices of the next layer and c)S contains all the vertices of current and upper layers. For k=0, C contains vertex v, which is the only vertex at layer 0. N contains all the neighbors of v, namely, the vertices of layer 1, and P contains only vertex v since there is no upper layer of layer 0. If for k=n the induction hypothesis is correct at VC5. At step VC7 N is assigned to C, and P is set to be the union of C and P. C and P will not be changed until it reaches VC5 again. Hence, for k=n+1, both C and P contain correct members. Upon reaching step VC7, we will clear N. At step VC20, N is the union of N′ of each vertex in C. Step VC8 guarantees that every vertex in C will go through the loop from VC10 to VC18 since it can only be marked at step VC11. VC14 expands set C′, (the connected subgraph of C), and makes sure each vertex in C can go through the loop only once by limiting the expansion of C′. Now every vertex in C will go through the inner loop exactly once and generate its next layer neighbor N′, so finally at step VC20 we will have a correct next layer set N. The induction is therefore complete.

Now we will prove that for all k at step VC20, we will have the correct ν values. Since at VC5 we have correct values of C, N and S, we will also have correct values of C and S at VC7. We mentioned in the previous paragraph that every vertex in C will go through the loop from VC11 to VC 17 exactly once. So at step VC20, ν_1^k is the total degrees of the vertices of C. ν_4^k at step VC18 counts how many C′ there are for current C. (C′ is the connected subgraph of C.) N′ is the number of edges from each vertex u in C to N and ν_2^k is the sum of them. Finally, ν_3^k gives the size of set N, i.e., the number of vertices in N. Consequently, VCA gives the correct v-invariant ν(G,v).

Step VC1 to VC4 will take only constant time. For the loop from VC5 to VC23, every vertex in G will pass through only once. This is also true for the inner loop VC10 to VC17 due to the fact that every vertex will be marked exactly once. At step VC13 and VC14 we need to check the neighbors of each vertex, so for the whole loop we need to check |E| times VCA now has time complexity O(|E|). *The total space needed to store the ν values is of the order of the vertex* diameter of v.

Algorithm GC (graph coding) employs VCA for every vertex in G and sorts these v-invariants in ascending order. The total time for getting v-invariants is O(|V| $\times$ |E|) *The average vertex-diameters of G is of order* O(|V|) *and the total* length of $\mathcal{I}$(G) is of order O(|V|^2). The range of ν_i^k is also of order O(|V|^2). The lexicographic sort of these v-invariants would require O(|V|^2) following the result of Theorem 3.2 of Aho, Hopcroft and Ullman [Ah74]. Therefore, the time complexity for GC is O(|V| $\times$ |E|). The space requirement for GC is O(|V|^2).

Algorithm CC (comparing codes) will take O(|V|) steps since the number of partitions is of order O(|V|). Algorithm RP (Refining Partitions) is a modification of the algorithm proposed by Hopcroft and used by Kubo et. al. By the proof of Hopcroft and Kubo et. al., this algorithm will generate correct partitions in O(|E|log|V|). Finally, the algorithm MG(matching graphs) is also of time complexity O(|E|log|V|) since it needs to check the connections of every vertex in G.

From theorem 3.1 and previous analysis, GITestA generates g-invariants for two graphs, partitions them by the v-invariants, refines the partitions according to the connections of each vertex, and compares the results to decide if they are isomorphic. The time complexity of GITestA is O(|V|$\times$| E|) and the space complexity is O(|V|^2).

Example 3.2 For graph G1,G2 and G3 in figure 4.2. their g-invariants are listed as follows:

$\mathcal{I}$(G1):(4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1)

$\mathcal{I}$(G2)=$\mathcal{I}$(G1).

$\mathcal{I}$(G3)=(4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 6, 2, 1, 8, 0, 0, 1, 4, 16, 8, 2, 2, 8, 0, 0, 2, 4, 16, 8, 2, 2, 8, 0, 0, 2, 4, 16, 8, 2, 2, 8, 0, 0, 2)

In fact, GITestA generates (7, 4, 16, 6, 2, 1, 8, 0, 0, 1) for graph G1 and G2, and generates (4, 4, 16, 6, 2, 1, 8, 0, 0, 1, 3, 4, 16, 8, 2, 2, 8, 0, 0, 2) for graph G3. G1 is isomorphic to G2 and each of them has an automorphism partition with 7 elements. G3 has two automorphism partitions, one with 3 elements and another with four elements. G3 is not isomorphic to G1 or G2.

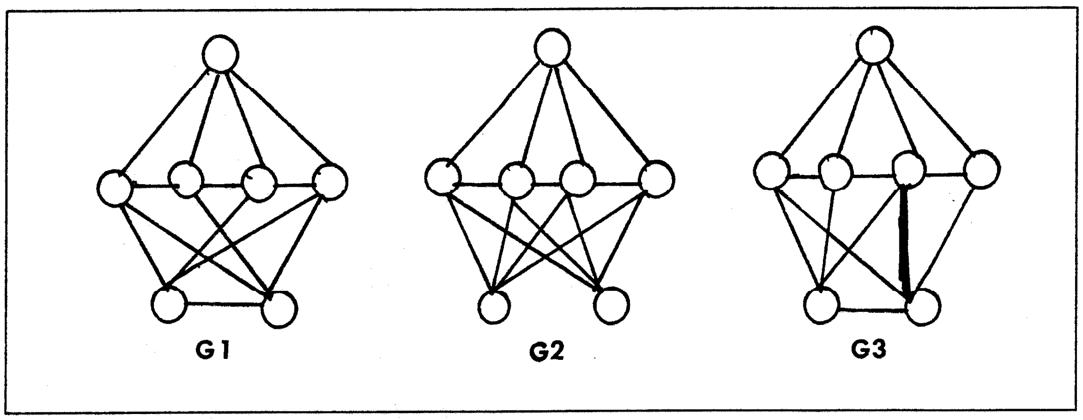

Figure 4.

Primitive graph isomorphism testing algorithms use only degree-sequences as the graph invariants. Corneil and Gotlieb [Co70] employ further partitioning to classify the vertices of a graph in $O(|V|^3)$. Schmidt and Druffel [Sc76] partition the vertices according to the distance characteristic matrix of a graph also in $O(|V|^3)$. Any two graphs distinguishable by the previous algorithms can be differentiated by GITestA in $O(|V| \times |E|)$. Independently, Bhat [Bh80] developed an algorithm using fixed length code of the same complexity of GITest1. However, GITestA can tell the difference between the two strongly regular (16,6,2,2) graphs, even though they are similar in Bhat's fixed length code algorithm. They are also similar in Bhat's refined fixed length code algorithm which is of higher complexity than $O(|V|^3)$.

While we haven't found any two non-strongly-regular or non-BIBD graphs that are similar to GITest1, there are many strongly regular and BIBD graphs which are resistant to it. Example graphs are BIBD(10,4,2) graphs and strongly regular (25,12,5,6) graphs which will be discussed in section 5.

4. A Fast GITest Algorithm

There are linear graph isomorphism testing algorithms for several special classes of graphs such as trees [Ah74], planar graphs [Ho74], maximum outplanar graphs [Be79], and interval graphs [Co81]. For use on general graphs, most practical graph isomorphism testing algorithms are based on heuristics. Kubo, Shirakawa, and Ozaki [Ku79] developed a fast GITest algorithm based on Hopcroft's partition algorithm [Ho71]. For a graph G(V,E), their algorithm is of time complexity $O(|E|\log|V|)$. This algorithm is among the fastest GItest algorithms currently in use for general graphs. However, the initial partition in this algorithm is only based on the degree of each vertex. Therefore, it cannot distinguish between any two regular graphs of the same degree such as the ones in figure 5. Moreover, it also has problems in telling the difference between two graphs with many vertices of the same degree which is unfortunate since these kind of graphs occur quite frequently in practical applications.

Except for some strongly regular graphs and BIBD graphs, which are not very common for isomorphism testing, we have not found any two graphs which are not distinguishable by GITestA. Unfortunately, the time complexity $O(|V| \times |E|)$ and space complexity $O(|V|^2)$ make it unsuitable for testing isomorphism of graphs with many vertices.

Observation 1: The bottleneck of GITestA is graph coding and partition GC.

In GC, it takes $O(|V|^2)$ steps to do the lexicographic sort and it takes $O(|V|^2)$ registers to store all the v-invariants. A hashing scheme would reduce the space requirement to $O(|V|)$ and speed up the sorting to $O(|V|\log|V|)$. GC4 is modified to be:

GC4': G(k):= Hash($\nu(G, v)$,|V|);

Additionally, instead of using lexicographic sort in GC6, a quick sort or heap sort will be used. A simple hashing algorithm is shown below:

```
ALGORITHM HA(hashing):
INPUT: a string of numbers $\nu(G, v)$:(a1,a2,...,an) and |V| of G.
OUTPUT: a hashed number h in the range of 1 to k $\times$ |V| where k
        is an arbitrary constant.
HA1:    a:=a1 + a2 + ... + an;
HA2:    h:=mod(a, 2 $\times$ |V|) + 1;
```

The advantages of using hashing in GITestA is:

1. save space, (this is the major advantage,) and reduce space requirement from O($|V|^2$) to O(|V|).
2. save sort time from O($|V|^2$) to O(|V|).
3. easy implementation, since we don't have to worry about hashing collision.

However, even using hashing in GITestA, the time complexity is still O($|V| \times |E|$) since the time complexity of VC is O(|E|).

Observation 2: most vertices distinguishable by $\nu(G, v)$ are also distinguishable by (degree(v),ν^1(G,v)).

Observation 3: in graph G(V,E), if a vertex v is connected to all the other vertices in G then v's connection would not provide any information to distinguish its neighbors.

Instead of using ν, a new vertex invariant υ(G,v) will be used for vertex coding. Let $LG^1(V^1, E^1)$ be the 1-layer graph of a vertex v in G, and Let $LG^2(V^2, E^2)$ be the 2-layer graph of a vertex v in G. k is a small constant, and V′ is the set of vertices in V^1 whose degree is no larger than k. LG′(V′,E′) is a subgraph of LG^1 where E′ is the intersection of V′ $\times$ V′ and E^1. Five v-invariants υ_1, υ_2, υ_3, υ_4, and υ_5 are used in our new algorithm, GITestB, and their definitions are as follows:

υ_1(G,v) : degree of v;

υ_2(G,v) : if degree of v $>$ k, then set this to be 0, otherwise set it to be total degree of V^1, namely, the sum of the degrees of the vertices in V^1;

υ_3 : if degree of v $>$ k, then set this to be 0, otherwise set it to be the number of edges from LG′ to the 2-layer graph of v, $LG^2(V^2, E^2)$, i.e., the size of the set E ;

υ_4 : if degree of v $>$ k, then set this to be 0, otherwise set it to be the number of vertices adjacent to V′ in the 2-layer graph of v.

υ_5 : if degree of v $>$ k, then set this to be 0, otherwise set it to be the number of connected components in LG′.

Vertex invariant υ(G,v) is defined to be $(\upsilon_1, \upsilon_2, \upsilon_3, \upsilon_4, \upsilon_5)$ and graph invariant $\mathcal{I}$ is a lexicographically ascending sequence of υ(G,v). We modify VCA to be VCB to get a vertex invariant υ(G,v). GITestB and VCB are shown below.

ALGORITHM GITestB:

INPUT: Two graphs; G1(V1,E1) and G2(V2,E2), and their vertex adjacency

list.

OUTPUT: Partitions P1 of V1 and P2 of V2. Two vertices of a graph are in the same partition if they have same v-invariants and similar connections. Answer either *YES*, when G1 and G2 are isomorphic to each other; *NO*, when they are not; or *SIMILAR* when GITestB cannot tell.

GIB1: graph coding and vertex partitioning

Use algorithm GC and VCB to get v-invariants, g-invariants and partition vertices by their v-invariants.

GIB2: comparing codes

Compare g-invariants of both graphs and the v-invariants as well as size of each partition, if they fail to match, answer *NO* and terminate. Use algorithm CC in GITestA.

GIB3: refining partitions

Refine the partitions obtained in GIB2 by the connections of each vertex. Check if the refining of these two graphs are the same or not. Answer *NO* and terminate if they are not the same. Use algorithm RP in GITestA.

GIB4: Matching graphs

Try to match graph G1 and G2. If there were some partitions containing more than one vertex, a random match of a pair of vertices (u,v) will be selected with u from G and v from the corresponding partition in G2. Use algorithm MG in GITestA.

ALGORITHM VCB (Vertex Coding B):

INPUT: vertex adjacency list of graph $G(V,E)$, a vertex v in V and a constant k.

OUTPUT: vertex code $\upsilon(G,v)$.

```
VB1:  clear array $\upsilon$.
VB2:  $\upsilon(1) \leftarrow$ degree(v);
      $\upsilon(2) \leftarrow 0$;
      $\upsilon(3) \leftarrow 0$;
      $\upsilon(4) \leftarrow 0$;
      $\upsilon(5) \leftarrow 0$;
VB3:  If degree(v) $>$ k then return;
VB4:  put v in 'scanned set' S;
      put v's neighbors in 'current layer set' C;
      'next layer set' N $\leftarrow \emptyset$;
VB5:  for all the vertices v in C do mark(v) $\leftarrow 0$;
VB6:  while there is a vertex u in C such that mark(u)=0 do
      begin
VB7:      $C' \leftarrow \{u\}$; ($C'$ is a connected component)
VB8:      clear D, subset of C with vertices of degree $>$ k;
VB9:      while there is a vertex u in $C'$ such that mark(u)=0 do
          begin
VB10:         $\upsilon(2) \leftarrow \upsilon(2)$+degree(u);
```

```
                mark(u) ← 1;
VB11:           if (degree(u) > k) then put u in D
                else begin
VB12:               N′ ← neighbors(u) ⋂ (V - S - C);
VB13:               C′ ← C′ ⋃ (neighbors(u) ⋂ {unmarked
                    vertices ∈ (C-D)});
VB14:               $\upsilon(3) \leftarrow \upsilon(3)+|N'|$;
VB15:               N ← N ⋃ N′;
VB16:           end;
VB17:       end ;
VB18:       $\upsilon(5) \leftarrow \upsilon(5)+1$;
VB19:   end;
VB20:   $\upsilon(4) \leftarrow |N|$;
```

Theorem 4.1: Assume vertex v of graph G(V,E) is of degree d and N is the set of v's neighbors. A vertex u in N is also in D, (a subset of N), if degree of u is less than k, where k is a small constant. Assume that the total degree of vertices in D is t. Algorithm VCB will output the v-invariant $\upsilon(G,v)$ for vertex v of graph G with time complexity O(t) and constant space complexity O(t).

Proof:

First we prove the correctness of algorithm VCB.

At step VB2 υ_1 is assigned the correct value degree(v). If υ_1 is larger than k then we also get correct values for $\upsilon_2,\upsilon_3,\upsilon_4$, and υ_5. VCB will terminate at step VB3.

Otherwise, since any vertex that is u adjacent to vertex v is put into C at VB4 and mark(u) is set to 0 at VB5, u will be put into C′ at VB7 or VB13.

Hence, degree(u) will be added to υ_2 at VB10. Because we also mark u at VB10, degree(u) will be added to υ_2 exactly once. Therefore we will have correct value of υ_2. For each vertex u in C, (with degree(u) no larger than k), at VB12 N′ is valued the set of vertices in the next layer. In other words, $|N'|$ is the number of edges from u to the next layer graph. At VB15 N is set to be the union of N′ for all the vertices in C. Therefore at VB12 and VB20, we get correct values for υ_3,υ_4. In current layer set C, if two vertices u and v are in the same component, they will be put in the same C′ at VB13. At VB18 we count how many such C′ we have in C and that is the number of strong components in the current layer graph. Hence υ_5 is the correct value.
Next we prove for the complexity of VCB.
For space complexity, the main cost is the temporary storage for S, C, N etc., N being the biggest set among them. The size of N is at most t when no two vertices in C have common neighbors in the next layer set N. Therefore, the space complexity is of O(t).
For time complexity, the main loop is from VB6 to VB19, and the most costly time step is VB12 and VB13. To calculate VB12 and VB13 for a vertex u, information about u's neighbors is required. u has at most k neighbors and we need only concern ourselves with those neighbors of degree less than k. Therefore the time complexity is of the order of the total degrees of these vertices, namely, O(t). ∎

Because the O(t) space requirement for VCB is only temporary storage and can be reused for all the vertices in V , to get v-invariants for all the vertices in V would actually require space $O(|V|+t)$ which is of linear order of the input. If a vertex has degree d and $d < k$, then it will be scanned at most d^2 times. The summation of all the d^2 over V would be smaller than $k^2 \times |V|$, and the algorithm GC would take $O(k^2 \times |V|)$. Therefore, GITestB would have time complexity $O(|E|\log|V| + k^2 \times |V|)$ since algorithm RP is of complexity O(|E|log

$|V|)$.

The choice of k may be dependent on the input graph. For practical applications, almost all the graphs we are concerned with are very sparse, namely, $|E|$is of the order of $|V|$. Therefore we may set $k = \lceil |E| / |V| \rceil$ and the time complexity of GITestB would be $O(|E|^2 / |V| + |E| \log |V|)$.

GITestB works very well in testing graphs distinguishable to GITestA. However, by using some graph transform we can contrive graphs which are distinguishable to GITestA but not GITestB. Given a graph G, a k-subdivision graph of G $S^k(G)$, $k \geq 0$, is transformed from k by placing k new vertices on each edge of G. In fig.5, G1′ and G2′ are the 1-subdivision graphs of G1 and G2 respectively. G1 and G2 are distinguishable to both GITestA and GITestB. G1′ and G2′ are only distinguishable to GITestA.

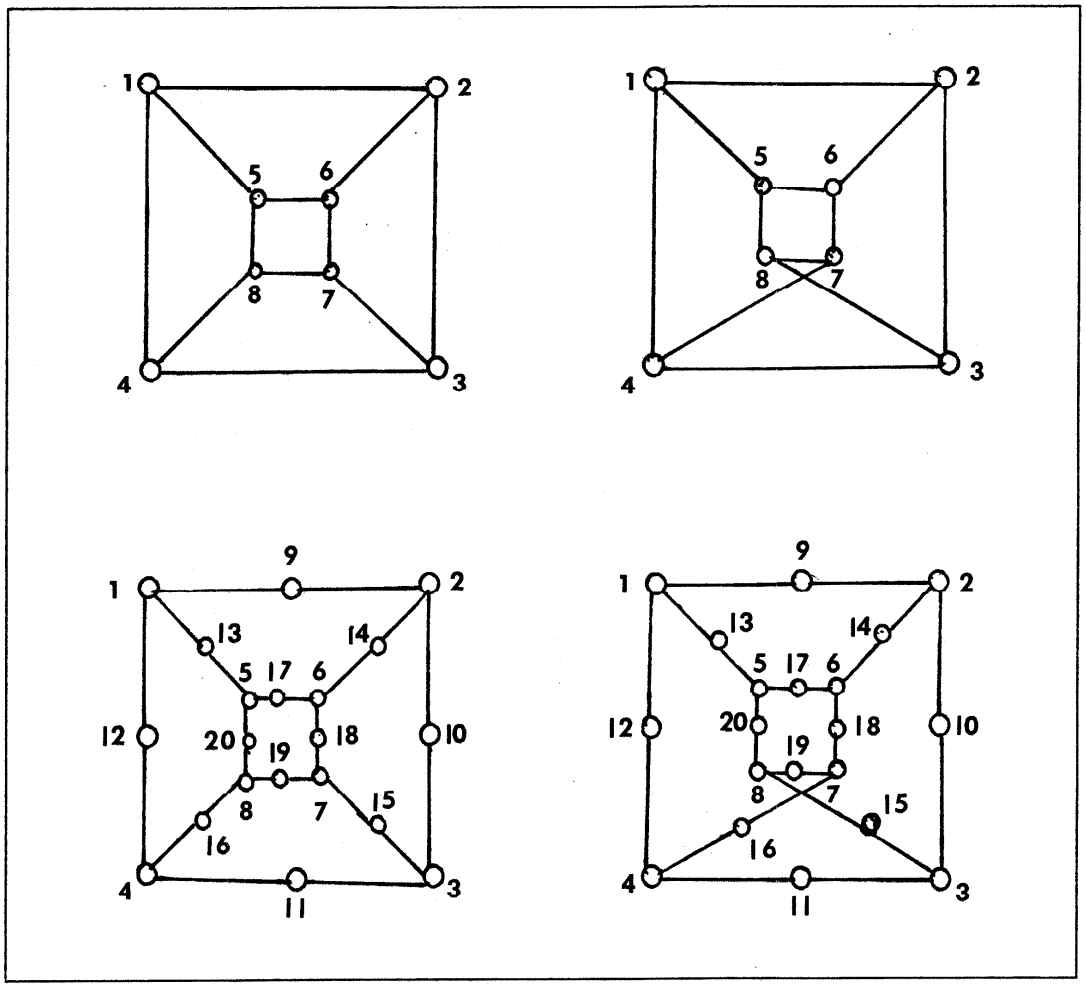

Figure 5.

Example 4.1 In figure 5,

Both GITestA and GITestB partition graph G1 and G2 as follows:

Partitions of G1:((1,2,3,4,5,6,7,8))

Partitions of G2:((1),(2),(3),(4),(5),(6),(7),(8))

So both of them lead to the conclusion that G1 is not isomorphic to G2.

For graph G1′, both GITestA and GITestB generates the following partition:((1,2,3,4,5,6,7,8),(9,10,11,12,13,14,15,16,17,18,19,20))
For graph G2′, GITestB generates the same g-invariants and partitions as for G1′; it cannot distinguish G2' from G1′.
GITestA Generates a different g-invariant and partition for G2′
((1), (2), (3), (4), (5), (6), (7), (8), (9), (10),(11), (12),
(13), (14), (15), (16), (17), (18), (19), (20))
It determines that G2′ is not isomorphic to G1′.

GITestB may be extended to include v-invariants ν^2(G,v) for the 2-layer graphs. In this way G1′ and G2′ can be distinguished by this new v-invariant. However, by using graph G1, G2 and the k-subdivision graph transform, we can show that there are always some graphs distinguishable to GITestA but not to the v-invariants (ν^1,...,ν^k) extension of GITestB.

Both GITestA and GITestB can be easily extended to test isomorphism of directed graphs. A straightforward approach would be to simply use the outward edges to define $forward\ k - layergraphs$ and the inward edges to define $backward\ k - layergraphs$. All the vertex invariants will be defined similarly.

5. Graph Transform and GITest Hierarchy

A graph G(V,E) with $|V| = n$ is a strongly regular (SR) graph if there exist three numbers k,λ, and μ such that G satisfies the following conditions:

1. G is of degree k, $0 > k \geq n-1$; namely, every vertex in G is of degree k.

2. Given any two distinct adjacent vertices u and v, the number of vertices adjacent to both u and v is λ.

3. Given any two distinct adjacent vertices u and v, the number of vertices adjacent to both u and v is μ.

The four integers (n,k,λ, μ) are called parameters of a SR graphs.

Example 5.1 Two non-isomorphic SR graphs of parameters (16,6,2,2) and their adjacency lists are shown below.

G1(V1,E1)

vertex	neighbors					
1	2	3	4	5	6	7
2	1	3	4	8	9	10
3	1	2	4	11	12	13
4	1	2	3	14	15	16
5	1	6	7	8	11	14
6	1	5	7	9	12	15
7	1	5	6	10	13	16
8	2	5	9	10	11	14

9	2	6	8	10	12	15
10	2	7	8	9	13	16
11	3	5	8	12	13	14
12	3	6	9	11	13	15
13	3	7	10	11	12	16
14	4	5	8	11	15	16
15	4	6	9	12	14	16
16	4	7	10	13	14	15

G2(V2,E2)

vertex	neighbors					
1	2	3	4	5	6	7
2	1	3	4	8	9	10
3	1	2	5	8	11	12
4	1	2	6	9	13	14
5	1	3	7	11	13	15
6	1	4	7	12	14	16
7	1	5	6	10	15	16
8	2	3	10	12	14	15
9	2	4	10	11	13	16
10	2	7	8	9	15	16
11	3	5	9	12	13	16
12	3	6	8	11	14	16
13	4	5	9	11	14	15
14	4	6	8	12	13	15
15	5	7	8	10	13	14
16	6	7	9	10	11	12

While G1 and G2 can be identified to be isomorphic by both GITestA and

GITestB, there is no other published graph isomorphism testing algorithm of time complexity less than $O(|V|^4)$ that can distinguish their nonisomorphism.

To generalize the concept of SR graphs, Mathon proposed an idea about k-regular graphs which is based on the k-adjacency partition (as defined in his paper [Ma78]). By definition, a k-level regular graph is m-level regular for any m, $1 \leq m \leq k$. A graph G is a 1-level regular graph if and only if it is an SR graph while a SR graph may also be a k-level regular graph for some $k \geq 1$.

A (b,v,r,k,λ)- Balanced Incomplete Block Design (BIBD) is an arrangement of v objects into b blocks so that:

1. each object appears in exactly r blocks;

2. each block contains exactly k objects, where $k < v$;

3. each pair of distinct objects appear together in exactly λ blocks.

The parameters v,b,r,k,λ are not independent. It is very easy to show that $vr = bk$, and $r(k-1) = \lambda(v-1)$. There are several other similar equations about BIBD parameters that can be found in combinatorics [Ry63]. A symmetrical BIBD is a BIBD with $b = v$ and $r = k$.

A graph G is said to be a BIBD graph if it is a bipartite graph with two vertex subset V1 and V2 so that:

1. $|V1| = v$, each vertex in V1 corresponding to an object in a BIBD and V1 corresponding to the set of objects.

2. $|V2| = b$, each vertex in V2 corresponding to a block in a BIBD and V2 corresponding to the set of blocks.

3. a vertex u in V1 is adjacent to a vertex in V2 if and only if the corresponding object of u is contained in the corresponding block of v.

Example 5.2 Two BIBD graphs G1 and G2 are shown below. They are not isomorphic, but they are of the same parameters (10,15,6,4,2).

G1(V1,E1), V1 = V1O $\bigcup$ V1B where V1O corresponds to the set of objects and V1B corresponds to the set of blocks.
V1O = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
V1B = 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
The following is the adjacency list for vertices in V1O.

vertex	neighbors					
1	11	12	13	14	15	16
2	11	12	17	18	19	20
3	11	13	17	21	22	23
4	11	14	18	21	24	25
5	12	13	19	22	24	25
6	12	15	20	21	23	24
7	13	16	18	20	23	25
8	14	15	17	19	23	25
9	14	16	17	20	22	24
10	15	16	18	19	21	22

G2(V2,E2), V2 = V2O $\bigcup$ V2B where V2O corresponds to the set of objects and V2B corresponds to the set of blocks.

V2O = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

V2B = 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,

The following is the adjacency list for vertices in V2O:

vertex	neighbors					
1	11	12	13	14	15	16
2	11	12	17	18	19	20
3	11	13	17	21	22	23
4	11	14	18	21	24	25
5	12	13	19	22	24	25
6	12	15	20	21	23	24
7	13	16	18	20	23	25
8	14	15	17	19	23	25
9	14	16	19	20	21	22
10	15	16	17	18	21	22

SR graphs and BIBD graphs are the most difficult graphs for isomorphism testing. Many graphs of these two classes are indistinguishable to both GITestA and GITestB. In the previous section, we showed that there exist graph transforms which make graphs more difficult to be distinguished. However, there are also graph transforms which make graphs more distinguishable.

A graph transform is called an invariant graph transform if it transforms a pair of isomorphic graphs to another pair of isomorphic graphs. Complement graph transform, which transforms a graph G to its complement G$'$, is an example of an invariant graph transform. The k-subdivision transform mentioned in the previous section is another example of invariant graph transform. We are only interested in invariant graph transform for testing graph isomorphism.

Fowler et al. developed a k-superline graph transform [FHG83] which is an invariant graph transform. A k-line graph $L^k(G)$ is defined as:

1. $L^0(G)=G$.
2. $L^1(G)=L(G)$, the line graph of G.
3. $L^k(G)=L^1(L^{k-1}(G))$.

k-superline graph transform $S^k(G)$ is a superimposition of the graphs $L^0(G)$,..., and $L^k(G)$. If graph G(V,E) has m vertices and n edges, $S^1(G)$ will have (m+n) vertices and $(n+(1/2)\left(\sum_{i\in V}(d_i)^2\right)$ where d_i is degree of vertex i. An example of superline graphs are in figure 6. $S^k(G)$ is used in our isomorphism testing hierarchy with lower level contains algorithms of lower complexity.

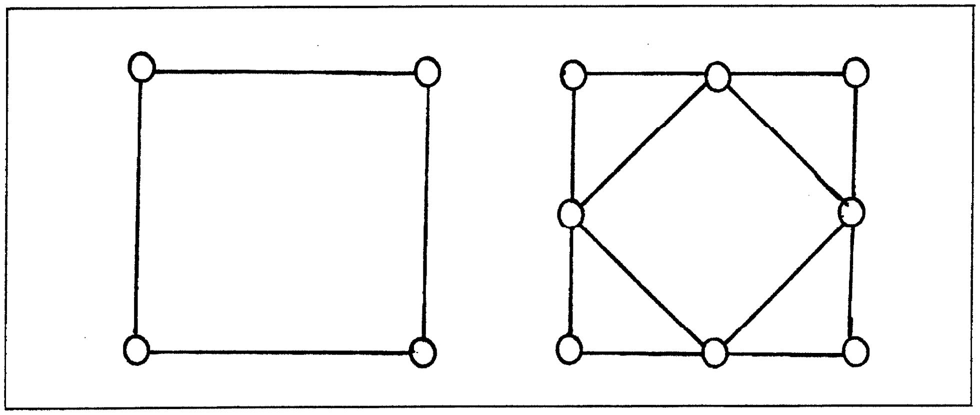

Figure 6.

ALGORITHM H

```
INPUT: Two graphs G1(V1,E1) and G2(V2,E2), and their vertex adjacency
        list.
OUTPUT: YES if G1 and G2 are isomorphic, otherwise NO.
H1:   GITestB(G1,G2).
H2:   terminate if answer is either YES or NO.
H3:   delete all the distinguishable vertices in G1 and G2.
H4:   apply 1-superline transform to G1 and G2.
H5:   GITestA(G1,G2), the initial partitioning is based on both
      the previous partition from lower hierarchy and the new
      v-invariants.
H6:   go to H2.
```

After a 1-superline transform of a graph G(V,E), the new graph $S^1(G)$ has $O(|E|)$ vertices and $O(|E|^2)$ edges in the worst case. And in the worst case , a 2-superline transformed graph $S^2(G)$ has $O(|E|^2)$ vertices and $O(|E|^4)$ edges. However, this only happens in very rare cases such as transforming a complete graph (which is incidentally very easy to test for isomorphism at step H1). And since all the distinguishable vertices are deleted at step H3, the transformed graph will be of much lower order than the worst case.

Most graphs can be tested in H1. Level 2 strongly regular graphs and BIBD graphs may require two superline transforms. We have not found any pair of graphs takes more than two superline transforms. A 3-level strongly regular graphs may require up to three transforms. The only known non-transitive 3-level SR graphs, (found by Mathon), has 139,300 vertices [Ma78].

However, there is a major problem with Algorithm H, and it is that superline transforms will not guarantee this algorithm to terminate. Therefore we need to set a limit L for the level of superline transform. The algorithm H is modified as follows:

```
ALGORITHM HA:
INPUT: Two graphs G1(V1,E1) and G2(V2,E2), and their vertex adjacency
        list.
OUTPUT: YES if G1 and G2 are isomorphic, otherwise NO.
HA1:   GITestB(G1,G2);
       k ← 0;
HA2:   terminate if answer is either YES or NO.
HA3:   delete all the distinguishable vertices in G1 and G2.
       k ← k+1;
HA4:   apply 1-superline transform to G1 and G2.
HA5:   GITestA(G1,G2), the initial partitioning is based on both
       the previous partition from lower hierarchy and the new
       v-invariants.
HA6:   if k < L go to H2.

HA7:  while there is unmatched partition do
        begin

HA8:      find the smallest partition in G1 and G2 and match these two
          group of vertices.

HA9:      applying algorithm RP in GITest to refine partition.
       end
```

```
HA10: check if match is correct, if so answer YES and terminate
      else check if there is any possible match,
           if not, answer NO and terminate
           else backtrack and go to step HA7.
```

HA7 to HA11 are backtracking steps and in the worst case they may take O(n!) steps. Next we will show that on average, Algorithm HA will only take O(|E| × log|V|) steps. We prove this by using a lemma of Babai, Erdos and Selkow [BES80]. Lemma 6.1 is a direct result of Theorem 1.2 in [BES80]. Algorithm BES is listed in their paper without a name and it is called BES here for easy reference.

Lemma 6.1 The probability that a random graph on n vertices belongs to the class K specified by BES, is greater than $1-\sqrt[7]{1/n}$ (for sufficiently large n).

```
Algorithm BES:
INPUT: a graph X having n vertices.
OUTPUT: a canonical labeling for graph X if it is in class K else just
        specify it is not in class K.

BES1  compute r=[3 log n/log 2];

BES2  compute the degree of each vertex of X;

BES3  order the vertices by degree, call them v(1), v(2),...,v(n),
```

denote by d(i) the degree of v(i) and d(i)$\geq$d(j) if i<j;

BES4 if d(i)=d(i+1) for some i, $1\leq i\leq r$, set X $\notin$ K
and terminate.

BES5 compute

$$f(v(i)) = \sum_{i=1}^{r} a(i,j)2^{i}$$

(i=r+1,...,n); where a(i,j)=1 if v(i) and v(j) are adjacent,
otherwise a(i,j)=0;

BES6 order vertices v(r+1),...,v(n) according to their f-values:
w(r+1),...,w(n) where f(w(r+1)) $\geq$ ... $\geq$ f(w(n));

BES7 if f(w(i))=f(w(i+1)) for some i, $r+1\leq i\leq n-1$,
set X $\notin$ K, terminate.

BES8 Label v(i) by i for i=1,...,r and w(i) by i for i=r+1,...,n
This labeling will be called canonical, set X $\in$ K. terminate.

Next we prove every graph in K can be identified by GITestB.

Theorem 6.2 Any two graphs in K can be distinguished by GITestB.
Proof:

Let G1(V1,E1) and G2(V2,E2) both be in K. Vertices u1,u2,...,un are in V1 and vertices v1,v2,...,vn are in V2. From step BES3 and BES4, we know that in G1 each of those r vertices with largest degree has a unique degree different from any other vertex in G1. The same case also happens in G2. Hence, they will have different v-invariant values of υ_1 in GITestB. The remaining (n-r) vertices are connected differently to the first r vertices, so they will mainly be distinguished by the v-invariant υ_2. In some cases, υ_2 may fail even if two vertices connect differently to other vertices, since the total degree of their neighbors might be the same. In these cases, algorithm RP can help us out because those r vertices have already been uniquely identified and the other vertices are connected to them differently. ∎

From lemma 6.1 and theorem 6.2, we know that almost all graphs can be tested by using GITestB. Therefore, the average complexity of algorithm H is the same as GITestB.

6. Conclusion

In this report, several graph isomorphism testing algorithms are proposed and analyzed. Vertex invariants and graph invariants are used to do the initial partitioning of the vertices of a graph. Characteristics of these invariants are also analyzed. Algorithm GITestA is based on these invariants and we have not found any graph, except for SR graphs and BIBD graphs, to be immune from its test. Algorithm GITestB is almost as powerful as GITestA and it has much improved time and space complexity. It is the most efficient algorithm for testing isomorphism of non-SR and non-BIBD graphs.

Graph transforms are used to make SR and BIBD graphs easier to test. A hierarchy of graph isomorphism testing algorithms is proposed such that it is able to test isomorphism of difficult graphs while keeping the complexity as low as possible. A probabilistic analysis similar to the one proposed by Babai, Erdos and Selkow [BES80] shows this hierarchy to be of average time complexity ,$O(|E| \log |V|)$, which is almost linear. Moreover, these algorithms can be easily modified to be used in a parallel machine.

There is not much theoretical analysis done on the the heuristic algorithms yet, so further research on this subject would be very useful. Graph transforms are very powerful in general, and further study on this topic may be profitable in solving the general graph isomorphism problem.

References

[ABB73] Ambler, A.P., Barrow, H.G., Brown, C.N., Burstall, R.M., and Popplestone, R.J., "A Versatile Computer-Controlled Assembly System", Artificial Intelligence Conference 1973.

[AHU74] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

[Ba80] Babai, L., "On the Complexity of Canonical Labeling of Strongly Regular Graphs", SIAM J. Comput. Vol.9, No.1, February, 1980.

[BES80] Babai L., Erdos, P. and Selkow S.M., "Random Graph Isomorphism", SIAM J. Comput. Vol. 9, No.3, August 1980.

[Bh80] Bhat, K.V., "Refined Vertex Codes and Vertex Partitioning Methodology for Graph Isomorphism Testing" IEEE Trans on System, Man, and Cybernetics, Vol. SMC-10, No. 10, October 1980.

[BJM79] Beyer, T., Jones, W. and Mitchell S., "Linear Algorithms for Isomorphism of Maximal Outplanar Graphs", JACM, Vol.26, No.4 October 1979, pp.603-610.

[BGM82] Babai, L.,Grigoryev, D.Y., and Mount, D.M., "Isomorphism of Graphs with Bounded Eigenvalue Multiplicity" Proceedings 14th annual ACM Symposium on Theory of Computing, 1982, pp.310-324.

[BP80] Balasubramanian, K., and Parthsarathy, k.R., "In Search of a Complete Graph Invariant for Graphs", Lecture Notes in Mathematics No.885, S.B. Rao Ed. Combinatorics and Graph Theory Proceedings, Caleutta 1980, pp.42-59.

[CB81] Colbourn, C.J. and Booth, K.S., "Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs", SIAM J. Comput., Vol.10, No.1, February, 1981.

[CG70] Corneil, D.G. and Gotlieb, C.C., "An Efficient Algorithm for Graph Isomorphism", JACM Vol. 17, No. 1, January 1970, pp.51-64.

[CK80] Corneil, D.G. and Kirkpatrick D.G., "A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem", SIAM J. Computing, Vol. 9, No. 2, May, 1980.

[FHG83] Fowler, G., Haralick, R., Gray, F.G., Feustel, C. and Grinstead, C., "Efficient Graph Automorphism by Vertex Partitioning", Artificial Intelligence 21(1983) pp.245-269.

[FM80] Filotti, I.S. and Mayer, J.N., "A Polynomial-time Algorithm for Determining the Isomorphism of Graphs of Fixed Genus" Proceedings 12th annual ACM Symposium on Theory of Computing, 1980, pp.236-243.

[Ga79] Gati, G., "Further Annotated Bibliography on the Isomorphism Disease", J. of Graph Theory, Vol. 3,1979, pp.95-109.

[GJ79] Garey, M.R. and Johnson, D.S., "Computer and Intractability: A guide to The theory of NP-completeness", Freeman, San Francisco, 1979.

[Gr82] Grigoryev D.Y., "Two Reductions of Graph Isomorphisms on Polynomials", Journal of Soviet Mathematics, Vol.20, No.4, Nov.,1982, pp.2296-2298.

[Ha69] Harary F., "Graph Theory", Addison-Wesley, 1969.

[Hoff] Hoffmann, C.H., "Testing Isomorphism of Cone Graphs", Proceedings 12th annual ACM Symposium on Theory of Computing, 1980, pp.244-251.

[HW74] Hopcroft, J. and Wong, J., "A Linear Algorithm for Isomorphism of

Planar Graphs", Proceedings 6th annual ACM Symposium on Theory of Computing, 1974, pp.172-184.

[KSO80] Kubo, N., Shirakawa, I., and Ozaki, H., "A Fast Algorithm for Testing Graph Isomorphism", International Symposium on Circuits and Systems (ISCAS), pp.641-644, 1982.

[LB79] Leuker G. and Booth K., "A Linear Time Algorithm for Deciding Interval Graph Isomorphism" , JACM 26, 1979, pp.183-195.

[Lu81] Lubiw, A., "Some NP-Complete Problems Similar to Graph Isomorphism", SIAM J. Computing, Vol.10, No.1, February, 1981.

[Li80] Lichtenstein, D., "Isomorphism for Graphs Embeddable on the Projective Plane", Proceedings 12th annual ACM Symposium on Theory of Computing, 1980, pp.218-224.

[Lu80] Luks, E.M., "Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time", Proceedings 12th annual ACM Symposium on Theory of Computing, 1980, pp.42-49

[Ma78] Mathon, R., "Sample Graphs for Graph Isomorphsim Testing", Proc., 9th S-E Conf. Combinatorics, Graph Theory, and Computing, 1978, pp.499-517.

[Me83] Megdal B.B. "VLSI Computational Structures applied to Fingerprint Image Analysis" Technical Report 5015, Cal Tech, 1983.

[Mi78] Miller, G.L., "On the $n^{\log n}$ Isomorphism Technique", Proceedings 10th annual ACM Symposium on Theory of Computing, 1978, pp.51-58.

[Mi79] Miller, G.L., "Graph Isomorphism, General Remarks" Journal of Computer and System Science, Vol. 18, No. 2, April, 1979, pp.128-141

[Mi80] Miller, G.L., "Isomorphism Testing for Graphs of Bounded Genus"

Proceedings 12th annual ACM Symposium on Theory of Computing, 1980, pp.225-235.

[RC77] Read, R.C. and Corneil, D.G., "The Graph Isomorphism Disease" Journal of Graph Theory, Vol.1, 1977, pp.339-363.

[Ry63] Ryser, H.J., "Combinatorial Mathematics" Carus Math. Monographs No. 14, New York: Math. Ass. Amer., 1963.

[SD76] Schmidt, D.C., and Druffel, L.E., "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices", Journel of the ACM, Vol.23, No.3(Jul.,1976), pp.433-445.

[Su65] Sussenguth, E.H. Jr., "A Graph-Theoretic Algorithm for Matching Chemical Structures" J. Chemical Documentation, Vol. 5, pp. 36-44, 1965.

[Ta72] Tarjan R., "Depth-First Search and Linear Graph Agorithms" SIAM J. Comput., Vol. 1, No. 2, June 1972.

Acknowledgments

Special thanks to my advisor, Randy Bryant, for his continued guidance and support on this research.

I would also like to thank Chao-lin Chiang, Tzu-mu Lin, and Bill Athas for their help and suggestions and my officemate Mike Newton who provides solutions whenever I got any question with Unix. Andy Matsuda did the painful job of reading my draft and smoothing it out. His patience and help are deeply appreciated.