A GENERAL COMPUTATIONAL TOOL FOR STRUCTURE SYNTHESIS

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

In the Department of Mechanical Engineering

University of Saskatchewan

Saskatoon

By

PEIREN HE

Keywords: structure synthesis, graph isomorphism, graph automorphism, canonical labeling, eigenvalue, eigenvector

ABSTRACT

Synthesis of structures is a very difficult task even with only a small number of components that form a system; yet it is the catalyst of innovation. Molecular structures and nanostructures typically have a large number of similar components but different connections, which manifests a more challenging task for their synthesis.

This thesis presents a novel method and its related algorithms and computer programs for the synthesis of structures. This novel method is based on several concepts: (1) the structure is represented by a graph and further by the adjacency matrix; and (2) instead of only exploiting the eigenvalue of the adjacency matrix, both the eigenvalue and the eigenvector are exploited; specifically the components of the eigenvector have been found very useful in algorithm development. This novel method is called the Eigensystem method.

The complexity of the Eigensystem method is equal to that of the famous program called Nauty in the combinatorial world. However, the Eigensystem method can work for the weighted and both directed and undirected graph, while the Nauty program can only work for the non-weighted and both directed and undirected graph. The cause for this is the different philosophies underlying these two methods. The Nauty program is based on the recursive component decomposition strategy, which could involve some unmanageable complexities when dealing with the weighted graph, albeit no such an attempt has been reported in the literature. It is noted that in practical applications of structure synthesis, weighted graphs are more useful than non-weighted graphs for representing physical systems.

Pivoted at the Eigensystem method, this thesis presents the algorithms and computer programs for the three fundamental problems in structure synthesis, namely the isomorphism/automorphism, the unique labeling, and the enumeration of the structures or graphs.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere thanks to my supervisor, Professor C. Zhang, for his invaluable guidance, stimulating discussion, and never-ending encouragement in the whole research as well as the critical review of the manuscript.

I would like to extend special thanks to other members of my advisory committee: Professor R. Burton, Professor A. Dolovich, and Professor K. Takaya, for their worthwhile guidance and suggestions as my advisory committee members in the whole research.

I wish to acknowledge Dr. Q. Li from Nanyang Technological University (Singapore) for her valuable suggestions, Mr. F. X. Wu for the discussion on the algorithm, and Mr. Edwin Zhang for the partial correction of my English writing. Ms. Wanda Drury has kindly made English correction for the first 4 chapters.

I also wish to gratefully acknowledge the support for this work from the Natural Science and Engineering Research Council (NSERC) of Canada.

This thesis is dedicated to my wife ***Yi*** *and my son* ***Xu****.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

AAM:	adjusted adjacency matrix
AM:	adjacency matrix
CSP:	constraint satisfaction problem
DAG:	directed acyclic graph
DOF:	degree of freedom
EA:	evolutionary algorithms
FBS:	function-behavior-state
GA:	graph automorphism
GI:	graph isomorphism
NP:	non-deterministic polynomial-time

NOMENCLATURE

λ_i:	The ith eigenvalue
Λ_a:	The eigenvalue matrix of matrix A
Λ'_a:	The eigenvalue matrix, in which the eigenvalues are in an ascending order, of matrix A
ϕ:	A vertex mapping among the vertices between two graphs
ϕ_{i+}:	The vertex mapping based on the eigenvector pair (**x**, **y**) corresponding to the ith distinct eigenvalue between two graphs
ϕ_{i-}:	The vertex mapping based on the eigenvector pair (**x**, -**y**) corresponding to the ith distinct eigenvalue between two graphs
Φ_i:	The common mapping among vertices up to the ith distinct eigenvalue, i.e., $\Phi_i = (\phi_{i+} \cap \Phi_{i-1}) \cup (\phi_{i-} \cap \Phi_{i-1})$
$\mathbf{x}_i^a$:	The eigenvector corresponding to the ith eigenvalue of matrix A
$\mathbf{x}_i'^a$:	The eigenvector, in which the components are in an ascending order, corresponding to the ith eigenvalue of matrix A

CHAPTER 1
INTRODUCTION

1.1 Design of Physical Systems or Artifacts

Physical systems (or artifacts) consist of a set of physical elements connected in a semantically sensible manner. Such a view of systems is also called a *structural* view. *Behaviors* of the artifact consist of a causal model of dynamic interactions between these structural elements in the form of a sequence of behavioral states and transitions between these states. *Functions* of the artifact describe the overall intentional consequences of the behavior. The structure (or the artifact) has a boundary which separates the structure from the rest of world. The portion of things in the rest of world which interacts with the artifact is called the *environment*. An artifact may be further decomposed into sub-artifacts and components. Components are those elements that are not further decomposed per se. Artifacts or sub-artifacts have two patterns in terms of connectivity among their elements: the tree pattern and the network pattern.

Design (of artifacts) is defined as an activity that generates a description of an artifact based on functional specifications and constraints [Tong and Sriram 1992]. Constraints can be of structural, behavioral, or resource limitations. The design process therefore involves three main types of knowledge about a domain: functional, behavioral, and structural. Therefore, the mapping between these three types of knowledge is central to the design process [Reich 1991].

Design is further classified into routine and non-routine designs [Tong and Sriram 1992; Gero 1994]. In *routine design*, the designer knows both the space of possible problems and the space of possible solutions. Therefore, the design in this case reduces to finding a mapping function between these two known spaces. *Non-routine design* is further divided into two subclasses: innovative and creative. In *innovative design*, the problem and solution spaces can be augmented by way of changing the dimensions of prescribed structures. In *creative design*, either the problem space or the solution space is lacking. Designs generated in the creative design process thus define novel classes of artifacts. Therefore, the creative design process is a process where new ideas or solutions are synthesized in the absence of prior examples [Suh 1990].

1.2 Structure Synthesis

A closer look at design can find that a design process includes many steps. The function needs to be transformed into the expected behavior. The actual behavior of a structure, which is potentially being a solution to a design problem, then needs to be evaluated against the expected behavior. An unsatisfactory evaluation will trigger a process to look for new structures. This process is called *synthesis*. Informally, synthesis means putting things together. Formally, *structure synthesis* involves configuring elements into a system structure that satisfies the expected behavior.

It may be clear that structure synthesis is an essential step in a creative design process, as there is not any pre-known structure prototype. Structure synthesis may also be useful in the *re-design process* which is a design between the routine design and the creative design. A typical scenario in the re-design process where structure synthesis is useful can be described as follows. The motivation of a design is dissatisfaction with an existing artifact with respect to some newly added functional and/or constraint requirements. The designer would then like to seek a 'new' structure which could be 'merged' into the existing structure to form an overall structure that can meet the updated requirements. It is clear that finding a new structure is a creative design process. The merging process may require changing the dimensions of the existing structures, which is characterized as routine design.

Structure synthesis is found in many applications in science and engineering disciplines. In mechanical engineering, for example, the structural synthesis of kinematic chains or mechanisms requires enumerating all potential isomers under a given set of constraints in order to find the optimal design [Johnson and Towfigh 1978; Crossley 1965]. Design and development of modular robotic systems requires the enumeration of all possible assembly configurations out of the modules and to find an optimal one among them for a specific task [Chen and Burdick 1998]. In biology, identification of isomorph/isomer could be used in tracking the evolution of the genotype and the phenotype of a virtual creature, which further infers what kind of creature it is [Sims 1994]. In chemistry, organic chemists must identify all possible molecular structures for chemical documentation systems [Randic 1974]. As a recent trend in mathematical and computational chemistry, topological indices of molecular structures are studied to describe molecular similarity/dissimilarity and to estimate molecular properties for novel drug discovery, molecular design, and toxicological hazard assessment [Basak and Magnuson 1988; Basak *et al.* 2001]. In biochemistry, structure synthesis is used for predicting molecular networks from massive amounts of genome information and for functionally detecting related enzyme clusters [Ogata *et al.* 2000].

1.3 Fundamental Questions in Structure Synthesis

There are two fundament questions to be answered with structure synthesis. The first question is whether the description of the structure is unique or canonical. The second question is how to transform from a description of function and constraint into a description of structure.

The uniqueness of the description or the representation is important. For example, two different structures, both of which are good candidates for solution to a design problem, may be regarded to be the same because of a non-unique representation of them. In an opposite situation, two similar designs may be regarded as different and they then go through a design process, which implies a waste of design resources (the designer's time and effort).

The transformation process in the second question is very challenging and is, in fact, the substance in a creative design process – the mapping from the (new) problem space to the (new) solution space. The challenge is brought in because it is very difficult to model the transformation problem and to develop a computational algorithm for such a transformation. Such an effort was attempted by Tomiyama and Yoshikawa [1987], but they have not succeeded. Approaches complementary to the computational approach are the empirical ones. In the empirical approach, the basic procedure is to develop a knowledge base which includes the function and structure and then find the correspondence between them at different levels. With the help of such a knowledge base, a design problem is modeled by decomposing its function into a suitable function lattice, and then this lattice is matched with those in the knowledge base. A successful matching results in a set of structures, which are to be integrated into an artifact.

Another useful idea for structure synthesis is to enumerate all possible structures subject to a set of constraints. This synthesis process starts with a 'known' structure that meets the functional and constraint requirements. The synthesis problem is formulated as finding all possible alterative structures for that known structure (which may be called the *seed structure*) subject to the constraint applied to the seed structure.

The following are more formal statements of the fundamental problems with structure synthesis, which lay the foundation for this thesis research.

Characterization of structures. The characterization of structures is to create structure patterns by encoding the structure features which can completely and efficiently identify the structures. An incomplete characterization of a structure does not correctly represent the structure and would cause a mistake in structure synthesis, while an inefficient characterization of a structure will result in an unmanageable time-consuming effort.

Similarity of structures. Once a structure is characterized, there is a need to match it to known structures to determine its novelty. Structures could vary from each other. Depending on different abstraction levels and associated viewpoints or contexts that make the levels meaningful, two structures may be recognized either as same or different. For example, two TV sets may be considered as the same from their functions (one abstraction level), while they may be considered as different because of their different sizes (another abstraction level). When a kind of database with reference index is built for structures (say TV sets), similarities of structure together with their different abstraction levels need to be addressed. If two structures are the same, this thesis will refer to them as having hard similarity; otherwise they will be considered to have soft similarity.

Enumeration of structures. In order to find a new or an optimal design in various structure configurations, designers usually need to answer the following question: How many different configurations are possible from a type of structure? The process to find all distinct configurations is called *enumeration*. Enumeration of distinct structures is thus an important kernel for structure synthesis.

It should be noted that throughout this thesis the term 'structure' in the context of structure synthesis represents topological information of a physical system. The topological information includes, depending on design interests, (1) types of components, (2) types of connections, and (3) patterns of connectivities.

1.4 Related Work

1.4.1 Basic Concepts

The complexity of structure synthesis is such that synthesis requires (1) the selection of one among tens, hundreds, or thousands of structure options seemingly similar, and (2) the comparison/identification/differentiation of structures. Therefore, the structure synthesis process must be aided by computer.

A formal representation of structures is thus needed for computer processing. The graph is a natural choice for such a representation. Basically, a graph consists of a set of vertices and a set of edges that connect the vertices in various ways. A more formal description of graph theory is found in Chapter 2. Structure synthesis then becomes graph synthesis. Comparison of two structures becomes comparison of two graphs, and enumeration of structures becomes enumeration of graphs. In graph theory, graph sameness (similarity) and graph enumeration are called, respectively, graph isomorphism (subgraph isomorphism) and the counting of graphs. Again, for a more formal discussion of them refer to Chapter 2. Graph isomorphism is such that two graphs are exactly the same, while subgraph isomorphism is such that a graph, say *A*, is 'matched' with another graph, say *B*, in the sense that graph *B* contains graph *A*. Subgraph isomorphism becomes graph isomorphism if graph *B* has the same size as graph *A*.

1.4.2 Graph Isomorphism/Enumeration

There are two basic ideas underlying various approaches for graph isomorphism. The first idea is to manipulate a graph via a permutation procedure (row and column exchange). The second idea is to define a variable which is related to the graph and (further) to define a function on the variable; the optimization of this function leads to a representation of the graph. Approaches based on the first idea may be called the graph theory approach, while approaches based on the second idea may be called the evolutionary computational approach.

It is difficult to solve the graph isomorphic identification problem for a general graph (a graph without any constraint on it). Many studies have been developed attempting to solve particular classes of graph isomorphism problems [Babai 1995; Babel 1995; Bodlaender 1990; Fortin 1996; Luks 1982; Read and Corneil 1977]. In this thesis, the isomorphism problem for general graphs is examined.

Currently, the algorithm for (general) graph isomorphism, which has yet to be challenged by any counterexample, would refer to the Nauty (No AUTomorphisms, Yes?) program presented by McKay [1981]. Nauty is a backtrack program for computing automorphism groups of undirected graphs and digraphs. It can also produce a canonical labeling for graphs. However, this program cannot handle weighted graphs which are often seen in many applications. Furthermore, as the Nauty program has not resolved the NP hard nature of graph isomorphism, it would be a good strategy to examine the graph isomorphism with its related issues from a different angle.

In the evolutionary computational approach, the current state of the art is such that the most complex graph (which has been tested) is small in size in terms of the number of vertices. The evolutionary computation approach is still in an exploratory stage. The capability of this kind of approach can be limited by the particular evolutionary algorithm (e.g., Genetic Algorithm) employed.

1.5 Objectives and Scope of the Thesis

As discussed before, graph isomorphism and its relevant problems are still not solved well. With the rapid advancement of nanotechnology and biotechnology, the problem of synthesis of structures with hundreds and thousands of nano objects is emerging as critical. Based on a preliminary finding (which the author obtained several years ago) that a graph could be represented by a quadratic surface, and that both eigenvalues and eigenvectors are useful to characterize the graph or the quadratic surface [He *et al.* 2000, 2001, 2002a, 2002b, 2002c, 2003], this thesis study aims to further elaborate this finding for developing a more effective method for graph isomorphism and its relevant problems (i.e., the three fundamental issues discussed before) in the general area of structure synthesis.

1.6 Organization of the Thesis

Chapter 2 gives a background and literature review about graph and graph-based methods for structure synthesis. Selected concepts in graph theory that are involved in graph-based structure synthesis are introduced. The most known graph-based algorithms on structure enumeration are introduced and analyzed.

The approach developed in this thesis, called the 'Eigensystem' approach, is introduced and elaborated in Chapters 3, 4, and 5, respectively. Chapter 3 gives a basic introduction to the Eigensystem approach, and its geometric significance. Chapter 4 describes in detail (1) the basic algorithms associated with the Eigensystem approach, (2) a new matrix called 'adjusted adjacency matrix', (3) the computational complexity of the Eigensystem approach, and (4) the method as well as the algorithm for digraph isomorphic identification. As a result, the discussion in Chapter 4 addresses the second fundamental issue (see Section 1.3). Chapter 5 gives the algorithm for solving the graph counting problem using the Eigensystem approach. Subsequently, structure enumeration and structure characteristic problems (i.e., the first and third fundamental issues) are solved based on this algorithm.

Chapters 6 and 7 describe the applications of the proposed Eigensystem approach in structure synthesis of mechanism design and molecular structure design, respectively.

Chapter 8 summarizes the Eigensystem approach, gives the conclusion, and discusses the future direction of work in this field.

CHAPTER 2
BACKGROUND AND LITERATURE REVIEW

2.1 Introduction

As mentioned previously in Chapter 1, graph models and graph algorithms have been widely used for representing and synthesizing structures. There exist numerous research projects and publications about graph theory and applications. The applications include various areas ranging from mechanism structures, electrical networks and communications networks, chemistry, and geography, to architecture [Wilson and Beineke 1979]. The purpose of this chapter is to provide background and a literature review of the graph-based methods closely related to the research objectives defined in Chapter 1. In particular, Section 2.2 presents some primary concepts/notions to provide a set of unified terminologies for the reminder of the thesis. Section 2.3 discusses knowledge representation using graphs for various applications. Some typical methods known in the literature for structure synthesis are introduced in Section 2.4. A summary is given in Section 2.5.

2.2 Preliminaries

2.2.1 Graph

A *graph* is an ordered pair $G = (V, E)$, where V is a finite, non-empty set of objects called *vertices* (or nodes) and E is a set of pairs of vertices called *edges* (or arcs). The sets V and

E are also denoted as *V*(*G*) and *E*(*G*), respectively. Further, a graph is denoted as a *simple graph* if *E* is a set of distinct elements of 2-subsets of *V*, that is, there is at most one edge between any two vertices in the graph and there is no self-loop at any vertex. Figure 2.1 shows examples of a simple graph and two non-simple graphs. The graphs shown in Figure 2.1 are *connected* because there is a path connecting every pair of vertices. The reminder of the thesis will use the term 'graph' for the simple and connected graphs.

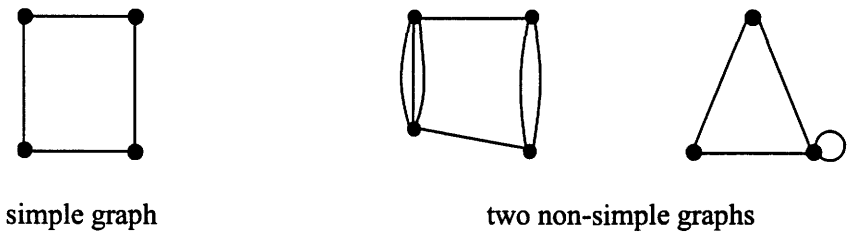


Figure 2.1 Examples of a simple graph and two non-simple graphs.

If *E*(*G*) of graph *G* is a set of ordered pairs of vertices, that is, edge $e = \{u, v\} \in E(G)$ is directed from initial vertex *u* of *e* to terminal vertex *v* of *e*, then the graph *G* is denoted as a *directed graph* (or *digraph*); otherwise, the graph *G* is called an *undirected graph* (see Figure 2.1). Figure 2.2 shows a directed graph.

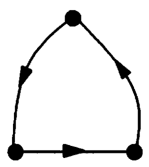


Figure 2.2 A directed graph (digraph).

A *labeled graph* refers to the graph whose vertices are labeled (see Figure 2.3); a labeled graph is denoted as $V_n(G) = \{1, 2, 3, \ldots, n\}$. Graphs shown in Figure 2.1 and Figure 2.2 are *unlabeled graphs*.

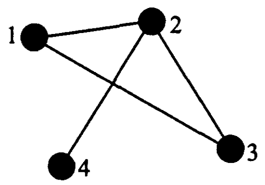


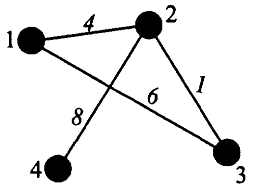
Figure 2.3 A labeled graph.

Two vertices of a graph are *adjacent* if they are connected by an edge. The number of the vertices adjacent to a vertex is called the *degree* of the vertex. The graph in which all vertices have the same degree is a *regular graph*. The *adjacency matrix* (AM) of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j), according to whether v_i and v_j are adjacent or not. The AM must have 0s on the diagonal because of no self-loop. The following is the AM of the graph shown in Figure 2.3:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}.$$

It should be noticed that the above-mentioned AM is a symmetric. Indeed, for an undirected graph, the AM is a symmetric. The set of eigenvalues of the AM of a graph is called the *spectrum* of the graph. Graphs having the same graph spectrum are called the *cospectral graphs.*

Edge describes a very generic property, i.e., connectivity between any two vertices. When information beyond the connectivity needs to be represented, a descriptor is associated with the edge. This descriptor can be represented by a number (e.g., 1, 2, etc.). This number is also called the weight. Therefore, a *weighted graph* is a graph having a weight (or a number) associated with each edge. The AM of a weighted graph can be represented by assigning the weight of each edge at the position (v_i, v_j). Figure 2.4 illustrates a weighted graph and its AM.



$$\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 4 & 6 & 0 \\ 4 & 0 & 1 & 8 \\ 6 & 1 & 0 & 0 \\ 0 & 8 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 2.4 A weighted graph and its AM.

Distance of any two vertices is defined as the shortest path between them. The shortest path is the least number of connected edges from one vertex to another. A *distance matrix*, which is another way to represent a graph, is defined as putting the distance at the position (v_i, v_j). The distance matrix for the graph shown in Figure 2.3 is as follows:

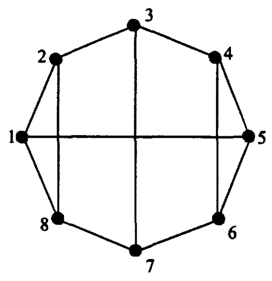
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{array}.$$

A detailed introduction of these concepts can be found in Harary [1969] and Carre [1979].

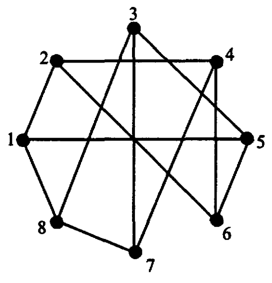
2.2.2 Isomorphism and Automorphism

Two graphs are equal if they have the same vertex set and the same edge set. There are other ways in which two graphs may be regarded as the 'same'. For example, one could regard two graphs as being the 'same' if it is possible to relabel the vertices of one and obtain the other. Such graphs are identical in every respect except for the labels of the vertices. In this case, two graphs are called *isomorphic*. *Graph isomorphism* (GI) is the problem of determining if two graphs are isomorphic. Two graphs, A and B, are isomorphic if (1) there is a one-to-one correspondence between their vertices, and (2) there is an edge between two vertices of graph A if and only if there is an edge between the two corresponding vertices of graph B. Mathematically, given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, a one-to-one mapping σ of V_1 onto V_2 is called an isomorphism if and only if $(u, v) \in E_1 \Leftrightarrow (\sigma(u), \sigma(v)) \in E_2\ \forall u, v \in V_1$.

It is possible that more than one isomorphism exists for two isomorphic graphs. For example, Figure 2.5 shows two isomorphic graphs with eight vertices. There exist four isomorphisms (one-to-one mappings), as listed in Table 1, $\sigma_1(v)$, $\sigma_2(v)$, $\sigma_3(v)$, and $\sigma_4(v)$, between these two graphs.



(a)



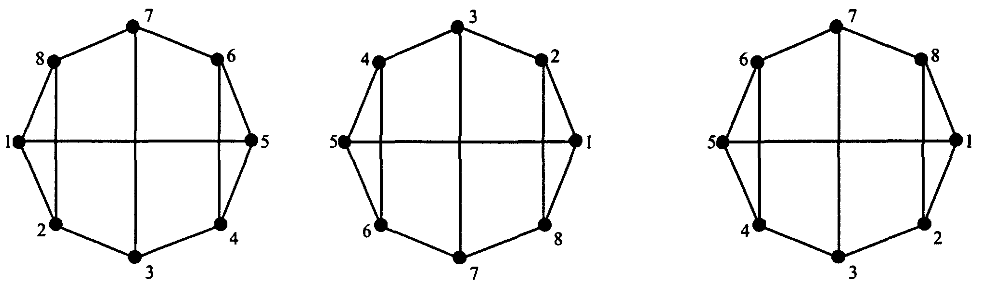
(b)

Figure 2.5 Two isomorphic graphs with eight vertices.

Table 2.1 Four isomorphisms of two isomorphic graphs shown in Figure 2.5.

Vertex v in Fig. 2.5a	1	2	3	4	5	6	7	8
Vertex $\sigma_1(v)$ in Fig. 2.5b	4	6	5	3	7	8	1	2
Vertex $\sigma_2(v)$ in Fig. 2.5b	4	2	1	8	7	3	5	6
Vertex $\sigma_3(v)$ in Fig. 2.5b	7	3	5	6	4	2	1	8
Vertex $\sigma_4(v)$ in Fig. 2.5b	7	8	1	2	4	6	5	3

If there is an isomorphism (one-to-one mapping) between a set of vertices of a graph and the set of vertices of the original graph, it is called *automorphism*. Mathematically, *graph automorphism* (GA) can be stated as: Given a graph $G(V, E)$, a one-to-one mapping σ of V onto V is called an automorphism if and only if $(u, v) \in E \Leftrightarrow (\sigma(u), \sigma(v)) \in E$ $\forall u, v \in V$. It is obvious from the definition that every graph has at least one automorphism. The automorphism group (a set of automorphisms) of a graph characterizes its symmetric property. From Figure 2.5a, it can be seen that the graph is symmetric to edge $\overline{15}$, edge $\overline{37}$, and both edge $\overline{15}$ and edge $\overline{37}$, respectively. This means that there are three automorphisms associated with this graph. Figure 2.6 illustrates these three automorphisms. Note that the graph in Figure 2.5a has four automorphisms in total, as the graph is an automorphism of itself. As a special situation, if a graph, when viewed from any vertex or edge, looks the same, the graph is called the *symmetric graph*. An example of the symmetric graph is shown in Figure 2.7. In general, the stronger the symmetry of a graph is, the more the number of the automorphisms is. For instance, 3,840 automorphisms exist for the graph with 32 vertices shown in Figure 2.8a [McKay 1981], and 40,320 automorphisms exist for the graph with 8 vertices shown in Figure 2.8b!



(a) symmetry to edge $\overline{15}$ (b) symmetry to edge $\overline{37}$ (c) symmetry to both $\overline{15}$ and $\overline{37}$

Figure 2.6 Three automorphisms of the graph shown in Figure 2.5a.

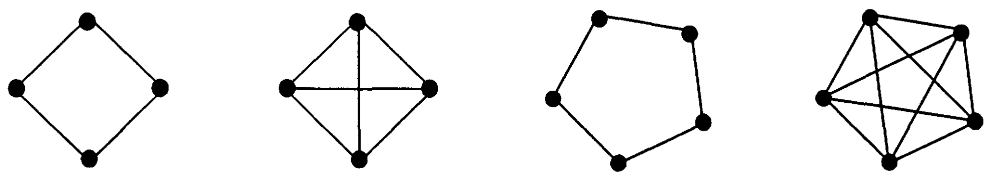
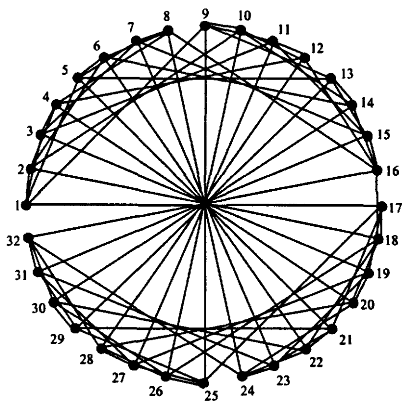
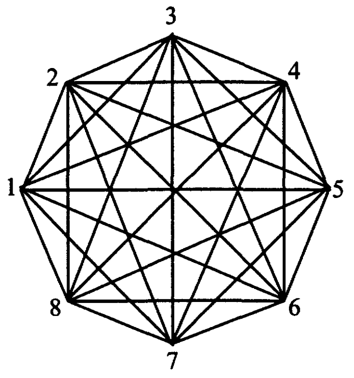


Figure 2.7 Examples of symmetric graphs.



(a) 3,840 automorphisms



(b) 40,320 automorphisms

Figure 2.8 The number of automorphisms of two graphs with 32 and 8 vertices.

It is believed that given a graph, a set of its isomorphic graphs and a set of its automorphic graphs should share some common properties and behaviors. Enumeration of these two sets of graphs for a given graph should be of potential interest and significance in structure synthesis. The enumeration of all isomorphisms between two graphs is called the *graph isomorphism counting problem*. The enumeration of all automorphic graphs for a graph is called the *graph automorphism counting problem*.

A graph *invariant* is a graph-theoretical property or parameter that is preserved by isomorphism. In other words, it is a property that does not depend on the way in which a graph is labelled. Typical invariants include the number of vertices, the number of edges, the degree of each vertex, etc. The AM is not a graph invariant because it depends on how the graph is labelled. The set of graph invariants which can uniquely identify a graph is called the *complete invariants* of the graph.

For more materials on GI and GA refer to Hoffmann [1982] and Harary [1969].

2.2.3 Computational Complexity

A naïve algorithm enumerating or counting isomorphisms between two graphs is done by factorizing all possible vertex mappings of a graph and then comparing them with one another. For this technique, the comparisons of up to $n!$ times are needed. Such an amount of comparison is too huge to be completed with contemporary computing technology. For example, for a graph with 15 vertices, the number of such comparisons is 1,307,674,368,000 (15!). Research into efficient algorithms for graph isomorphisms and automorphisms aims to reduce the number of comparisons. Finding the number of such comparisons, given an algorithm for graph isomorphism, is not a simple task, and this is based on complexity theory. *Complexity Theory* deals with the resources required during computation to solve a problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem). Time complexity is most commonly discussed in complexity analysis. The *time complexity* of a problem is the number of steps that it takes to solve an instance, as a function of the size of the instance. If an instance that is n bits long can be solved in n^2 steps, then this is said to have a time complexity of n^2. Of course, the exact number of steps will depend on what machine or language is used. To avoid this machine or language dependent problem, a symbolism called *Big O* notation is used in complexity theory, computer science, and mathematics, to describe the asymptotic behavior of functions. Basically, it tells about how fast a function grows or declines. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$

on most other computers. So this notation allows us to generalize away from the details of a particular computer. The time complexity is a *polynomial time* if the execution time of a computation is no more than a polynomial function of the problem size n, i.e., $O(n^k)$ where k is a constant.

Four well-known complexity classes in complexity theory are associated with the GI problem as follows:

P. In complexity theory, the class P consists of those decision problems that can be solved on a deterministic Turing machine in an amount of time that is *polynomial* in the size of the input. *Turing machine* is an abstract model of computer, which has an unlimited amount of information storage, to give a mathematically precise definition of algorithm. *Deterministic* means permitting at most one next move at any step in a computation. The question "For a positive integer N, is there a positive integer m such that $N = 4m$?" is a P problem.

NP. NP (*Nondeterministic Polynomial-time*) is the set of decision problems solvable in polynomial time on a nondeterministic Turing machine. The term *nondeterministic* means permitting more than one choice of next move in some steps in a computation.

NP-complete. The complexity class NP-complete is the set of problems that are the hardest problems in NP in the sense that they are the ones most likely not to be in P. If a particular algorithm can work on a particular NP-complete problem quickly, then it works on all NP problems quickly. Problem *B* is called *NP-hard* if there exists NP-complete problem *A* that can be solved in polynomial time using a polynomial algorithm for the problem *B*. Every NP-complete is thus an NP-hard problem. As an example, the well-known Traveling Salesman problem belongs to a NP-complete problem. The description of this problem is: Given a set of cities, the distances between them, and a bound *C*, does there exist a tour of all the cities having total length *C* or less?.

#P. The complexity class #P (pronounced "*sharp P*") is the set of counting problems associated with the decision problems in the set NP. While an NP problem is often of the form "Are there any solutions that satisfy certain constraints?" the corresponding #P problem asks 'how many' rather than 'are there any'. Clearly, an #P problem must be at least as hard as the corresponding NP problem. If it is easy to count answers, then it must be easy to tell whether there are any answers.

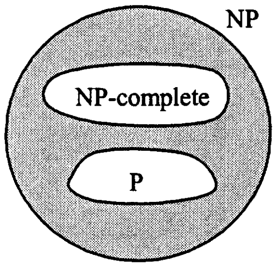


Figure 2.9 The world of NP [Garey and Johnson 1979].

Figure 2.9 illustrates the world of NP assuming that P ≠ NP. The GI problem occupies an important position in the world of complexity analysis. It is one of the few problems which is in NP but is known neither in P nor NP-complete [Read and Corneil 1977;

Schoning 1988; Kobler *et al.* 1993; Fortin 1996]. Therefore, the graph isomorphism/ automorphism counting problems are #P problems [Mathon 1979].

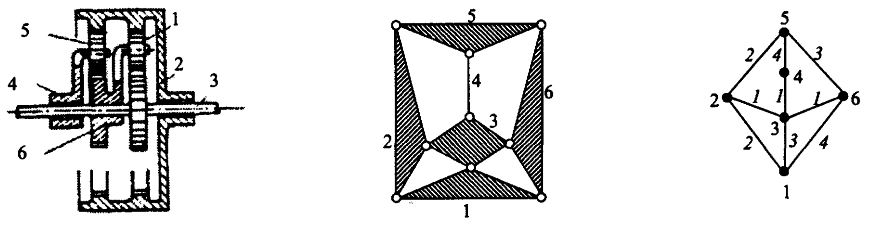
For further information on computational complexity refer to Garey and Johnson [1979] as well as Balcazar *et al.* [1994].

2.3 Knowledge Representations of Artifacts Using Graphs

To treat a structure synthesis problem in a particular application using graph theory, there should be a way to represent domain knowledge of that particular application using graphs. In the following, graph or graph-like or network representation for selected applications is discussed. The term *artifact* refers to a generic object or system in any application domain.

2.3.1 Graph Representation of Mechanisms

Figure 2.10 describes a typical application of graph representation for mechanisms/ kinematic chains in machine design. The engineering drawing of an industrial gear drive shown in Figure 2.10a is converted into a schematic kinematic diagram shown in Figure 2.10b, in which shaded shapes and lines represent the mechanical components in Figure 2.10a, while the circles represent connections. The representation shown in Figure 2.10b ignores information regarding physical shapes of components, while focuses on the connectivity among components. The representation shown in Figure 2.10b is still in the scope of mechanical system design and is further converted into Figure 2.10c in which labeled vertices represent the components in Figures 2.10a and 2.10b, edges represent the connections among the components. In Figure 2.10c, assigned weights on the edges represent different types of the connections. Throughout this thesis, the graph shown in Figure 2.10c is under investigation. Studies on graph representation of kinematic chains can be found in [Crossley 1965, 1966; Davies and Crossley 1966; Woo 1967; Mruthyunjaya and Raghavan 1979; Sohn and Freudenstein 1986; Ambekar and Agrawal 1987; Tang and Liu 1993; Schmidt *et al.* 2000].

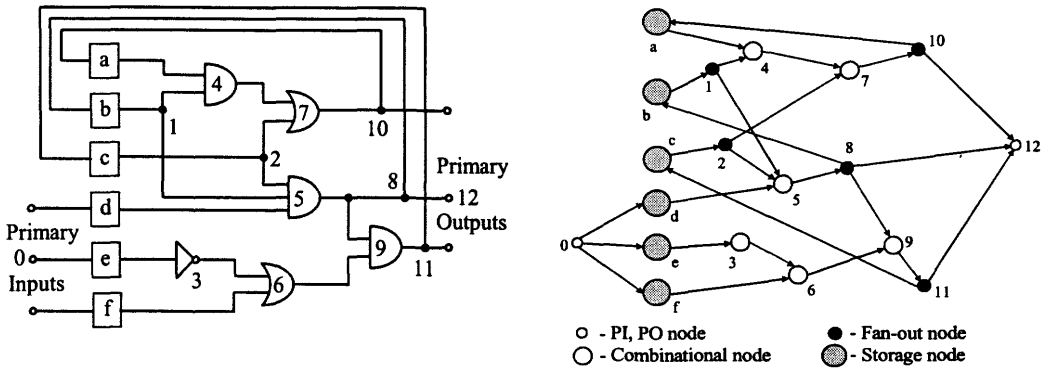


(a) physical representation (b) schematic representation (c) weighted graph

Figure 2.10 An industrial gear drive and its graph representation [Ambekar and Agrawal 1987].

2.3.2 Graph Representation of Circuits

Graph has been also applied for representations of digital systems in synthesis [Wilson and Beineke 1979; Giovanni 1992; Ubar 1996]. Figure 2.11 shows the directed graph model of a cyclic sequential circuit whose components convert into the vertices and flows convert into the directed edges. It should be noted that the information captured with the graph is of topolospecifically including the type of elements and the flow direction from one element to another.



(a) cyclic sequential circuit (b) graph model

Figure 2.11 Cyclic sequential circuit and graph model [Giovanni 1992].

2.3.3 Graph Representation of Molecules

Graph representations of chemical compounds have a long history in molecule design and chemical documentation [Sussenguth 1963; Randic 1974; Wilson and Beineke 1979; Basak *et al.* 1994]. Weighted graphs are usually used for representing chemical compounds where the weighted vertices are the atoms and the weighted edges are the covalent bonds. Figure 2.12 illustrates a typical organic molecule and its graph model in which hydrogen atoms are omitted in the graph model for similarity. Addition of the hydrogen atoms on the present graph model is always possible.

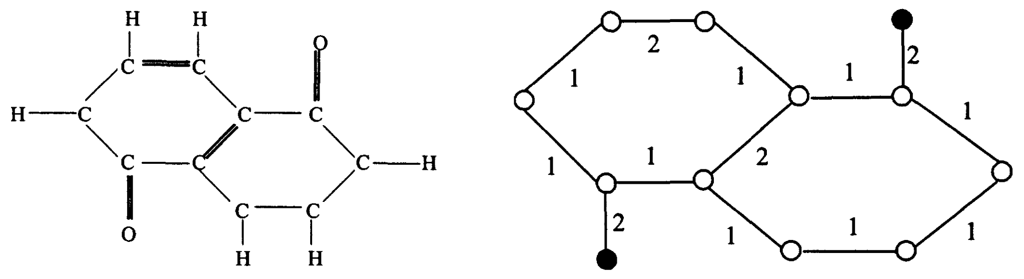


Figure 2.12 A typical organic molecule and its graph model [Fortin 1996].

2.3.4 Graph Representation of Assemblies

Heisserman [1999] introduced the Boeing Company's approach of comparing designs between complex aircraft assembly structures, which was implemented in the Boeing's Genesis generative design system. An aircraft hydraulic assembly was represented by a directed acyclic graph (DAG) which constructs a hierarchical assembly, the occurrence tree, and the connections between the DAG and the tree. Figure 2.13 shows a simple aircraft hydraulic assembly and its associated DAG. Three entities form the basis of tree representation: parts, part-usages, and occurrences. Each part is defined once with all users sharing that common definition: Part-usages locate 'child' parts. For example, the 'child' part 'filter' shown in Figure 2.13 is located by two part-usages, A and B in the coordinate system of a 'parent' assembly. Occurrences are instances of part-usages. Between part-usage and its occurrence is a one-to-many relationship represented with arcs. Some operations, such as comparison operation and merge operation, can then be performed on the assembly graph to compare and merge different versions of a design.

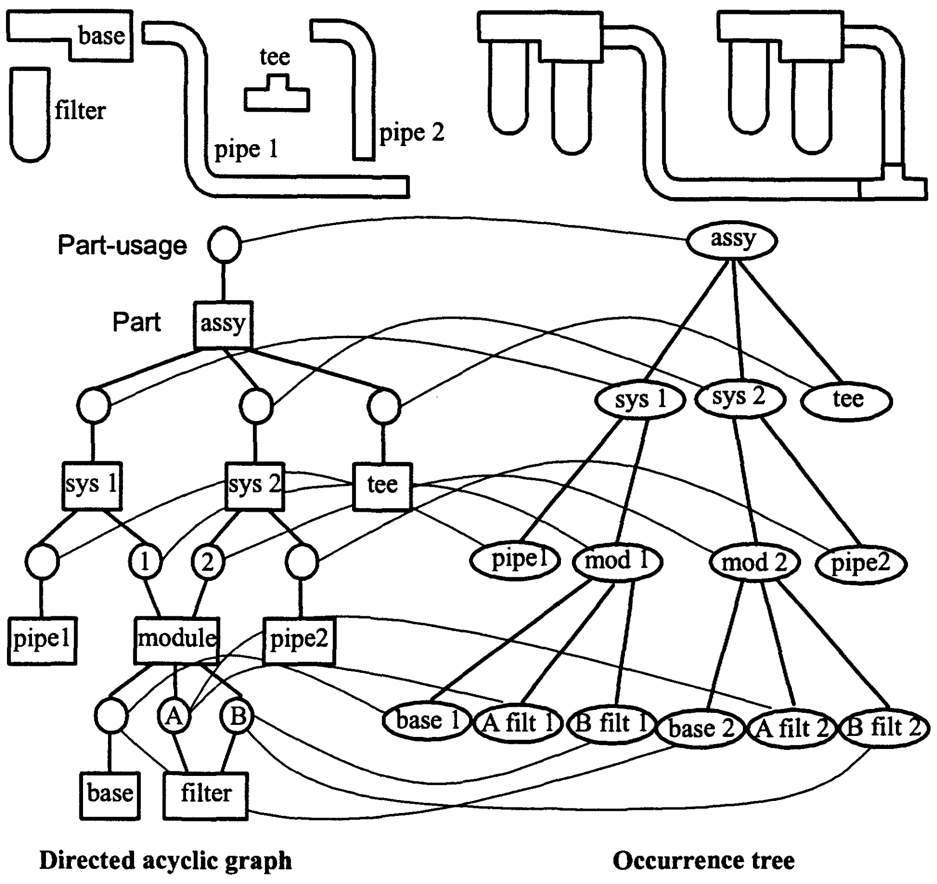


Figure 2.13 An aircraft hydraulic assembly and its assembly graph [Heisserman 1999].

2.3.5 Graph Representation of Design Processes

In an attempt to automate a design or structure synthesis process fully, a framework which decides what information should be recorded needs to be developed. One of such frameworks, which is based on the concept of function-behavior-state (FBS), was proposed [Li and Zhang 1999]. This framework suggests a hybrid graph representation shown in Figure 2.14, which is constructed on the basis of line graphs with both directed and undirected subgraphs and both weighted and non-weighted graphs, for various categories of design knowledge based on the FBS architecture, and then a general algorithm was developed for the hybrid graph-based comparison.

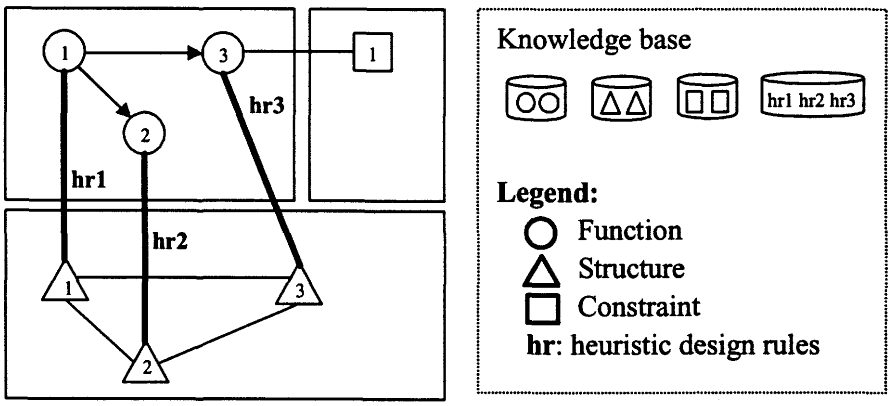


Figure 2.14 A hybrid graph representation of the FBS architecture [Li and Zhang 1999].

2.4 Algorithms for Graph Isomorphism

Structure synthesis reduces to graph synthesis. In graph synthesis, the most important algorithm is graph isomorphism algorithm. Numerous studies have been devoted to this subject but they have not produced an algorithm having a provable polynomial worst case that exists for general graphs. Some have developed special polynomial time algorithms for a restricted class of graphs [Luks 1982; Bodlaender 1990; Babel 1995], but most of them do not have a polynomial time worst case [Read and Corneil 1977; Babai 1995; Fortin 1996]. Theoretical unable of a robust algorithm does not discourage the development of 'fast' algorithms for practical problems if they work intuitively. Therefore, development of algorithms for graph isomorphism becomes business not for mathematician only. Around the world, graph isomorphism articles are published in a variety of journals, e.g., Journal of Graph Theory, Journal of Algorithms, SIAM Journal on Computing, Journal of Computer and System Sciences, Information Processing Letters, Mechanism and Machine Theory, ASME Transactions Journal of Mechanical Design, IMECH Proceedings, Journal of Chemical Information and Computer Sciences, and The Journal of Chemical Physics. Among all these algorithms two categories of algorithms most relevant to the method and algorithm developed in this thesis study are (1) characteristic polynomial-based algorithms and (2) canonical labeling-based algorithms.

2.4.1 Characteristic Polynomial-based Algorithms

Harary made a conjecture in 1962 [Harary 1962] that two graphs G_1 and G_2 are isomorphic if their AMs (A_1 and A_2) have the same graph spectrum (eigenvalues or characteristic polynomials). However, this conjecture was immediately announced not true by a counterexample and then more counterexamples were provided [Harary *et al.* 1971]. In fact, it was already a matter of public record that the conjecture was not true, since in 1957 Collatz and Sinogowitz displayed two different trees with 8 nodes (see Figure 2.15) having the same characteristic polynomial, i.e., $P(\lambda) = \lambda^8 - 7\lambda^6 + 9\lambda^4$ [Collatz and Sinogowitz 1957]. Though this conjecture is not true for general graphs, it may be true for a restricted class of graphs [Harary 1962; Harary *et al.* 1971; Mowshowitz 1972]. In order to adopt the idea of characteristic polynomial to characterize graphs up to isomorphism, the generalized matrix functions or the *immanants* [Littlewood 1940] were used instead of AMs. The generalized matrix functions are defined and applied to AMs with the properties of all permutations of the symmetric group. The generalized matrix functions thus imply more information on a graph than the AM. However, counterexamples were found for the generalized characteristic polynomial approach [Turner 1968].

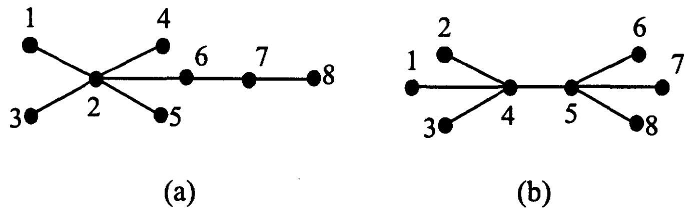


Figure 2.15 Two 8-vertex non-isomorphic trees having the same graph spectrum [Collatz and Sinogowitz 1957].

Many publications proposed characteristic polynomials to characterize the graphs representing various engineering applications [Uicker and Raicu 1975; Yan and Hall 1981, 1982; Spialter 1963, 1964; Balaban and Harary 1971; Kudo *et al.* 1973]. Uicker and Raicu [1975] applied the characteristic polynomials of two kinematic chains to determine if they are isomorphic. They presented a theorem that two kinematic chains that are isomorphic to each other have identical characteristic polynomials for their associated AMs. They proved that this is a necessary condition for isomorphism, but has not been proven a sufficient condition. Unfortunately, the converse of this theorem was not true since many counterexamples of kinematic chains having same characteristic polynomial but distinct structure were found. Figure 2.16 describes two distinct kinematic chains with the same characteristic polynomial, i.e., $P(\lambda) = \lambda^{12} - 16\lambda^{10} + 90\lambda^8 - 4\lambda^7 - 226\lambda^6 + 24\lambda^5 + 252\lambda^4 - 48\lambda^3 - 96\lambda^2 + 32\lambda$.

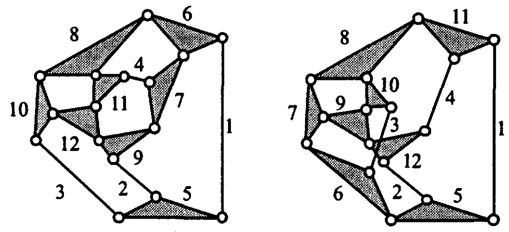


Figure 2.16 Two non-isomorphic kinematic chains both with the same graph spectrum.

In chemistry, Spialter asserted in 1963 that the characteristic polynomial of the atom connectivity matrix (closely related to AM) of the graph of a molecule structure was sufficient for the purposes of chemical documentation [Spialter 1963, 1964]. This assertion was refuted by Balaban and Harary [1971]. Some time after this refutation, in the same journal, Kudo [Kudo *et al.* 1973] claimed that the original assertion was correct but no proof was offered.

From all these findings, it is evident that the characteristic polynomial of a graph is an invariant of the graph but not complete invariants of the graph.

2.4.2 Canonical Labeling-based Algorithms

The *canonical label* of a graph means that the canonical label of one graph is the same as the canonical label of another graph if and only if these two graphs are isomorphic. Partitions are commonly used for generating the canonical label of a graph [Corneil and Gotlieb 1970; Schmidt and Druffel 1976; Babai *et al.* 1980; McKay 1981; Mittal 1988]. A *partition* of a set V is the set of disjoint non-empty subsets of V whose union is V. An *ordered partition* of V is a sequence $(V_1, V_2, \ldots, V_r)$ such that $\{V_1, V_2, \ldots, V_r\}$ is a partition of V. Corneil and Gotlieb [1970] gave an algorithm for canonical labeling:

(1) Partitioning the vertices of a graph into m cells where each *cell* consists of the vertices having the same degree; sorting these cells with the degree sequence in a descending order.

(2) To each vertex $u \in V$, associate a list $(a_1, a_2, \ldots, a_m)$ where a_i is the number of the vertices in cell i $(1 \leq i \leq m)$ which are adjacent to the vertex u; sorting the lists lexicographically for each cell in a descending order.

(3) If the associated lists of cell i $(1 \leq i \leq m)$ are not identical, then splitting cell i into n new cells such that each cell has the identical lists, and let $m \leftarrow m+n-1$, go to (2).

(4) If there exists only one vertex for each cell then the refinement is finished; otherwise go to (5).

(5) Defining the directed quotient graph of the graph based on the refined partition of V.

(6) For each cell i, if it has more than one vertex (but, of course, the lists are identical), then for a vertex v in cell i, splitting cell i into two cells, one which only contains the vertex v and one is the cell i without v (a new child for vertex v). Refining the partition (let $m \leftarrow m+1$) by going though step (2) to (5). Then backtracking to the partition without splitting the cell i, splitting the cell i with another vertex and refining the partition until each vertex in the cell i has been done. If not all directed quotient graphs generated from refining the cell i are identical, then partitioning the vertices in the cell i such that two vertices belong to the same subcell if and only if they possess identical vertex quotient graphs.

(7) Generating the terminal quotient graph based on the final partition.

Take the graph in Figure 2.5a as an example using the algorithm. The graph in Figure 2.5a is a regular graph with 3 degrees for each vertex. Therefore, from step (1) of the algorithm, the degree partition of *V* includes only one cell including 8 vertices. Assume that vertex 1 is split into a new cell. The division of cells is as follows:

Cell Index	Vertices
I	1
II	2, 3, 4, 5, 6, 7, 8

Following step (2), it is found that vertex 1 in cell I has only three connections with the vertices in cell II and thus, the list is (0, 3). Vertices 2, 5, and 8 in cell II are adjacent to vertex 1 in cell I and have two connections with the vertices in cell II, respectively. Therefore, these three vertices have the list (1, 2). Similarly, vertices 3, 4, 6, and 7 in cell II have the list (0, 3); see Table 2.2a. According to step (3), cell II can be splitted into two cells as they have two different lists, i.e., the list (1, 2) for vertices 2, 5, and 8 as well as the list (0, 3) for vertices 3, 4, 6, and 7. Table 2.2b shows the cells. Further partition can be done based on the cells shown in Table 2.2b. Table 2.2 shows a complete situation of partition.

Table 2.2 A case of partitioning procedure.

a. from step (2) of the algorithm			**b.** from step (3), splitting cell II in **a**		
Cell Index	Vertex	List	Cell Index	Vertex	List
I	1	(0, 3)	I	1	(0, 0, 3)
II	2	(1, 2)	II	5	(1, 2, 0)
	5	(1, 2)		2	(1, 1, 1)
	8	(1, 2)		8	(1, 1, 1)
	3	(0, 3)			
	4	(0, 3)	III	3	(0, 2, 1)
	6	(0, 3)		4	(0, 2, 1)
	7	(0, 3)		6	(0, 2, 1)
				7	(0, 2, 1)
c. splitting cell II in **b**			**d.** splitting cell IV in **c**		
Cell Index	Vertex	List	Cell Index	Vertex	List
I	1	(0, 1, 2, 0)	I	1	(0, 1, 2, 0, 0)
II	5	(1, 0, 0, 2)	II	5	(1, 0, 0, 2, 0)
III	2	(1, 0, 1, 1)	III	2	(1, 0, 1, 0, 1)
	8	(1, 0, 1, 1)		8	(1, 0, 1, 0, 1)
IV	4	(0, 1, 0, 2)	IV	4	(0, 1, 0, 1, 1)
	6	(0, 1, 0, 2)		6	(0, 1, 0, 1, 1)
	3	(0, 0, 1, 2)			
	7	(0, 0, 1, 2)	V	3	(0, 0, 1, 1, 1)
				7	(0, 0, 1, 1, 1)

This algorithm is based on the conjecture that the final partitioning resulting from the algorithm is the automorphism partition of V. Unfortunately, this conjecture has been shown to be not true. However, the backtracking idea has been widely used for uniquely coding graphs [Shah *et al.* 1974; Berztiss 1973] and labeling graphs [McKay 1981]. Also, Schmidt and Druffel [1976] and Mittal [1988] applied the backtracking algorithm for GI of two graphs in which the partition is based on the distance matrices of graphs instead of the degrees of the vertices of graphs.

It is well known that the canonical labeling algorithm Nauty presented by McKay [1981] is a very powerful and, currently, the preferred method for the GI problem. Conceptually, Nauty examines every automorphism of a graph and computes a canonical label. This label is simply the AM of the 'smallest' automorphism. The scheme used to define the smallest automorphism is to construct a label for a graph by concatenating the rows of its AM to form a binary number, computing the label of every automorphism of the graph, and then returning the smallest one. Two major operations are performed on partition in Nauty, i.e., refining a partition and generating the children of a partition. Indeed, the basic algorithm in Nauty is basically a partitioning procedure similar to the algorithm discussed before. Nauty recognizes an automorphism by checking if two distinct final partitions have the same AM after relabeling the vertices. Therefore, all automorphisms of a graph can be recognized during partitioning and the smallest one is the canonical label of the graph. It is noted that once Nauty has found an automorphism, it immediately puts the automorphism to work to try to prune the search space. This pruning thus evidently reduces the running time to the results.

In order to reduce the number of reordering, heuristic isomorphism procedures have been applied by employing properties that are invariant under GI [Unger 1964; Sussenguth 1965; Corneil and Kirkpatrick 1980]. For example, no isomorphism between two undirected graphs, G_1 and G_2, may map vertex u of G_1 onto vertex v of G_2 if the degree of *u* does not equal the degree of *v*. This rule has been applied in the above-mentioned partition algorithms, including in Nauty.

Several studies have been done for uniquely coding or labeling graphs in engineering such as the min code of kinematic chains [Ambekar and Agrawal 1987], Hamming number code [Rao and Raju 1991], topological ordering of vertices [Kim *et al.* 1992], degree code [Tang *et al.* 1993], feature code [Zhang and Li 1999], and so on. These approaches have not provided mathematic proof of their sufficient condition to general graphs.

2.4.3 Direct Enumeration of Graphs

Enumeration of all possible graphs or structures given certain conditions is a useful step in structure synthesis (see discussions in Chapter 1). These graphs or structures must be non-isomorphic to each other. *Indirect approach* to structure enumeration means that enumeration involves a procedure for detection of isomorphic graphs, and *direct approach* means that enumeration does not involve any procedure for detection of isomorphic graphs.

A typical example of using the direct approach is found in structure synthesis of kinematic chains [Crossley 1964, 1966; Davies and Crossley 1966; Mruthyunjaya 1979; Tischler *et al.* 1995; Rao and Deshmukh 2001]. Structural synthesis of all distinct possible kinematic chains with the specified number of links and degrees of freedom is useful in order to select the best possible chain for the special task at the conceptual stage of design. The method used by Crossley [1964, 1966] is based on intuition to enumerate kinematic chains, while Davies and Crossley [1966] used a so-called Franke's notation to describe kinematic chains. Mruthyunjaya [1979] proposed a transformation of binary chains to construct all possible chains with required links and degrees of freedom. Tischler *et al.* [1995] developed a new method to produce a complete list of chains. These methods may still produce isomorphic chains, and the test for isomorphism, sometimes, is needed.

Rao and Deshmukh [2001] proposed a method based on a basic matrix, loose matrix, and chain matrix to enumerate planar kinematic chains without the need to test isomorphisms. This method followed the fact that a planar closed kinematic chain can be viewed as a combination of two structures: the outmost closed polygon (basic loop) and the remainder. Since the basic loop is formed by removing the remainder from the kinematic chains, both the basic loop and the remainder would exist as free joints. The ordered free joints (basic vertices) of the basic loop are represented by distance matrix called *basic matrix* while the free joints (loose vertices) of the remainder are represented by distance matrix called *loose matrix. Chain matrix* is a sum of basic matrix and loose matrix. All the possible combinations can be generated by joining any loose vertex to any basic vertex while keeping the labels of the basic vertices invariant and varying the labels of the loose vertices. The author stated in that paper that the possibility of obtaining isomorphs is nil, but no proof has been given.

It is noted that all these direct methods lack mathematic evidence proving that isomorphic graphs do not exist. Therefore, the results from these methods are often doubted and have even been proven incorrect.

2.4.4 Evolutionary Computation Approach

Another commonly used method in structure synthesis is the so-called Evolutionary Algorithms (EA) [Antonsson and Cagan 2001] which are stochastic search methods that mimic the metaphor of natural biological evolution. Different EA have evolved during the last 40 years: Genetic Algorithms (GA), Evolutionary Strategies (ES), and Evolutionary Programming (EP). However, all were inspired by the same principle of natural evolution. A good introductory survey of them can be found in [Fogel 1994]. Among GA, ES, and EP, GA is perhaps the most widely known type of EA today and has been successfully applied to many science and engineering problems in various domains [Goldberg 1989]. Basically, the GA starts its evolution with a random generation of a population of *individuals* (also denoted as *chromosomes*), usually represented as strings or arrays of genes (a *gene* is the smallest building block of the solution). This population is submitted to an iterative process (each iteration of the search is called a *generation*) composed of three principal steps: evaluation (according to a *fitness function*), selection of the best individuals, and application of the genetic operators which include: *crossover* (reconfiguration of the chromosomes) and *mutation* (random change of components of a chromosome). This process is repeated until a defined termination criterion is reached.

The graph isomorphism problem could be set up with GAs as follows: mapping chromosome of two graphs A and B is an array of pairing genes. Each gene is a pair of two values that are the vertex number in A and the mapping vertex number in B. N initial chromosomes may be generated by randomly matching each vertex in A with a vertex having the same degree in B for N times. The correctness ratio of the edges matched between two graphs could be defined as the fitness function and evaluated for each chromosome. All chromosomes are put together onto a roulette wheel where each individual is assigned a sector of the roulette wheel proportional to its correctness ratio. The offspring for the next generation is selected by spinning N times the wheel. The higher the correctness ratio of a chromosome is, the higher the probability that the chromosome is selected. Crossover operator happens with a certain probability for the selected N chromosomes to partially exchange the mapping information of two chromosomes and create the new population. Mutation operator on the chromosomes after crossover is applied with a certain probability for randomly changing the mapping of a vertex. The mutated N chromosomes are then evaluated with the fitness function and an iterative process repeats until a termination is reached.

2.5 Discussion and Concluding Remarks

From the previous discussion, it can be concluded that the characteristic polynomial is an invariant but not complete invariants for isomorphism. This means that having the same characteristic polynomial is a necessary but not a sufficient condition of isomorphic graphs. Clearly, algorithms based on the characteristic polynomial may not always work for general graphs.

Nauty is a practical canonical labeling-based algorithm which is powerful and provides the preferred method so far for the GI problem. The reason this method out performs (in general) the other canonical labeling algorithms is that by concentrating on one graph at a time, ideas from the realm of group theory can be brought to bear on the problem, decreasing the running time. Besides, Nauty can provide all automorphisms of a graph. Roughly, the computational complexity of Nauty is $O(mn^3)$, where n is the number of the vertices of a graph but m is an integer number without a reasonable bound. One of the other advantages of Nauty is that there is no floating point computing needed.

In addition, few researchers used non-traditional approaches for solving GI problems, such as Hopfield neural networks [Li and Zhang 1998] and optimization algorithms [McGregor 1979]. Non-traditional approaches applied so far to GI problems, either neural networks or optimization algorithms, do not seem intuitively inviting [Fortin 1996]. Theoretically, the approach based on the Hopfield network cannot prove a polynomial worst case, since it is not possible to guarantee that it will always return the correct result, while the approach based on optimization algorithms converts a GI problem into a constraint satisfaction problem, and then uses specially tuned constraint algorithms. The latter approach does not seem inviting because the constraint satisfaction problem is known to be NP-hard, and the GI problem could very possibly be in P [Fortin 1996].

The direct approach to the structure synthesis may do well without testing for isomorphism. However, two reasons limit the application of these kinds of methods. One is that these methods have not yet been proven mathematically sufficient to eliminate isomorphic graphs completely. In fact, sometimes, the test for isomorphism has to be done. The other reason is that the applications of these methods are limited in mechanical engineering, in particular in structure synthesis of kinematic chains. The generality of these methods is very poor. For instance, though the test results of the method given by Rao and Deshmukh [2001] are correct for kinematic chains up to 10 links with 1 or 2 degree-of-freedom, there is no proof that no isomorphic chain can appear in another case, say 11 links. In fact, this method of determining both distinct basic loops and their distinct relabeling of the loose vertices for a given links and degrees of freedom itself involves the isomorphism problem.

In short, the graph isomorphism problems, as well as their related structure synthesis problems, have not been satisfactorily solved. Nauty seems to the best method available today, in the sense that no counterexample has been found and that it has the complexity of $O(mn^3)$. Clearly, it is worthwhile to explore a new approach.

CHAPTER 3
A NEW APPROACH FOR GRAPH ISOMORPHISM: EIGENSYSTEM

3.1 Introduction

Adjacency matrices (AMs) of graphs play a very important role in solving the graph isomorphism (GI) problem. Many algorithms developed for GI tests start from AMs of graphs. The proposed approach in this thesis also comes from the analysis of AMs of graphs. However, unlike all other approaches, the proposed approach transforms graphs (which are described by their AMs) into their quadric surfaces, and thus converts the problem of graph isomorphism into the problem of quadric surface comparison.

This chapter presents the fundamentals of this new approach. In particular, Section 3.2 introduces the mathematical background for the quadratic form and quadric surface. Section 3.3 discusses the relationship between the quadric surface and the graph. Section 3.4 discusses linear transformation on the quadric surface. Section 3.5 discusses the unique representation of the quadric surface. Section 3.6 gives a summary with discussion.

3.2 Quadratic Form and Quadric Surface

For an $n{\times}n$ real matrix $A = [a_{ij}]$, $i, j = 1, 2, \ldots n$, if matrix A is pre-multiplied and post-multiplied by a row matrix and a column matrix of n variables $x_1, x_2, \ldots x_n$, respectively, a function is given by

$$\begin{aligned} F &= \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ &= a_{11}x_1^2 + a_{12}x_2x_1 + \cdots + a_{1n}x_nx_1 + \\ &\quad + a_{21}x_1x_2 + a_{22}x_2^2 + \cdots + a_{2n}x_nx_2 + \\ &\quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ &\quad + a_{n1}x_1x_n + a_{n2}x_2x_n + \cdots + a_{nn}x_n^2 \end{aligned} \tag{3.1}$$

This function is called the *quadratic form*, and matrix A is called the *matrix of the quadratic form*. The terms on the principal diagonal of this square array involve the squares of the variables $x_1, x_2, \ldots x_n$; the remaining terms involve all the possible cross-products.

Letting the function F equal to 1 leads to the following equation:

$$\begin{aligned} F = &\ a_{11}x_1^2 + a_{12}x_2x_1 + \cdots + a_{1n}x_nx_1 + \\ &+ a_{21}x_1x_2 + a_{22}x_2^2 + \cdots + a_{2n}x_nx_2 + \\ &\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ &+ a_{n1}x_1x_n + a_{n2}x_2x_n + \cdots + a_{nn}x_n^2 \\ = &\ 1 \end{aligned} \tag{3.2}$$

Alternatively, Equation (3.2) can be stated as: finding the values for variables $x_1, x_2, \ldots x_n$ such that the quadratic form can maintain a unity. Geometrically, Equation (3.2) describes a surface, i.e., a *quadric surface*. In the three-dimensional space, there are three types of quadric surfaces: elipsoid, paraboloid, or hyperboloid. Furthermore, function F contains no linear terms, and thus the sign of the variables does not affect the function. This implies that any vector $\mathbf{x}$, defined by $(x_1\ x_2 \ldots x_n)$ and satisfying Equation (3.2), will have the same length, regardless of the sign of vector $\mathbf{x}$. Hence, the quadric surface is symmetrical about its origin. A surface of this kind is referred to as a *central* quadric surface. It has either hyperbolic or ellipsoidal characteristics, or both, depending on the values and the algebraic signs of the coefficients a_{ij} in function F.

For a given matrix, its quadratic form is unique; yet the converse is not true. That is to say, for a given quadratic form, there may be many matrices corresponding to it. For example, the following matrices of A

$$\begin{bmatrix}1 & 2 & 3\\ 2 & 3 & 1\\ 3 & 1 & 2\end{bmatrix}, \begin{bmatrix}1 & 1 & 1\\ 3 & 3 & 0\\ 5 & 2 & 2\end{bmatrix}, \begin{bmatrix}1 & 0 & 2\\ 4 & 3 & 1\\ 4 & 1 & 2\end{bmatrix}, \begin{bmatrix}1 & -1 & 3\\ 5 & 3 & 3\\ 3 & -1 & 2\end{bmatrix} \tag{3.3}$$

all satisfy the quadratic form

$$\begin{aligned} F &= x_1^2 + 3x_2^2 + 2x_3^2 + 4x_1x_2 + 6x_1x_3 + 2x_2x_3 \\ &= \begin{bmatrix}x_1 & x_2 & x_3\end{bmatrix} A \begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix} \end{aligned} \tag{3.4}$$

Yet, if the matrix is symmetric, the matrix and the quadratic form will have a one-to-one correspondence.

3.3 Graph versus Quadric Surface

Considering the AM of a graph as the coefficient matrix A of Equation (3.1), the quadratic form expressed by Equation (3.1) is thus the quadratic form expressing the graph. Accordingly, the quadric surface expressed by Equation (3.2) is the quadric surface expressing the graph. As such, variable x_i in Equation (3.1) denotes vertex i in the graph, and item $a_{ij}x_ix_j$ for $i, j = 1, 2, \ldots n$ represents a connection, with a weight value of a_{ij}, between vertices i and j. Further, assume that $x_1, x_2, \ldots x_n$ are the co-ordinates in a rectangular co-ordinate system. A graph with n vertices can be transformed into a quadric surface in the n-dimensional space, where each dimension represents a vertex of the graph.

Without loss of generality, the graph considered here is the undirected labeled graph (in Chapter 4, the extension to the directed graph will be given). The AM of an undirected graph is symmetrical; therefore, a graph and the quadric surface of the graph have a one-to-one correspondence. The question arises:

Can the graph isomorphism problem be converted into the quadric surface identification problem?

This thesis study was actually started with trying to answer the above question. The first challenge in answering this question is whether there is a canonical expression for the quadric surface corresponding to a graph. This challenge is important, as the AM of a graph is not invariant (i.e., different labelings can lead to different AMs). This can be illustrated further using an example. Two isomorphic graphs shown in Figure 2.5 have different AMs and have different expressions of the quadric surfaces, see Figure 3.1.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \neq \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{aligned} F_1 = {} & 2x_1x_2 + 2x_1x_5 + 2x_1x_8 + 2x_2x_3 + \\ & 2x_2x_8 + 2x_3x_4 + 2x_3x_6 + 2x_4x_5 + \\ & 2x_4x_6 + 2x_5x_6 + 2x_6x_7 + 2x_7x_8 \\ = {} & 1 \end{aligned} \quad \neq \quad \begin{aligned} F_2 = {} & 2x_1'x_2' + 2x_1'x_5' + 2x_1'x_8' + 2x_2'x_4' + \\ & 2x_2'x_6' + 2x_3'x_5' + 2x_3'x_7' + 2x_3'x_8' + \\ & 2x_4'x_6' + 2x_4'x_7' + 2x_5'x_6' + 2x_7'x_8' \\ = {} & 1 \end{aligned}$$

(a) the graph in Fig. 2.5a (b) the graph in Fig. 2.5b

Figure 3.1 The AMs and their quadric surfaces for the graphs shown in Figure 2.5.

Finding a canonical expression for a quadric surface can be mathematically stated as: finding a one-to-one mapping σ between $x_1, x_2, \ldots x_n$ and $x_1', x_2', \cdots x_n'$, i.e., $x_j' = \sigma(x_i)$ for all $i, j = 1, 2, \ldots n$. It is clear that when such a mapping exists, the identification of two quadric surfaces (i.e., examining whether they are actually the same) is easily done using their corresponding canonical expressions. The hope to have such a mapping is enlightened because mathematically, a linear transformation could serve as a mapping between the variables $x_1, x_2, \ldots x_n$ and $x'_1, x'_2, \cdots x'_n$.

3.4 Linear Transformations for the Quadric Surface

In the quadric surface theory, different coordinate reference systems correspond to different forms of the representation of the same quadric surface, i.e., different matrices A (see Equation (3.4) in preceding discussions). In general, cross product terms, i.e., $a_{ij}x_ix_j$ $(i \neq j)$, are present in the expression of a quadric surface. It corresponds that matrix A is not a diagonal matrix. The principal coordinate reference system is the one upon which the cross-product terms disappear. According to the quadric surface theory, there are one or more than one linear transformation that can find the principal coordinate reference system from any reference system.

Consider the quadratic form F given by Equation (3.1). Applying a linear transformation to variables $x_1, x_2, \ldots x_n$ in F leads to

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = C \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{3.5}$$

in which C is an arbitrary sequence matrix of order n and orthogonal, and $y_1, y_2, \ldots y_n$ are new variables. When matrix C is nonsingular (there is always an nonsingular orthogonal C), the new variables are uniquely related to the original variables $x_1, x_2, \ldots x_n$, and hence for given variables $x_1, x_2, \ldots x_n$, substituting Equation (3.5) into Equation (3.1) results in

$$[x_1 \quad x_2 \quad \cdots \quad x_n] = [y_1 \quad y_2 \quad \cdots \quad y_n]C^T \tag{3.6}$$

and

$$F = [y_1 \quad y_2 \quad \cdots \quad y_n]C^T AC\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [y_1 \quad y_2 \quad \cdots \quad y_n]\Lambda\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{3.7}$$

where

$$\Lambda = C^T AC \tag{3.8}$$

Matrix Λ in Equation (3.8) is a diagonal one [Guillemin 1949], i.e.,

$$\Lambda = C^T AC = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{3.9}$$

The quadratic form F expressed in terms of the new variables $y_1, y_2, \ldots y_n$ is

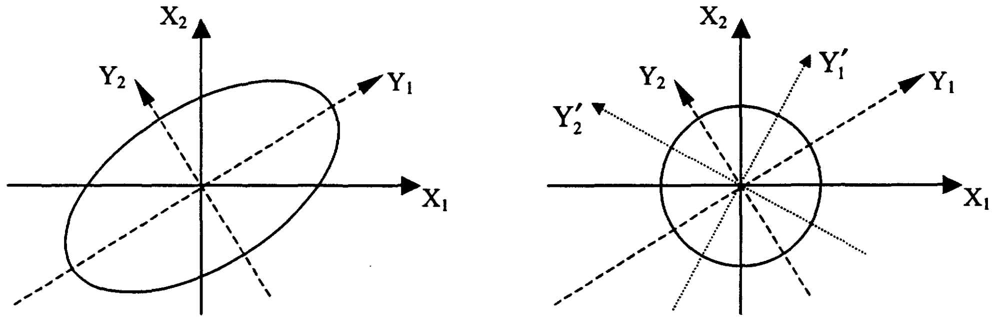
$$\begin{aligned} F &= [y_1 \quad y_2 \quad \cdots \quad y_n]\Lambda\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = [y_1 \quad y_2 \quad \cdots \quad y_n]\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \\ &= \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 \end{aligned} \tag{3.10}$$

Letting F=1 leads to

$$F = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \cdots + \lambda_n y_n^2 = 1 \tag{3.11}$$

Equation (3.11) is also called the *normal expression* of the quadric surface. In Equation (3.11), the principal axes $y_1, y_2, \ldots y_n$ of the quadric surface form a mutually orthogonal set, where $\frac{1}{\sqrt{|\lambda_1|}}, \frac{1}{\sqrt{|\lambda_2|}}, \cdots, \frac{1}{\sqrt{|\lambda_n|}}$ are called the lengths (or the *semiaxes*) of the quadric surface. Geometrically, the orthogonal matrix C in Equation (3.5) determines the *directions* of these semiaxes, while these semiaxes determine the shape of the quadric surface.

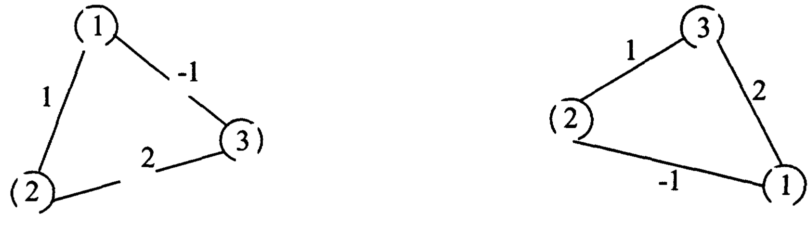
The quadric surface is ellipsoidal when all λ in Equation (3.11) are positive. When λ is negative the respective semiaxe is imaginary, and hence the respective surface, if containing both positive λ and negative λ, is of both the hyperbolic and ellipsoidal characteristics. When all λ are negative, the surface is an imaginary ellipse; it is customary to include this situation in the classification of completely ellipsoidal surfaces. It is noted that coincident λ in Equation (3.11) indicates a certain degree of the degeneracy of the corresponding quadric surface. For example, in the three-dimensional space, the coincidence of two λ in the case of an ellipsoid results in an ellipsoid of revolution, and the coincidence of all three λ results in a sphere. Furthermore, the coincidence of λ implies that the directions of the corresponding semiaxes are not unique. This can be illustrated using Figure 3.2. Figure 3.2a describes an ellipse and its unique directions of the semiaxes, while Figure 3.2b shows a circle, degenerated from the ellipse shown in Figure 3.2a, in which the directions of the semiaxes are not unique. This further implies that if there is no coincident λ in Equation (3.9), matrix C is unique; otherwise matrix C is not unique. This is because the column vectors of matrix C corresponding to λ are not unique.



(a) An ellipse and its semiaxes (b) A circle and its semiaxes

Figure 3.2 An ellipse, its circle degeneracy, and their semiaxes.

It is noted that the linear transformation does not change the shape of a quadric surface regardless of whether there are coincident λ. This means that the diagonal matrix Λ in Equation (3.9) is unique for a given matrix A. Therefore, the principal axes and the semiaxes of a quadric surface may be used as a canonical representation of a respective graph. An example is given in Figure 3.3 for illustrating this point.



(a) (b)

Figure 3.3 Two weighted graphs both with 3 vertices.

3.5 Canonical Representation of the Quadric Surface

Figures 3.3a and 3.3b show two graphs, respectively, and their isomorphisms are examined. The quadratic matrices for the graphs in Figure 3.3a and 3.3b are, respectively,

$$A=\begin{bmatrix}0 & 1 & -1\\ 1 & 0 & 2\\ -1 & 2 & 0\end{bmatrix},\quad B=\begin{bmatrix}0 & -1 & 2\\ -1 & 0 & 1\\ 2 & 1 & 0\end{bmatrix} \tag{3.12}$$

Their quadric surfaces F_a and F_b are, respectively,

$$\begin{aligned}F_a &= [x_1' \quad x_2' \quad x_3']A\begin{bmatrix}x_1'\\ x_2'\\ x_3'\end{bmatrix} = [x_1' \quad x_2' \quad x_3']\begin{bmatrix}0 & 1 & -1\\ 1 & 0 & 2\\ -1 & 2 & 0\end{bmatrix}\begin{bmatrix}x_1'\\ x_2'\\ x_3'\end{bmatrix}\\ &= 2x_1'x_2' - 2x_1'x_3' + 4x_2'x_3' = 1\end{aligned} \tag{3.13}$$

$$\begin{aligned}F_b &= [x_1'' \quad x_2'' \quad x_3'']B\begin{bmatrix}x_1''\\ x_2''\\ x_3''\end{bmatrix} = [x_1'' \quad x_2'' \quad x_3'']\begin{bmatrix}0 & -1 & 2\\ -1 & 0 & 1\\ 2 & 1 & 0\end{bmatrix}\begin{bmatrix}x_1''\\ x_2''\\ x_3''\end{bmatrix}\\ &= -2x_1''x_2'' + 4x_1''x_3'' + 2x_2''x_3'' = 1\end{aligned} \tag{3.14}$$

The two quadric surfaces have the same normal expression, i.e.,

$$F_a' = F_b' = 0.7321y_1^2 + 2.0000y_2^2 - 2.7321y_3^2 = 1 \tag{3.15}$$

Their linear transformation matrices are, respectively,

$$\begin{bmatrix}x_1'\\ x_2'\\ x_3'\end{bmatrix} = \begin{bmatrix}0.8881 & 0.0000 & -0.4597\\ 0.3251 & 0.7071 & 0.6280\\ -0.3251 & 0.7071 & -0.6280\end{bmatrix}\begin{bmatrix}y_1\\ y_2\\ y_3\end{bmatrix},\quad \begin{bmatrix}x_1''\\ x_2''\\ x_3''\end{bmatrix} = \begin{bmatrix}-0.3251 & 0.7071 & -0.6280\\ 0.8881 & 0.0000 & -0.4597\\ 0.3251 & 0.7071 & 0.6280\end{bmatrix}\begin{bmatrix}y_1\\ y_2\\ y_3\end{bmatrix}$$

Equation (3.15) implies that the quadric surface has the hyperbolic as well as ellipsoidal characteristics because each intersection of the surface along Y_3 axis is an ellipse, and each intersection along Y_1 or Y_2 axis is a hyperbola. Figure 3.4 illustrates the quadric surface and the relationships among the co-ordinate systems $Y_1Y_2Y_3$, $X_1'X_2'X_3'$, and $X_1''X_2''X_3''$. It can be seen from this figure that after the linear transformations these two quadric surfaces F_a and F_b overlap completely in the $Y_1Y_2Y_3$ system, and also that there exists a one-to-one mapping between two co-ordinate systems $X_1'X_2'X_3'$ and $X_1''X_2''X_3''$, that is, $X_1' \leftrightarrow X_2''$, $X_2' \leftrightarrow X_3''$, and $X_3' \leftrightarrow X_1''$. Therefore, the two graphs shown in Figure 3.3 are isomorphic, and further there is only one such a mapping. Note that variables X_1, X_2, and X_3 correspond to the vertices of a graph. The one-to-one mapping as shown implies that there are correspondences in vertices between the two graphs, i.e., $A(1) \leftrightarrow B(2)$, $A(2) \leftrightarrow B(3)$, and $A(3) \leftrightarrow B(1)$, where the number within the parenthesis means the vertex label.

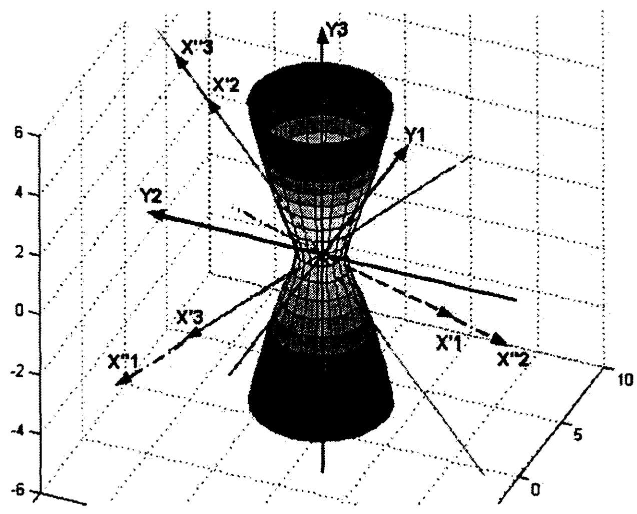


Figure 3.4 The quadric surface of the graphs having all distinct semiaxes.

There is an exception to the above procedure for graph isomorphism, where some λ are coincident. In this case, the linear transformation is not unique; see Figure 3.2. An example is given to illustrate this exception. Figure 3.5 shows two isomorphic weighted graphs. The quadric surfaces F_a and F_b correspond to the graphs shown in Figure 3.5a and 3.5b (respectively), and their normal expressions are, corresponding

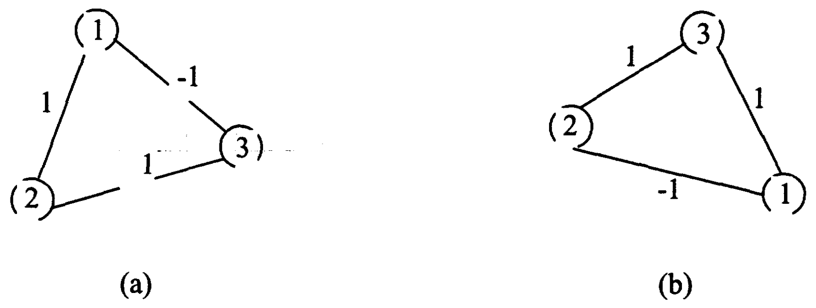


Figure 3.5 Two isomorphic weighted graphs both with 3 vertices.

$$F_a = \begin{bmatrix} x_1' & x_2' & x_3' \end{bmatrix} \begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = 2x_1'x_2' - 2x_1'x_3' + 2x_2'x_3' = 1 \tag{3.16}$$

$$F_b = \begin{bmatrix} x_1'' & x_2'' & x_3'' \end{bmatrix} \begin{bmatrix} 0 & -1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1'' \\ x_2'' \\ x_3'' \end{bmatrix} = -2x_1''x_2'' + 2x_1''x_3'' + 2x_2''x_3'' = 1 \tag{3.17}$$

$$F_a' = F_b' = y_1^2 + y_2^2 - 2y_3^2 = 1 \tag{3.18}$$

From Equation (3.18), the normal quadric surface is a hyperboloid of revolution where the semiaxes on Y_1 and Y_2 are the same. The quadric surface and the relationships among the co-ordinate systems $X_1'X_2'X_3'$, $X_1''X_2''X_3''$, and $Y_1Y_2Y_3$ are shown in Figure 3.6, where only the semiaxis on X'_2 is coincident with that on X''_3 (i.e., $A(2) \leftrightarrow B(3)$). The other two axes are not coincident. This does not, however, imply that these two graphs are not isomorphic. In fact, there are two isomorphic mappings for the other two vertices between the two graphs (in Figure 3.5), that is, $A(1) \leftrightarrow B(1)$ and $A(3) \leftrightarrow B(2)$, or $A(1) \leftrightarrow B(2)$ and $A(3) \leftrightarrow B(1)$. It is clear that further analysis is needed for this situation to determine whether there is an isomorphic mapping among the other vertices between the two graphs.

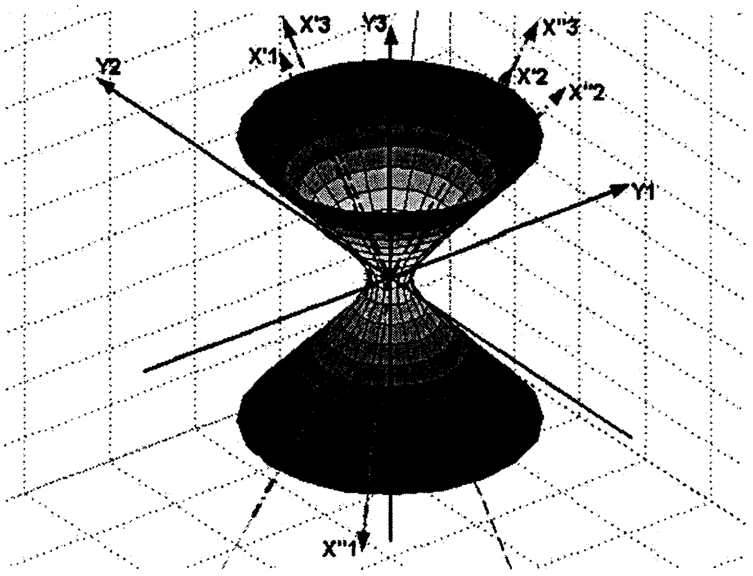


Figure 3.6 The quadric surface of the graphs having coincident semiaxes.

3.6 Quadric Surface and Eigensystem

The kernel of the proposed approach is to transform the quadric surfaces of two graphs into their normal expressions and then to examine whether the quadric surfaces have the same semiaxes and whether the original co-ordinate axes of the quadric surfaces are coincident to each other in the new co-ordinate system. The procedure can also be considered as comparing diagonal matrices Λ and linear transformation matrices C in Equation (3.9) of two graphs.

Since linear transformation matrix C given by Equation (3.9) is a nonsingular orthogonal one, it has

$$C^T = C^{-1} \tag{3.19}$$

Submitting Equation (3.19) into Equation (3.9) yields

$$A = C\Lambda C^T \tag{3.20}$$

in which A is a symmetrical matrix and Λ is a diagonal matrix.

Indeed, Equation (3.20) describes an eigendecomposition of A where λ in diagonal matrix Λ is the *eigenvalues* of A, and the i^{th} column in the transformation matrix C is the *eigenvector* corresponding to eigenvalue λ_i. When both the eigenvalues and the corresponding eigenvectors of a matrix are put together this is named the *eigensystem* of the matrix. Therefore, the proposed approach for GI can be geometrically interpreted using the linear transformations of quadric surfaces and realized using the eigendecomposition of the AMs of graphs. In this sense, the proposed approach is called the Eigensystem approach. Figure 3.7 illustrates the framework of the Eigensystem approach for graph isomorphism.

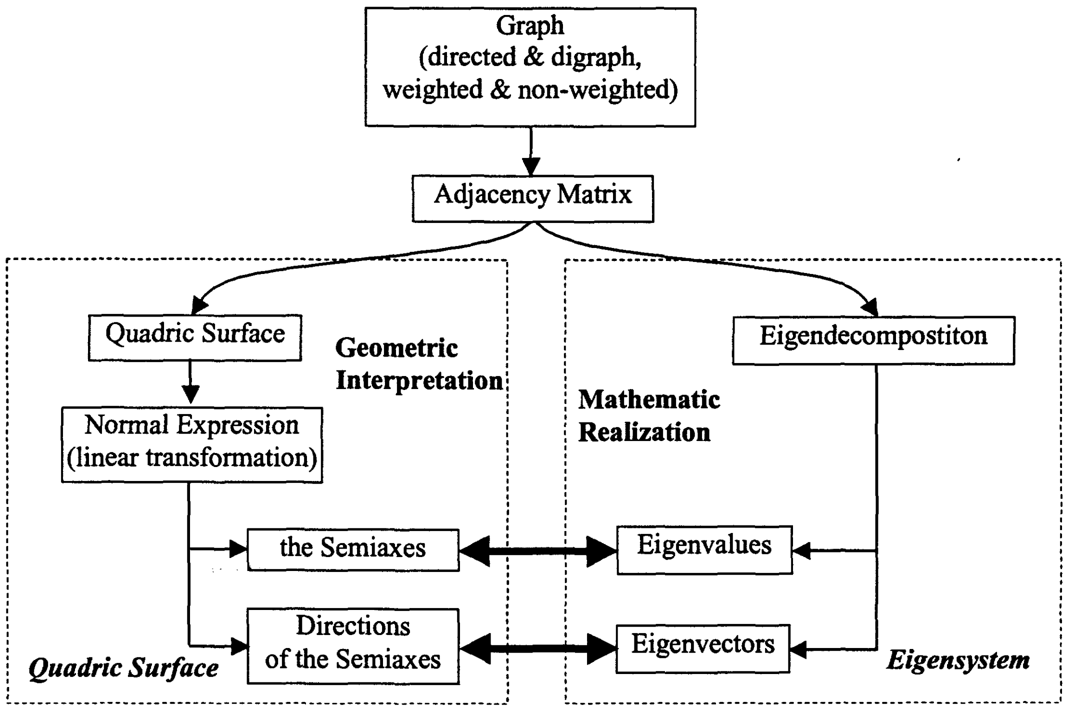


Figure 3.7 The framework of the Eigensystem approach for graph isomorphism.

3.7 Summary and Discussion

The concepts of quadratic forms and quadric surfaces were applied to develop a new approach called the Eigensystem approach. In this approach, an undirected graph with *n* vertices was transformed into a quadric surface in an *n*-dimensional space, and the adjacency matrix of the graph becomes the matrix of the quadratic form. This transformation is unique for a given undirected labeled graph. The problem of solving the graph isomorphism becomes the problem of comparing quadric surfaces. The general quadratic form of a graph was further converted into its normal expression. This conversion process leads to two sets of parameters: the principal reference system and the semiaxes. If two quadric surfaces have different semiaxes (i.e., different geometrical shapes), the two graphs, corresponding to the two quadric surfaces, are not isomorphic. If two quadric surfaces have the same semiaxes, the two graphs may be isomorphic. In this case, further comparison of the direction of the semiaxes (or the principal axes) between the two quadric surfaces is needed. Two graphs are isomorphic if the directions of the semiaxes of their quadric surfaces are coincident to each other. In other words, two graphs are isomorphic if their corresponding quadric surfaces have the same geometrical shape and coincident principal axes. It is noted that the geometrical shape of a quadric surface is unique and unchangeable during the linear transformations for the normal expression, and this kind of linear transformation is unique unless there is at least a semiaxis whose length is coincident with the length of one of the other semiaxes. In the geometrical sense, the situation can easily be interpreted by the case of degenerating an ellipsoid into a sphere where the lengths on the three axes are the same, and thus the co-ordinate system having the normal expression is not unique.

Finding the normal expression of a quadric surface and the linear transformation of the co-ordinate system from its original expression to its normal expression is equivalent to eigendecomposing the matrix of the quadric form. The lengths of the semiaxes of the quadric surface correspond to the eigenvalues of the matrix, and the directions of their semiaxes are the eigenvectors corresponding to the eigenvalues. In short, the graph isomorphism problem could geometrically be interpreted by the quadric surfaces of the graphs and solved by the comparison of the eigensystems of the graphs.

The Eigensystem approach goes beyond the characteristic polynomial approach which is only determined by the eigenvalues of the AM of a graph. The information of the eigenvectors contributes to a further differentiation of graphs. However, when there are coincident eigenvalues in two graphs, the corresponding eigenvectors are not unique. In this case, if the principal axes of two quadric surfaces are coincident, the two corresponding graphs are isomorphic, but if the principal axes of two quadric surfaces are not coincident, the two graphs may also be isomorphic. This situation introduces a challenge. The reminder of the thesis will address the challenge and present algorithms for implementing the Eigensystem approach.

CHAPTER 4
ALGORITHM FOR GRAPH ISOMORPHISMS

4.1 Introduction

In Chapter 3, a new approach called the Eigensystem approach was described. In this approach, a graph is viewed as a quadratic surface, and a canonical representation of the graph may be developed through the canonical representation of the quadratic surface. In this chapter, algorithms for implementing this new approach into computer codes are developed. The presentation of the algorithms will start with simple situations and then move to more complex situations. In particular, Section 4.2 presents algorithm I for the graphs having all distinct eigenvalues. Section 4.3 presents algorithm II for the graphs having part of distinct eigenvalues (at least one distinct eigenvalue). In order to handle the special situation in which all eigenvalues of graphs are coincident, Section 4.4 presents a new matrix representation for the graph called 'adjusted adjacency matrix' (AAM). AAM ensures that there is at least one distinct eigenvalue. By replacing AM with AAM, algorithms I and II can then work for all the cases. Section 4.5 introduces the method based on algorithm I and II to solve the graph isomorphism (GI) problem for digraphs. The computational complexity of the Eigensystem approach is analyzed in Section 4.6. Finally, Section 4.7 gives the discussion and concluding remarks.

4.2 Algorithm I – All Eigenvalues Are Distinct

4.2.1 Theorem 1

Without loss of generality, assume that the graph concerned is an undirected, weighted with positive values, and labeled graph. The adjacency matrix of this kind of graph is thus a nonnegative symmetrical matrix, and each entry on the principal diagonal of the AM is zero. Let A be the AM of a graph with n vertices, $\mathbf{x}$ a nonzero n by 1 vector (a column vector), and λ a scalar, such that $A\mathbf{x}=\lambda\mathbf{x}$. Then λ is an eigenvalue of A, and $\mathbf{x}$ is an eigenvector of A corresponding to λ. $(\lambda, \mathbf{x})$ is referred to as an *eigenpair* of A. The factorization $A=X\Lambda X^T$ is the eigendecomposition of A where $\Lambda=\text{diag}(\lambda_1, \lambda_2, \ldots\lambda_n)$ is the *eigenvalue matrix* of A and $X=[\mathbf{x}_1\ \mathbf{x}_2 \ldots \mathbf{x}_n]$ is the *eigenvector matrix* of A, respectively. The collection of both Λ and X is the *eigensystem* of A.

Since A is a real symmetrical matrix, all values of eigenpairs are real. When an eigenvalue is distinct from all other eigenvalues, its normalized eigenvector is unique [Ortega 1987; Bai *et al.* 2000; Liu and Lai 2000]. This situation is called a unique eigenpair.

For the unique eigenpair, the component in each eigenvector has a one-to-one correspondence with the vertex in a graph. In Chapter 3, the unique eigenvectors were used for identification of graph isomorphism. Assume that there is a row permutation matrix P which exchanges the rows of an adjacency matrix A between the i^{th} and j^{th}. PAP^T thus exchanges both the rows and columns of A between the i^{th} and j^{th}, which further corresponds to the exchange of labels of the i^{th} vertex and the j^{th} vertex. It is further noted that

$$PAP^T=P(X\Lambda X^T)P^T=(PX)\,\Lambda(\,PX)^T.$$

Here, PX exchanges the rows of eigenvector matrix X between the i^{th} and j^{th}. This means that all eigenvectors in X have to exchange the components between the i^{th} and j^{th}, while the i^{th} vertex and the j^{th} vertex in a graph are exchanged by their labeling.

If two graphs are isomorphic, the eigenvalues of the two respective AMs should be the same, and the unique eigenvectors corresponding to the distinct eigenvalues should be equivalent. This leads to the following theorem of the Eigensystem approach:

Theorem 1. *Two graphs G_a and G_b, both with distinct eigenvalues, are isomorphic if and only if they have the same graph spectrum and there exists a mapping ϕ for the vertices from G_b onto G_a such that $X_a=\phi X_b S$ between the eigenvectors of X_a and the eigenvectors of X_b, where S is a diagonal matrix with ± 1 on the diagonal entry.*

Theorem 1 gives a necessary and sufficient condition for identification of the graph isomorphism when two graphs have distinct eigenvalues. The following is the proof of Theorem 1.

Proof of necessity:

Suppose that A and B are the AMs of two graphs G_a and G_b, respectively, and the two graphs are isomorphic. The goal of the proof is to lead to the expression $X_a=\phi X_b S$.

First, according to the definition of graph isomorphism, there should be the following equation

$$A = PBP^T \tag{4.1}$$

where P is an elementary permutation matrix.

Second, the characteristic polynomials of A and B are, respectively,

$$\begin{aligned} \det(A - \lambda I) &= \det(PBP^T - \lambda I) = \det(PBP^T - \lambda(PIP^T)) = \det(P(B - \lambda I)P^T) \\ &= \det(P)\det(B - \lambda I)\det(P^T) = \det(B - \lambda I) \end{aligned} \tag{4.2}$$

in which $P = PI$ and $I = PP^T = PIP^T$.

Equation (4.2) indicates that two isomorphic graphs have the same characteristic polynomial and, thus, have the same eigenvalues $\Lambda = \Lambda_a = \Lambda_b$ (graph spectrum).

Third, A and B are eigendecomposed into, respectively,

$$\begin{cases} A = X_a \Lambda X_a^{\ T} \\ B = X_b \Lambda X_b^{\ T} \end{cases} \tag{4.3}$$

Substituting Equation (4.3) into Equation (4.1) yields

$$X_a \Lambda X_a^{\ T} = P(X_b \Lambda X_b^{\ T})P^T = (PX_b)\Lambda(PX_b)^T \tag{4.4}$$

Since each eigenvalue of Λ is distinct from all other eigenvalues, its corresponding normalized eigenvector must be unique. This means that both the eigenvector matrices X_a and X_b are unique.

Fourth, the uniqueness of a normalized eigenvector is established upon its scalar, i.e., regardless of the direction of the eigenvector. For instance, if there exists a normalized eigenvector **x** for a distinct λ such that

$$A\mathbf{x}=\lambda\mathbf{x}, \tag{4.5}$$

the vector -**x** is also an eigenvector corresponding to λ. Define a matrix S as follows

$$S=\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_n \end{bmatrix} \tag{4.6}$$

in which $s_1, s_2 \cdots s_n$ can only be ± 1. Therefore, if X is the eigenvector matrix related to Equation (4.5), XS should also be a valid eigenvector matrix. The following deduction shows this point.

$$\begin{aligned} A &= (XS)\Lambda(XS)^T = XS\Lambda S^T X^T = X(S\Lambda S^T)X^T \\ &= X\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & s_n \end{bmatrix}\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & s_n \end{bmatrix}^T X^T \\ &= X\begin{bmatrix} s_1^2\lambda_1 & 0 & \cdots & 0 \\ 0 & s_2^2\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & & & s_n^2\lambda_n \end{bmatrix}X^T = X\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}X^T \\ &= X\Lambda X^T = A \end{aligned} \tag{4.7}$$

Hence, from Equation (4.4) there should be

$$X_a=PX_bS \tag{4.8}$$

Take ϕ=P. This completes the proof of the necessity of Theorem 1. ■

Proof of sufficiency:

Let two graphs have the same spectrum, i.e., $\Lambda_a = \Lambda_b$, and their eigenvector matrices have the relationship $X_a = PX_bS$ where P is a row permutation matrix and S is a sign matrix defined in Equation (4.6). Matrix A can be eigendecomposed into

$$\begin{aligned} A &= X_a\Lambda_aX_a^T = (PX_bS)\Lambda_b(PX_bS)^T = PX_b\left(S\Lambda_bS^T\right)X_b^TP^T \\ &= PX_b\Lambda_bX_b^TP^T = P\left(X_b\Lambda_bX_b^T\right)P^T = PBP^T \end{aligned} \tag{4.9}$$

Equation (4.9) indicates that B can be transformed into the same matrix as A after performing a row and column permutation. Therefore, graph G_a is isomorphic to graph G_b. This completes the proof of sufficiency of Theorem 1. ■

According to Theorem 1, the procedure for the identification of graph isomorphism for each graph having distinct eigenvalues involves two steps: (1) determining if the spectrums of two graphs are the same, and (2) finding a possible mapping ϕ (permutation matrix P) between unique eigenvectors of two graphs. The comparison of the spectrums of two graphs can be performed by sorting the eigenvalues of each spectrum in an ascending order (or in a descending order), respectively, and then by comparing the eigenvalues one by one between the two sorted sets. The following is algorithm I-1 which compares the spectrums of two graphs.

I-1 ***comparison of the spectrums of two graphs* (*A, B*)**

```
1.  calculating eigenvalues $\Lambda_a$ and $\Lambda_b$ of matrices $A$ and $B$, respectively
2.  sorting $\Lambda_a$ and $\Lambda_b$ in an ascending order, e.g., $\Lambda_a'$ and $\Lambda_b'$, respectively
3.  $\lambda_0 = \varnothing$
4.  for $i$=1, 2, ...$n$, do
```

```
5.      comparing if $\lambda_i'^a$ in $\Lambda_a'$ is equal to $\lambda_i'^b$ in $\Lambda_b'$
6.      if $\lambda_i'^a$ is not equal to $\lambda_i'^b$ then stop with ‘non-isomorphic’
7.      if $\lambda_i'^a$ is equal to $\lambda_{i-1}$ then mark both $\lambda_{i-1}'^a$ and $\lambda_i'^a$ as ‘coincident’
8.      $\lambda_i = \lambda_i'^a$
9.  return ‘same graph spectrum’
```

Finding a permutation matrix P between two eigenvector matrices is a complicate task. There are primarily four steps in finding P: sorting, comparing, matching, and intersecting. These steps are described below.

(1) Sorting the components of the eigenvectors $\mathbf{x}_i^a$ in X_a, $\mathbf{x}_i^b$ and $-\mathbf{x}_i^b$ in X_b, which correspond to λ_i, in ascending orders (or in descending orders); this results in $\mathbf{x}_i'^a$, $\mathbf{x}_i'^b$ and $-\mathbf{x}_i'^b$, respectively.

(2) Comparing the sorted components one by one between two pairs of the eigenvectors ($\mathbf{x}_i'^a$, $\mathbf{x}_i'^b$) and ($\mathbf{x}_i'^a$, $-\mathbf{x}_i'^b$), respectively. If both of these two pairs are different, a kind of mapping ϕ in Theorem 1 does not exist. Appling Theorem 1, the two graphs are non-isomorphic.

(3) Finding a mapping between the vertex of graph A and the vertex of graph B using the sorted eigenvectors as a mediate because the component index in the eigenvector corresponds to the vertex labeling in the graph (see previous discussion). Denote such a mapping ϕ_{i+}, related to the pair ($\mathbf{x}_i^a$, $\mathbf{x}_i^b$), and ϕ_{i-}, related to the pair ($\mathbf{x}_i^a$, $-\mathbf{x}_i^b$).

(4) Intersecting mappings ϕ_{i+} and ϕ_{i-}, respectively, with previous common mapping set Φ_{i-1} to produce current common mapping Φ_i, i.e., $\Phi_i = (\phi_{i+} \cap \Phi_{i-1}) \cup (\phi_{i-} \cap \Phi_{i-1})$. If common mapping Φ_i is empty, two graphs are non-isomorphic; otherwise, nonempty mapping Φ_n is permutation matrix P.

The following is algorithm I-2 which finds permutation matrix P or Φ_n:

I-2 *finding permutation matrix of two graphs* (*A*, *B*)

```
1.  calculating eigenvectors $X_a$ and $X_b$ of matrices $A$ and $B$, respectively
2.  $\Phi_0 = 1$
3.  for each distinct eigenvalue $\lambda_i$, $i$=1, 2,...$r$ ($r \le n$), do
4.      sorting eigenvector $\mathbf{x}_i^a$ in $X_a$ in an ascending order, e.g., $\mathbf{x}_i'^a$
5.      sorting eigenvectors $\mathbf{x}_i^b$ and $-\mathbf{x}_i^b$ in $X_b$ in an ascending order, e.g., $\mathbf{x}_i'^b$ and $-\mathbf{x}_i'^b$,
        respectively
6.      $x_0' = \varnothing$; $\phi_{i+} = \varnothing$
7.      for each component $x_{ij}'^a$ in $\mathbf{x}_i'^a$ and $x_{ij}'^b$ in $\mathbf{x}_i'^b$, $j \in [1, n]$, do
8.          if $x_{ij}'^a \ne x_{ij}'^b$ then break with $\phi_{i+} = \varnothing$
9.          if $x_{ij}'^a = x_{j-1}'$ then mark the $j$th component of $\mathbf{x}_i'^a$ as the same mapping group as
            the ($j$-1)th component
10.         $\phi_{i+} \leftarrow$ a mapping pair of the original component indices of $x_{ij}'^a$ and $x_{ij}'^b$ in $\mathbf{x}_i^a$
            and $\mathbf{x}_i^b$, respectively
11.     $x_0' = \varnothing$; $\phi_{i-} = \varnothing$
12.     for each component $x_{ij}'^a$ in $\mathbf{x}_i'^a$ and $-x_{ij}'^b$ in $-\mathbf{x}_i'^b$, $j \in [1, n]$, do
13.         if $x_{ij}'^a \ne -x_{ij}'^b$ then break with $\phi_{i-} = \varnothing$
```

```
14.         if $x'^a_{ij} = x'_{j-1}$ then mark the $j$th component of $\mathbf{x}'^a_i$ as the same mapping group as
            the $(j-1)$th component
15.         $\phi_{i-} \leftarrow$ a mapping pair of the original component indices of $x'^a_{ij}$ and $-x'^b_{ij}$ in
            $\mathbf{x}^a_i$ and $-\mathbf{x}^b_i$, respectively
16.     if $\phi_{i+} = \varnothing$ and $\phi_{i-} = \varnothing$ then stop with 'non-isomorphic'
17.     $\Phi_i = (\Phi_{i-1} \cap \phi_{i+}) \cup (\Phi_{i-1} \cap \phi_{i-})$
18.     if $\Phi_i = \varnothing$ then stop with 'non-isomorphic'
19. if all eigenvalues are distinct then stop with 'isomorphic graphs'
20. return $\Phi_n$
```

The following examples provide an illustration for algorithms I-1 and I-2.

4.2.2 Example: Non-cospectral Graphs

Figure 4.1 illustrates two graphs with 8 vertices.

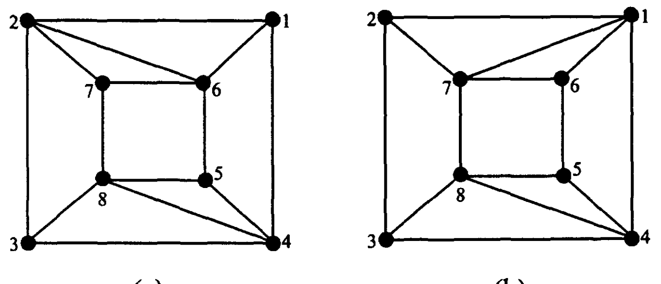


Figure 4.1 Two 8-vertex non-isomorphic graphs having different graph spectrums.

According to algorithm I-1, the eiegnvalues of these two graphs are, respectively,

Λ_a=diag(-1.6180, 0.6180, 0.6180, -1.6180, -0.6180, 1.6180, -2.5414, 3.5414)

Λ_b=diag(-1.0000, 1.0000, -0.5616, 0.0000, -2.0000, -2.5616, 1.5616, 3.5616)

Sorting Λ_a and Λ_b in ascending orders, respectively, yields

Λ_a'=diag(-2.5414, -1.6180, -1.6180, -0.6180, 0.6180, 0.6180, 1.6180, 3.5414)

Λ_b'=diag(-2.5616, -2.0000, -1.0000, -0.5616, 0.0000, 1.0000, 1.5616, 3.5616)

Comparing Λ_a' with Λ_b' one by one concludes that these two graphs have different spectrums, and thus they are non-isomorphic according to Theorem 1.

4.2.3 Examples: Cospectral Graphs

Figure 4.2 shows a pair of graphs with 12 vertices. Their graph spectrums Λ_a and Λ_b, as well as sorted spectrums Λ_a' and Λ_b' are, respectively,

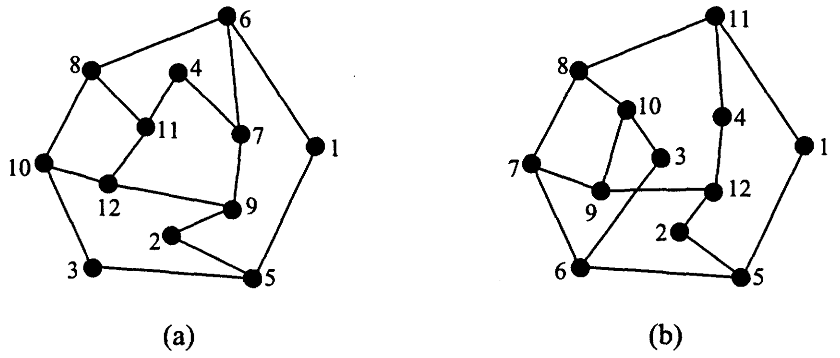


Figure 4.2 Two 12-vertex non-isomorphic cospectral graphs.

Λ_a=diag(0.6350, 0.4150, 1.4142, 1.5713, 1.7668, 0.0000, -0.9815, -1.5382, -1.4142, -2.0000, 2.7580, -2.6264)

Λ_b=diag(0.0000, -2.0000, 0.6350, 0.4150, 1.5713, 1.4142, 1.7668, -0.9815, -1.4142, -1.5382, 2.7580, -2.6264)

$\Lambda_a' = \Lambda_b'$ =diag(-2.6264, -2.0000, -1.5382, -1.4142, -0.9815, 0.0000, 0.4150, 0.6350, 1.4142, 1.5713, 1.7668, 2.7580)

Since these two graphs are cospectral and have all distinct eigenvalues, algorithm I-2 is to be applied for further determining whether they are isomorphic. Suppose that $\mathbf{x}_i^a$ ($\mathbf{x}_i^b$) represents the eigenvector of the graph shown in Figure 4.2a (4.2b), corresponding to λ_i in Λ_a' (Λ_b'). According to algorithm I-2, part of the procedure of finding the permutation matrix P is demonstrated in Figure 4.3. The 10th component (–0.3821) in $\mathbf{x}_1^a$ is the smallest one and thus ordered as the first component in $\mathbf{x}_1'^a$; while the 8th (9th) component (–0.3795) in $\mathbf{x}_1^b$ is the smallest one and thus ordered as the first (second) component in $\mathbf{x}_1'^b$. Since the first component in $\mathbf{x}_1'^a$ is not equal to the first one in $\mathbf{x}_1'^b$, there is no mapping ϕ_{1+} between $\mathbf{x}_1'^a$ and $\mathbf{x}_1'^b$. For the same reason, there is also no mapping ϕ_{1-} between $\mathbf{x}_1'^a$ and $-\mathbf{x}_1'^b$. This fact, i.e., both the mapping ϕ_{1+} and ϕ_{1-} are empty, results in $\Phi_1=\varnothing$. It implies that there is no permutation matrix P existing between these two graphs. Therefore, these two graphs are not isomorphic.

Figure 4.4 shows another pair of graphs with 17 vertices. Their graph spectrums Λ_a and Λ_b, as well as the ordered spectrums Λ_a' and Λ_b', are, respectively,

No.	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1+}	$\mathbf{x}_1'^b$	$\mathbf{x}_1^b$	No.
1	0.2192	-0.3821		-0.3795	-0.2192	1
2	0.2192	-0.3157		-0.3795	-0.2192	2
3	0.2445	-0.3157		-0.2445	-0.0260	3
4	0.0260	-0.2989		-0.2305	-0.2445	4
5	-0.2600	-0.2600		-0.2192	0.2547	5
6	-0.3157	0.0260	$\varnothing$	-0.2192	-0.2305	6
7	0.2305	0.2192		-0.0260	0.3767	7
8	0.3795	0.2192		0.2547	-0.3795	8
9	-0.3157	0.2305		0.2989	-0.3795	9
10	-0.3821	0.2445		0.3210	0.2989	10
11	-0.2989	0.3795		0.3210	0.3210	11
12	0.3795	0.3795		0.3767	0.3210	12

No.	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1-}	$-\mathbf{x}_1'^b$	$-\mathbf{x}_1^b$	No.
1	0.2192	-0.3821		-0.3767	0.2192	1
2	0.2192	-0.3157		-0.3210	0.2192	2
3	0.2445	-0.3157		-0.3210	0.0260	3
4	0.0260	-0.2989		-0.2989	0.2445	4
5	-0.2600	-0.2600		-0.2547	-0.2547	5
6	-0.3157	0.0260	$\varnothing$	0.0260	0.2305	6
7	0.2305	0.2192		0.2192	-0.3767	7
8	0.3795	0.2192		0.2192	0.3795	8
9	-0.3157	0.2305		0.2305	0.3795	9
10	-0.3821	0.2445		0.2445	-0.2989	10
11	-0.2989	0.3795		0.3795	-0.3210	11
12	0.3795	0.3795		0.3795	-0.3210	12

Figure 4.3 The procedure of finding mapping Φ_1 for the graphs shown in Figure 4.2.

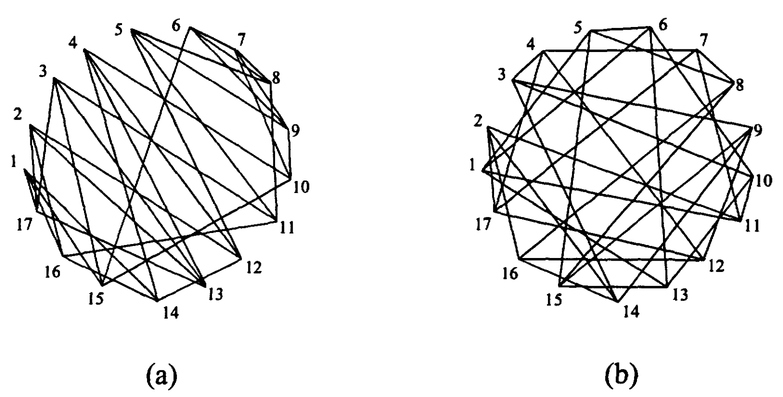


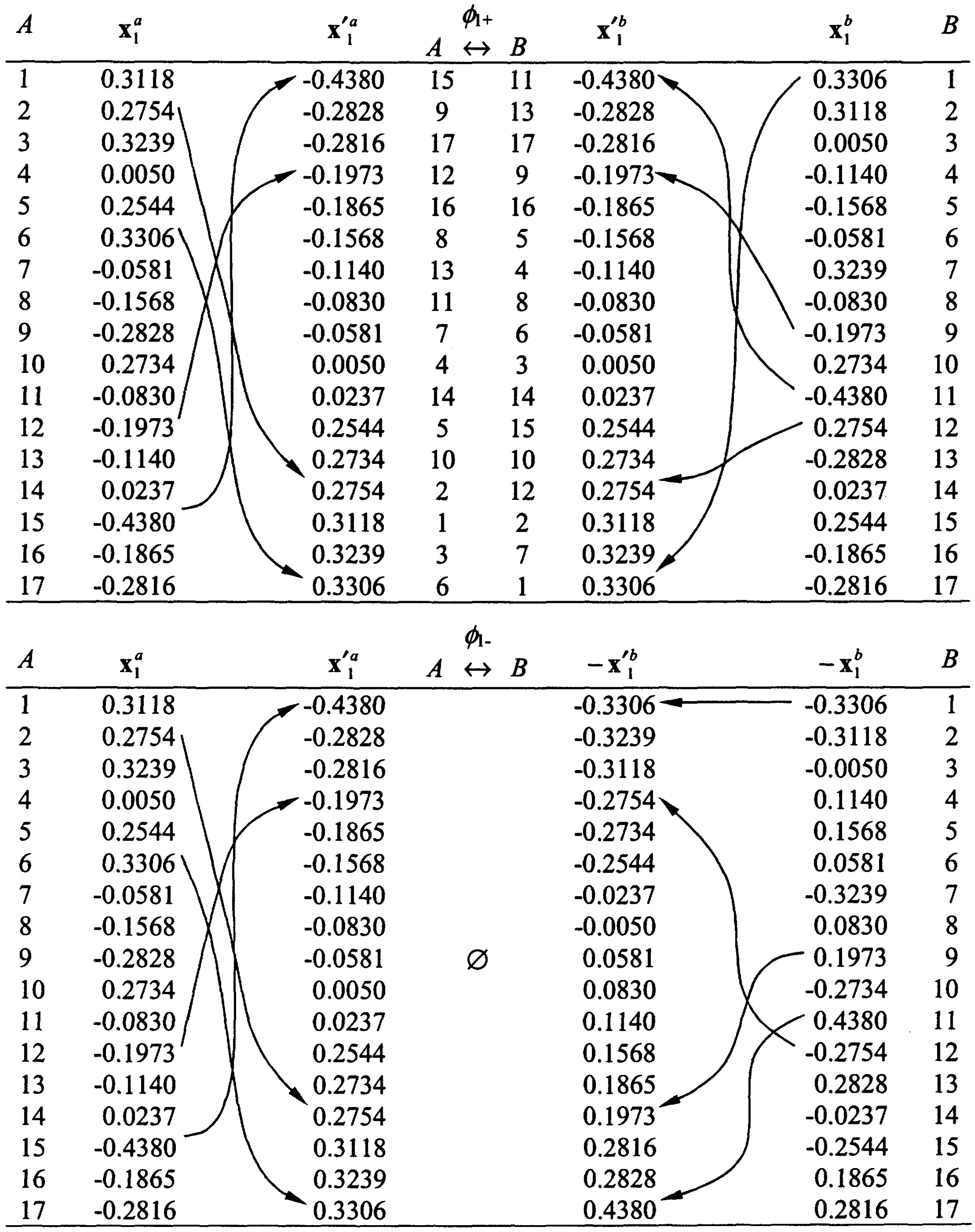
Figure 4.4 Two isomorphic graphs both with 17 vertices [Randic 1974].

Λ_a=Λ_b=diag(0.2724, 0.3167, -0.0735, -0.4794, -0.7891, 0.8935, -1.4324, -1.7687, -2.1954, -2.2543, -2.4962, -2.8303, 1.6312, 1.9591, 2.0818, 3.1644, 4.0000)

$\Lambda'_a = \Lambda'_b$=diag(-2.8303, -2.4962, -2.2543, -2.1954, -1.7687, -1.4324, -0.7891, -0.4794, -0.0735, 0.2724, 0.3167, 0.8935, 1.6312, 1.9591, 2.0818, 3.1644, 4.0000)

These two graphs have the same spectrum and all distinct eigenvalues. Algorithm I-2 is to be applied to determine whether there is a mapping ϕ between two eigenvector matrices. Figure 4.5 demonstrates the procedure of finding mapping Φ_1. The 15th component (–0.4380) in $\mathbf{x}_1^a$ is the smallest one and thus ordered as the first component in $\mathbf{x}'^a_1$; while the 11th component (–0.4380) in $\mathbf{x}_1^b$ is the smallest one and thus ordered as the first component in $\mathbf{x}'^b_1$. Since the first component in $\mathbf{x}'^a_1$ is equal to the first one in $\mathbf{x}'^b_1$, this means that there is a mapping between the 15th component in $\mathbf{x}_1^a$ and the 11th component in $\mathbf{x}_1^b$. Using the same procedure, the one-to-one mappings can be found for the other components of between $\mathbf{x}_1^a$ and $\mathbf{x}_1^b$. These result in a one-to-one mapping ϕ_{1+}. Note that it is easy to verify that ϕ_{1-} is empty. Hence, Φ_1 is the same as ϕ_{1+}, and is shown in Table 4.1.

Further, Figure 4.6 demonstrates the procedure of finding mapping Φ_2 in which mapping ϕ_{2-} has a one-to-one mapping but ϕ_{2+} is empty. It can be seen that mapping ϕ_{2-} in Figure 4.6 is the same as mapping Φ_1 in Table 4.1. Therefore, the intersection of them leads to Φ_2=Φ_1.



A	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1+} A	$\leftrightarrow$ B	$\mathbf{x}_1'^b$	$\mathbf{x}_1^b$	B
1	0.3118	-0.4380	15	11	-0.4380	0.3306	1
2	0.2754	-0.2828	9	13	-0.2828	0.3118	2
3	0.3239	-0.2816	17	17	-0.2816	0.0050	3
4	0.0050	-0.1973	12	9	-0.1973	-0.1140	4
5	0.2544	-0.1865	16	16	-0.1865	-0.1568	5
6	0.3306	-0.1568	8	5	-0.1568	-0.0581	6
7	-0.0581	-0.1140	13	4	-0.1140	0.3239	7
8	-0.1568	-0.0830	11	8	-0.0830	-0.0830	8
9	-0.2828	-0.0581	7	6	-0.0581	-0.1973	9
10	0.2734	0.0050	4	3	0.0050	0.2734	10
11	-0.0830	0.0237	14	14	0.0237	-0.4380	11
12	-0.1973	0.2544	5	15	0.2544	0.2754	12
13	-0.1140	0.2734	10	10	0.2734	-0.2828	13
14	0.0237	0.2754	2	12	0.2754	0.0237	14
15	-0.4380	0.3118	1	2	0.3118	0.2544	15
16	-0.1865	0.3239	3	7	0.3239	-0.1865	16
17	-0.2816	0.3306	6	1	0.3306	-0.2816	17

A	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1-} $A \leftrightarrow B$	$-\mathbf{x}_1'^b$	$-\mathbf{x}_1^b$	B
1	0.3118	-0.4380		-0.3306	-0.3306	1
2	0.2754	-0.2828		-0.3239	-0.3118	2
3	0.3239	-0.2816		-0.3118	-0.0050	3
4	0.0050	-0.1973		-0.2754	0.1140	4
5	0.2544	-0.1865		-0.2734	0.1568	5
6	0.3306	-0.1568		-0.2544	0.0581	6
7	-0.0581	-0.1140		-0.0237	-0.3239	7
8	-0.1568	-0.0830		-0.0050	0.0830	8
9	-0.2828	-0.0581	∅	0.0581	0.1973	9
10	0.2734	0.0050		0.0830	-0.2734	10
11	-0.0830	0.0237		0.1140	0.4380	11
12	-0.1973	0.2544		0.1568	-0.2754	12
13	-0.1140	0.2734		0.1865	0.2828	13
14	0.0237	0.2754		0.1973	-0.0237	14
15	-0.4380	0.3118		0.2816	-0.2544	15
16	-0.1865	0.3239		0.2828	0.1865	16
17	-0.2816	0.3306		0.4380	0.2816	17

Figure 4.5 The procedure of finding mapping Φ_1 for the graphs shown in Figure 4.4.

Table 4.1 Mapping Φ_1 for the graphs shown in Figure 4.4.

Graph	**Φ_1: vertex no. of graph (a) $\leftrightarrow$ vertex no. of graph (b)**																
Fig. 4.4a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Fig. 4.4b	2	12	7	3	15	1	6	5	13	10	8	9	4	14	11	16	17

The remaining eigenvector pairs $(\mathbf{x}_i^a, \mathbf{x}_i^b)$ and $(\mathbf{x}_i^a, -\mathbf{x}_i^b)$ for i=3, 4, ...17 follow the same procedure, and as a result, $\Phi_{17}=\Phi_1$. Finally, it can be concluded that these two graphs are isomorphic with the mapping as shown in Table 4.1. The matrix expression of Φ_{17} can be generated from Table 4.1, i.e.,

$$\Phi_{17} = \begin{bmatrix}
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

A	$\mathbf{x}_2^a$	$\mathbf{x}'^a_2$	Φ_{2+}		$\mathbf{x}'^b_2$	$\mathbf{x}_2^b$	B
			$A \leftrightarrow$	B			
1	0.0769	-0.4082			-0.4309	-0.1698	1
2	-0.3680	-0.3680			-0.3848	-0.0769	2
3	-0.0030	-0.2557			-0.3156	0.4082	3
4	-0.4082	-0.2517			-0.1808	-0.3156	4
5	-0.1870	-0.1870			-0.1698	-0.1284	5
6	0.1698	-0.1123			-0.1284	0.2557	6
7	-0.2557	-0.0478			-0.0769	0.0030	7
8	0.1284	-0.0448			0.0030	0.0478	8
9	-0.0448	-0.0086	∅		0.0086	-0.4309	9
10	0.3848	-0.0030			0.0448	-0.3848	10
11	-0.0478	0.0769			0.0478	0.2517	11
12	0.4309	0.1284			0.1123	0.3680	12
13	0.3156	0.1698			0.1870	0.0448	13
14	-0.1123	0.1808			0.2517	0.1123	14
15	-0.2517	0.3156			0.2557	0.1870	15
16	0.1808	0.3848			0.3680	-0.1808	16
17	-0.0086	0.4309			0.4082	0.0086	17

A	$\mathbf{x}_2^a$	$\mathbf{x}'^a_2$	Φ_{2-}		$-\mathbf{x}'^b_2$	$-\mathbf{x}_2^b$	B
			$A \leftrightarrow$	B			
1	0.0769	-0.4082	4	3	-0.4082	0.1698	1
2	-0.3680	-0.3680	2	12	-0.3680	0.0769	2
3	-0.0030	-0.2557	7	6	-0.2557	-0.4082	3
4	-0.4082	-0.2517	15	11	-0.2517	0.3156	4
5	-0.1870	-0.1870	5	15	-0.1870	0.1284	5
6	0.1698	-0.1123	14	14	-0.1123	-0.2557	6
7	-0.2557	-0.0478	11	8	-0.0478	-0.0030	7
8	0.1284	-0.0448	9	13	-0.0448	-0.0478	8
9	-0.0448	-0.0086	17	17	-0.0086	0.4309	9
10	0.3848	-0.0030	3	7	-0.0030	0.3848	10
11	-0.0478	0.0769	1	2	0.0769	-0.2517	11
12	0.4309	0.1284	8	5	0.1284	-0.3680	12
13	0.3156	0.1698	6	1	0.1698	-0.0448	13
14	-0.1123	0.1808	16	16	0.1808	-0.1123	14
15	-0.2517	0.3156	13	4	0.3156	-0.1870	15
16	0.1808	0.3848	10	10	0.3848	0.1808	16
17	-0.0086	0.4309	12	9	0.4309	-0.0086	17

Figure 4.6 The procedure of finding mapping Φ_2 for the graphs shown in Figure 4.4.

4.2.4 Example: Graphs with Group Mappings

Figure 4.7 gives two graphs both with 10 vertices. They have the same graph spectrum, i.e.,

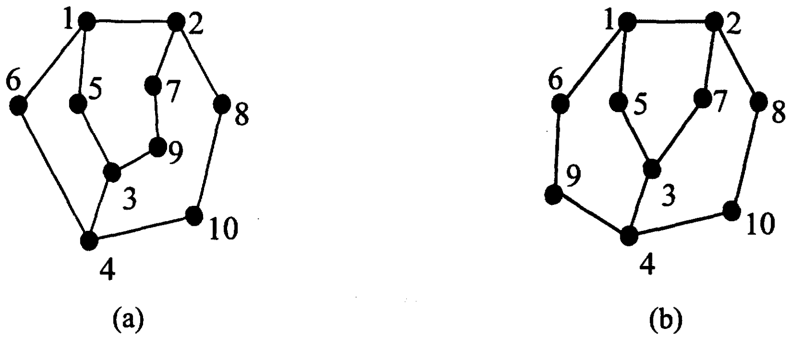
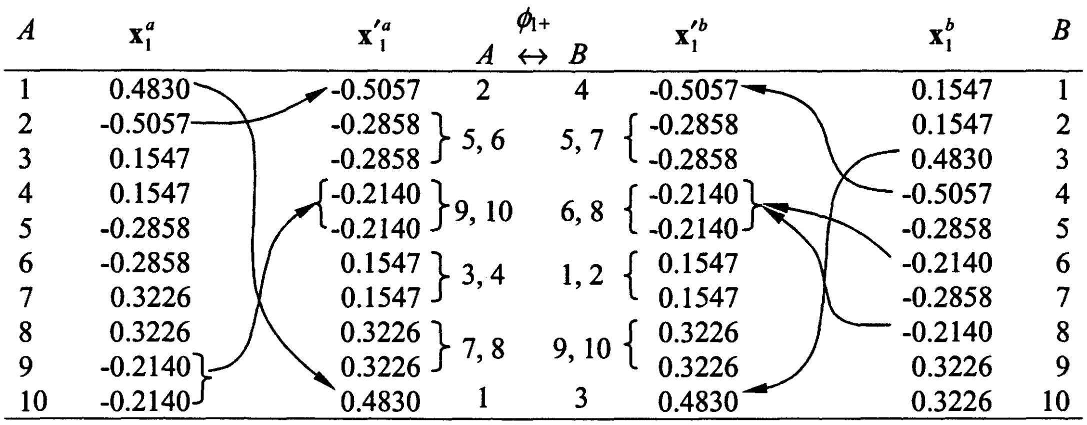


Figure 4.7 Two isomorphic graphs both with 10 vertices.

$\Lambda_a = \Lambda_b$ =diag(-2.2308, -2.0953, -1.2766, -0.7376, -0.6960, 0.4773, 1.2131, 1.3557, 1.5073, 2.4831)

Figure 4.8 illustrates a part of the procedure of finding permutation matrix P. It is seen from Figure 4.8 that the components in the eigenvector may be identical. The identical components are grouped. From Figure 4.8, after comparing the first eigenvector pair $(\mathbf{x}_1^a, \mathbf{x}_1^b)$, mapping Φ_1 (= ϕ_{1+}) includes *group-to-group mappings*, i.e., any element of a group in $\mathbf{x}_i^a$ can map to any element of the corresponding group in $\mathbf{x}_i^b$. For example, in mapping ϕ_{1+} shown in Figure 4.8, the group-to-group mappings exist between A(5, 6) and B(5, 7), A(9, 10) and B(6, 8), A(3, 4) and B(1, 2), and A(7, 8) and B(9, 10). After comparing the second eigenvector pair $(\mathbf{x}_2^a, \mathbf{x}_2^b)$, two mappings ϕ_{2+} and ϕ_{2-} are obtained.

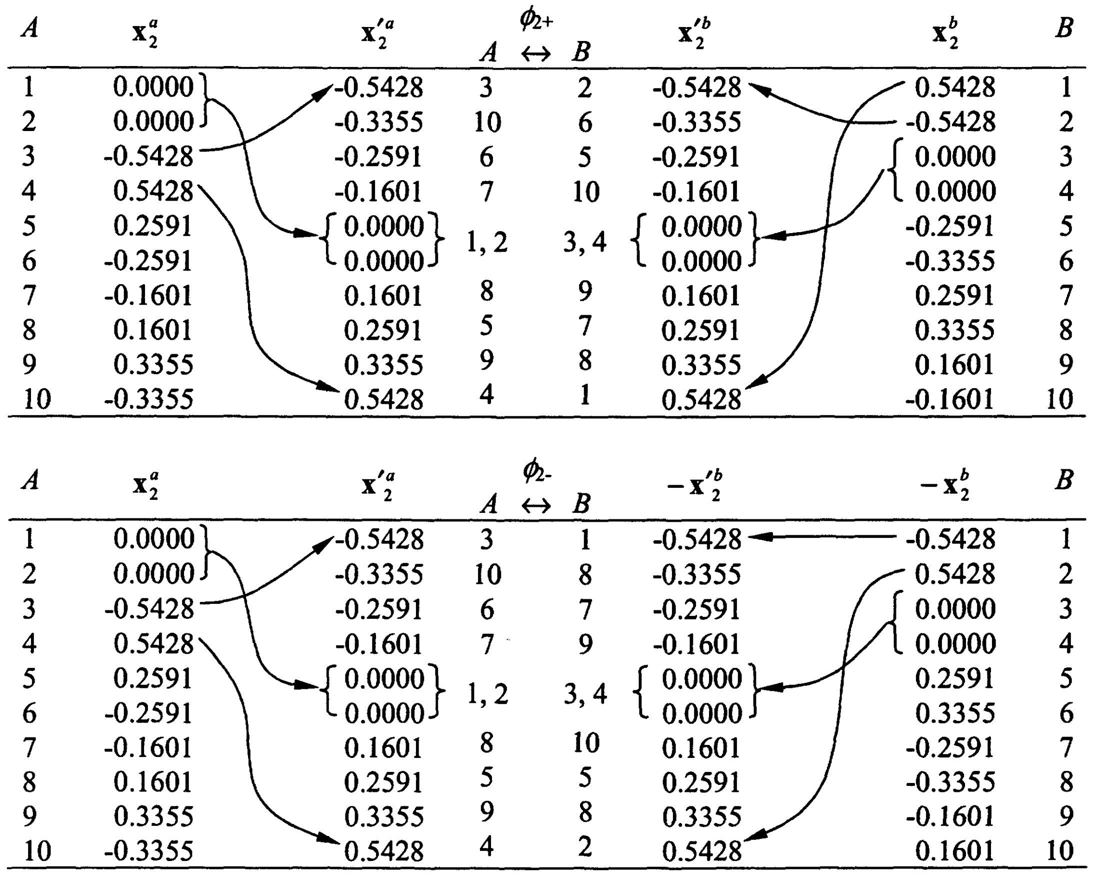
By intersecting mapping Φ_1 with mappings ϕ_{2+} and ϕ_{2-}, respectively, Φ_2 is found with two one-to-one mappings (see Table 4.2). Further comparisons conclude that the two mappings of Φ_2 are also valid to the remaining pairs of eigenvectors. Therefore, these two graphs are isomorphic.



A	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1+} *A*	$\leftrightarrow$	*B*	$\mathbf{x}_1'^b$	$\mathbf{x}_1^b$	*B*
1	0.4830	-0.5057	2		4	-0.5057	0.1547	1
2	-0.5057	-0.2858	5, 6		5, 7	-0.2858	0.1547	2
3	0.1547	-0.2858				-0.2858	0.4830	3
4	0.1547	-0.2140	9, 10		6, 8	-0.2140	-0.5057	4
5	-0.2858	-0.2140				-0.2140	-0.2858	5
6	-0.2858	0.1547	3, 4		1, 2	0.1547	-0.2140	6
7	0.3226	0.1547				0.1547	-0.2858	7
8	0.3226	0.3226	7, 8		9, 10	0.3226	-0.2140	8
9	-0.2140	0.3226				0.3226	0.3226	9
10	-0.2140	0.4830	1		3	0.4830	0.3226	10

A	$\mathbf{x}_1^a$	$\mathbf{x}_1'^a$	ϕ_{1-} *A* $\leftrightarrow$ *B*	$-\mathbf{x}_1'^b$	$-\mathbf{x}_1^b$	*B*
1	0.4830	-0.5057		-0.4830	-0.1547	1
2	-0.5057	-0.2858		-0.3226	-0.1547	2
3	0.1547	-0.2858		-0.3226	-0.4830	3
4	0.1547	-0.2140		-0.1547	0.5057	4
5	-0.2858	-0.2140	$\varnothing$	-0.1547	0.2858	5
6	-0.2858	0.1547		0.2140	0.2140	6
7	0.3226	0.1547		0.2140	0.2858	7
8	0.3226	0.3226		0.2858	0.2140	8
9	-0.2140	0.3226		0.2858	-0.3226	9
10	-0.2140	0.4830		0.5057	-0.3226	10

Figure 4.8 The procedure of finding mapping Φ_2 for the graphs shown in Figure 4.7.



A	$\mathbf{x}_2^a$	$\mathbf{x}_2'^a$	ϕ_{2+} A	$\leftrightarrow$ B	$\mathbf{x}_2'^b$	$\mathbf{x}_2^b$	B
1	0.0000	-0.5428	3	2	-0.5428	0.5428	1
2	0.0000	-0.3355	10	6	-0.3355	-0.5428	2
3	-0.5428	-0.2591	6	5	-0.2591	0.0000	3
4	0.5428	-0.1601	7	10	-0.1601	0.0000	4
5	0.2591	0.0000	1, 2	3, 4	0.0000	-0.2591	5
6	-0.2591	0.0000			0.0000	-0.3355	6
7	-0.1601	0.1601	8	9	0.1601	0.2591	7
8	0.1601	0.2591	5	7	0.2591	0.3355	8
9	0.3355	0.3355	9	8	0.3355	0.1601	9
10	-0.3355	0.5428	4	1	0.5428	-0.1601	10

A	$\mathbf{x}_2^a$	$\mathbf{x}_2'^a$	ϕ_{2-} A	$\leftrightarrow$ B	$-\mathbf{x}_2'^b$	$-\mathbf{x}_2^b$	B
1	0.0000	-0.5428	3	1	-0.5428	-0.5428	1
2	0.0000	-0.3355	10	8	-0.3355	0.5428	2
3	-0.5428	-0.2591	6	7	-0.2591	0.0000	3
4	0.5428	-0.1601	7	9	-0.1601	0.0000	4
5	0.2591	0.0000	1, 2	3, 4	0.0000	0.2591	5
6	-0.2591	0.0000			0.0000	0.3355	6
7	-0.1601	0.1601	8	10	0.1601	-0.2591	7
8	0.1601	0.2591	5	5	0.2591	-0.3355	8
9	0.3355	0.3355	9	8	0.3355	-0.1601	9
10	-0.3355	0.5428	4	2	0.5428	0.1601	10

Figure 4.8 (*Continued*)

Table 4.2 Mapping Φ_2 for the graphs shown in Figure 4.7.

Φ_2	Graph	Vertex No.									
$\Phi_1 \cap \phi_{2+}$	Fig. 4.7a	1	2	3	4	5	6	7	8	9	10
	Fig. 4.7b	3	4	2	1	7	5	10	9	8	6
$\Phi_1 \cap \phi_{2-}$	Fig. 4.7a	1	2	3	4	5	6	7	8	9	10
	Fig. 4.7b	3	4	1	2	5	7	9	10	6	8

4.3 Algorithm II – Part of Eigenvalues Are Distinct

4.3.1 Corollary 1

When an eigenvalue is not distinct from the others, the corresponding eigenvector is not unique. In this case, two isomorphic graphs may have different eigenvectors for coincident eigenvalues. Theorem 1 thus cannot be applied to identify the graphs having coincident eigenvalues. Nevertheless, the necessary condition of Theorem 1 (i.e., having the same set of eigenvalues and having the equivalent eigenvectors corresponding to distinct eigenvalues) still applies to the situation where there are coincident eigenvalues. This discussion results in the following corollary:

Corollary 1. *When two graphs G_a and G_b are isomorphic and they have part of their eigenvalues distinct, they are isomorphic if they also satisfy the following conditions: (1) the spectrums of G_a and G_b must be the same, and (2) there exists a one-to-one mapping ϕ such that $X'_a = \phi X'_b$ where X'_a and X'_b are the subsets of the eigenvectors of G_a and G_b, respectively, corresponding to the subsets of the eigenvalues which are distinct.*

Corollary 1 indicates a necessary condition for two isomorphic graphs which have coincident eigenvalues (partial). There is no guarantee that two graphs G_a and G_b, which have part of their eigenvalues distinct, are isomorphic if they meet the necessary condition. A further checking is necessary. This checking process is to examine whether ϕ (also P) satisfies Equation (4.1) or not. If such a case, these two graphs are isomorphic; otherwise, they are not isomorphic. Corollary 1 thus requires an algorithm to perform: (1) comparing eigenvalues, (2) finding mappings, and (3) checking the mappings. The first two steps involve algorithm I-1 and I-2, respectively. For the last step, if in mapping Φ_n, there is a one-to-one mapping ϕ, the checking procedure simply puts ϕ into Equation (4.1) to see whether ϕ meets the equation. However, if there is no such an one-to-one mapping existed in all mappings in Φ_n; in other words, all mappings in Φ_n are a group-to-group mapping, one has to find whether there is a one-to-one mapping common to all the group-to-group mappings. If a common one-to-one mapping does not exist, two graphs are not isomorphic. If a common one-to-one mapping exists, the checking procedure is triggered. This checking procedure is illustrated as follows.

Among all groups, the group having the least number of vertices is chosen as the first trial. If a pair of vertices in this group (e.g., vertex u_x of graph G_a and vertex v_y of graph G_b) has a one-to-one relation, elements a_{xx} and b_{yy} in A and B (respectively) are changed into $a_{xx}+x$ and $b_{yy}+x$, respectively. This leads to new matrices A' and B', respectively. Then a recursive process is applied by taking A' and B' as two new input matrices until a final result is obtained. An algorithm below fulfills this checking procedure.

II-3 ***checking isomorphism of two graphs (A, B, Φ)***

```
1.  if mapping $\Phi$ contains a one-to-one mapping, say $\phi$, checking if $A=\phi B\phi^T$. If $A=\phi B\phi^T$,
    then stop with 'isomorphic graphs'
2.  for each group-to-group mapping $\phi_i$ in $\Phi$, do
3.      find the mapping group $g$ in $\phi_i$ having the least number of vertices
4.      $u_x$ = randomly select one vertex from $g$ on $A$ side
5.      for each vertex $v_y$ in group $g$ on $B$ side, do
6.          let $u_x \leftrightarrow v_y$ and modify both the elements of $A$ at $a_{xx}$ and B at $b_{yy}$ with $a_{xx}+x$
            and $b_{yy}+x$ into $A'$ and $B'$, respectively
```

7. ***comparison of the spectrums of two graphs*** (A', B')

8. Φ = ***finding permutation matrix of two graphs*** (A', B')

9. if part of eigenvalues is coincident then ***checking permutation matrix of two graphs*** (A', B', Φ)

Algorithm II-3 is further illustrated by the following examples.

4.3.2 Example: One-to-One Mapping

Both two graphs shown in Figure 2.5 have 8 vertices. They have the same graph spectrum, i.e.,

$$\Lambda_a=\Lambda_b=\text{diag}(-2.4142, -1.7321, -1.0000, -1.0000, 0.4142, 1.0000, 1.7321, 3.0000)$$

and the corresponding eigenvector matrices are, respectively,

$$X_a = \begin{pmatrix} 0.0000 & -0.6280 & 0.2604 & 0.2392 & 0.0000 & 0.5000 & 0.3251 & 0.3536 \\ 0.3536 & 0.2299 & 0.0779 & -0.6074 & 0.3536 & 0.0000 & 0.4440 & 0.3536 \\ -0.5000 & 0.0000 & 0.2604 & 0.2392 & 0.5000 & -0.5000 & 0.0000 & 0.3536 \\ 0.3536 & -0.2299 & -0.5986 & 0.1290 & 0.3536 & 0.0000 & -0.4440 & 0.3536 \\ 0.0000 & 0.6280 & 0.2604 & 0.2392 & 0.0000 & 0.5000 & -0.3251 & 0.3536 \\ -0.3536 & -0.2299 & 0.0779 & -0.6074 & -0.3536 & 0.0000 & -0.4440 & 0.3536 \\ 0.5000 & 0.0000 & 0.2604 & 0.2392 & -0.5000 & -0.5000 & 0.0000 & 0.3536 \\ -0.3536 & 0.2299 & -0.5986 & 0.1290 & -0.3536 & 0.0000 & 0.4440 & 0.3536 \end{pmatrix}$$

$$X_b = \begin{pmatrix} 0.5000 & 0.0000 & 0.3416 & -0.0913 & -0.5000 & -0.5000 & 0.0000 & -0.3536 \\ -0.3536 & -0.2299 & -0.4707 & -0.3917 & -0.3536 & 0.0000 & 0.4440 & -0.3536 \\ 0.3536 & 0.2299 & -0.4707 & -0.3917 & 0.3536 & 0.0000 & -0.4440 & -0.3536 \\ 0.0000 & 0.6280 & 0.3416 & -0.0913 & -0.0000 & 0.5000 & 0.3251 & -0.3536 \\ -0.5000 & 0.0000 & 0.3416 & -0.0913 & 0.5000 & -0.5000 & 0.0000 & -0.3536 \\ 0.3536 & -0.2299 & -0.2124 & 0.5744 & 0.3536 & 0.0000 & 0.4440 & -0.3536 \\ 0.0000 & -0.6280 & 0.3416 & -0.0913 & 0.0000 & 0.5000 & -0.3251 & -0.3536 \\ -0.3536 & 0.2299 & -0.2124 & 0.5744 & -0.3536 & 0.0000 & -0.4440 & -0.3536 \end{pmatrix}$$

Theorem 1 is not suitable for this case because of partially coincident eigenvalues ($\lambda_3 = \lambda_4 = -1.0000$). However, according to Corollary 1, the eigenvectors corresponding to the distinct eigenvalues can still be used for finding possible isomorphic mappings. By algorithm I-2, part of the results for the comparison of unique eigenvectors between X_a and X_b can be obtained as shown in Table 4.3.

Table 4.3 The mapping by comparing unique eigenvectors between X_a and X_b.

ϕ_1+		ϕ_2+		ϕ_5+		ϕ_6+		ϕ_7-		ϕ_8-	
$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$	
3	5	1	7	7	1	3	1	4	2	1	1
6	2	4	2	6	2	7	5	6	6	2	2
8	8	6	6	8	8	2	2	5	4	3	3
1	4	3	1	1	4	4	3	3	1	4	4
5	7	7	5	5	7	6	6	7	5	5	5
2	3	2	3	2	3	8	8	1	7	6	6
4	6	8	8	4	6	1	4	2	3	7	7
7	1	5	4	3	5	5	7	8	8	8	8

Based on Table 4.3, a one-to-one mapping is created when mapping ϕ_{1+} intersects with mapping ϕ_{2+}, which is listed in Table 4.4.

Table 4.4 The one-to-one mapping created with the intersection between ϕ_{1+} and ϕ_{2+}.

Graph				$\phi_{1+} \cap \phi_{2+}$				
A	1	2	3	4	5	6	7	8
B	7	3	5	6	4	2	1	8

Further study indicates that this one-to-one mapping is also common to mappings ϕ_{5+}, ϕ_{6+}, ϕ_{7-}, and ϕ_{8-}. Therefore, according to Corollary 1, the checking procedure is needed for this one-to-one mapping to determine whether it is an isomorphic mapping between two graphs. It is easy to transform the mapping shown in Table 4.4 into a row permutation, say ϕ:

$$\phi = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

One can verify that ϕ satisfies Equation (4.1) in replacement of P. Therefore, these two graphs are isomorphic.

4.3.3 Examples: Group-to-Group Mapping

Figure 4.9 shows two graphs, both with 7 vertices. They have the same graph spectrum, i.e.,

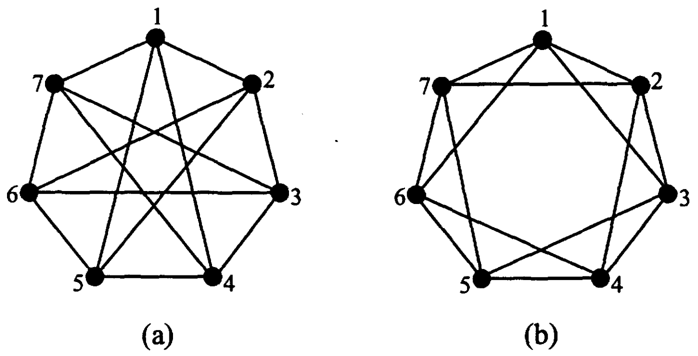


Figure 4.9 Two isomorphic graphs with partially coincident eigenvalues.

$$\Lambda_a=\Lambda_b=\text{diag}(-2.2470, -2.2470, -0.5550, -0.5550, 0.8019, 0.8019, 4.0000)$$

and the corresponding eigenvector matrices are, respectively,

$$X_a = \begin{pmatrix} -0.4097 & 0.3433 & 0.3329 & -0.4182 & -0.0703 & 0.5299 & -0.3780 \\ 0.2202 & -0.4871 & -0.1194 & -0.5210 & -0.5010 & -0.1864 & -0.3780 \\ 0.0130 & 0.5344 & -0.4818 & -0.2315 & 0.2932 & -0.4469 & -0.3780 \\ -0.2435 & -0.4758 & -0.4814 & 0.2323 & 0.3705 & 0.3853 & -0.3780 \\ 0.4259 & 0.3230 & -0.1185 & 0.5212 & -0.4581 & 0.2754 & -0.3780 \\ -0.5239 & -0.1063 & 0.3336 & 0.4176 & -0.1666 & -0.5079 & -0.3780 \\ 0.5181 & -0.1316 & 0.5345 & -0.0005 & 0.5322 & -0.0494 & -0.3780 \end{pmatrix}$$

$$X_b = \begin{pmatrix} 0.1652 & 0.5084 & 0.4920 & 0.2088 & -0.1979 & -0.4965 & -0.3780 \\ 0.4588 & -0.2742 & -0.3527 & -0.4016 & 0.2648 & -0.4643 & -0.3780 \\ -0.3694 & -0.3863 & 0.1435 & 0.5149 & 0.5281 & -0.0824 & -0.3780 \\ -0.2944 & 0.4461 & 0.0941 & -0.5262 & 0.3937 & 0.3615 & -0.3780 \\ 0.5004 & 0.1878 & -0.3131 & 0.4332 & -0.0372 & 0.5332 & -0.3780 \\ 0.0717 & -0.5297 & 0.4700 & -0.2545 & -0.4401 & 0.3034 & -0.3780 \\ -0.5324 & 0.0479 & -0.5339 & 0.0254 & -0.5116 & -0.1549 & -0.3780 \end{pmatrix}$$

Only one eigenvalue (λ_7=4.0000) is distinct from the others. Therefore, only the eigenvector corresponding to λ_7 can be used to find possible one-to-one mappings according to Corollary 1. Unfortunately, the comparison of this eigenvector between X_a and X_b gives a group-to-group mapping (see the last column of X_a and X_b), i.e., any vertex in A might map to any vertex in B. Algorithm II-3 is then applied to this situation. Suppose that vertex 1 in A has a one-to-one relation with vertex 1 in B if they are isomorphic. Change the entries a_{11} and b_{11} from 0 into 1 (a_{11}+1). This leads to 'new' matrices A' and B'. A' and B' are then considered as two 'new' matrices to algorithms I-1, I-2, and II-3, respectively. The eigenvalues and the corresponding eigenvectors of A' and B' are, respectively,

$$\Lambda'_a = \Lambda'_b = \text{diag}(-2.2470, -2.0302, -0.5550, -0.3052, 0.8019, 1.1592, 4.1763)$$

$$X'_a = \begin{pmatrix} 0.0000 & -0.3999 & 0.0000 & 0.4518 & 0.0000 & 0.6485 & -0.4641 \\ 0.2319 & 0.5237 & -0.4179 & 0.2734 & -0.5211 & -0.1296 & -0.3663 \\ -0.4179 & -0.3728 & -0.5211 & 0.0165 & 0.2319 & -0.4900 & -0.3474 \\ 0.5211 & 0.0822 & -0.2319 & -0.5683 & 0.4179 & 0.1812 & -0.3708 \\ -0.5211 & 0.0822 & 0.2319 & -0.5683 & -0.4179 & 0.1812 & -0.3708 \\ 0.4179 & -0.3728 & 0.5211 & 0.0165 & -0.2319 & -0.4900 & -0.3474 \\ -0.2319 & 0.5237 & 0.4179 & 0.2734 & 0.5211 & -0.1296 & -0.3663 \end{pmatrix}.$$

$$X_b' = \begin{pmatrix} 0.0000 & 0.3999 & 0.0000 & -0.4518 & 0.0000 & -0.6485 & -0.4641 \\ 0.5211 & -0.0822 & -0.2319 & 0.5683 & -0.4179 & -0.1812 & -0.3708 \\ -0.2319 & -0.5237 & 0.4179 & -0.2734 & -0.5211 & 0.1296 & -0.3663 \\ -0.4179 & 0.3728 & -0.5211 & -0.0165 & -0.2319 & 0.4900 & -0.3474 \\ 0.4179 & 0.3728 & 0.5211 & -0.0165 & 0.2319 & 0.4900 & -0.3474 \\ 0.2319 & -0.5237 & -0.4179 & -0.2734 & 0.5211 & 0.1296 & -0.3663 \\ -0.5211 & -0.0822 & 0.2319 & 0.5683 & 0.4179 & -0.1812 & -0.3708 \end{pmatrix}$$

They have the same eigenvalues, and they are all distinct. By algorithm I-2, part of the results of the comparison of unique eigenvectors between X_a' and X_b' can be obtained as shown in Table 4.5. A common one-to-one mapping is found when investigating all the mappings listed in Table 4.5. This common one-to-one mapping is shown in Table 4.6. A matrix can be created based on Table 4.6 and is shown below:

$$\phi = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Table 4.5 The mapping by comparing unique eigenvectors between X_a and X_b.

ϕ_{1+}		ϕ_{2-}		ϕ_{3+}		ϕ_{4-}		ϕ_{5-}		ϕ_{6-}		ϕ_{7+}	
$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$		$A \leftrightarrow B$	
5	7	1	1	3	4	4	2	2	6	3	4	1	1
3	4	3	4	2	6	5	7	5	7	6	5	4	2
7	3	6	5	4	2	3	4	6	5	2	3	5	7
1	1	4	2	1	1	6	5	1	1	7	6	2	3
2	6	5	7	5	7	2	3	3	4	4	2	7	6
6	5	2	3	7	3	7	6	4	2	5	7	3	4
4	2	7	6	6	5	1	1	7	3	1	1	6	5

Table 4.6 The one-to-one mapping created by investigating all the mappings.

Graph	$\phi_{1+} \cap \phi_{2-} \cap \phi_{3+} \cap \phi_{4-} \cap \phi_{5-} \cap \phi_{6-} \cap \phi_{7+}$						
A	1	2	3	4	5	6	7
B	1	6	4	2	7	5	3

Equation (4.1) is satisfied when ϕ replaces *P*. Hence, these two graphs are isomorphic.

Figure 4.10 illustrates another example, where two graphs have, respectively, 15 vertices. This pair of graphs has the same eigenvalues, i.e.,

$\Lambda_a=\Lambda_b=$diag(-3.1642, -3.1642, -1.6180, -1.6180, -0.7616, -0.4142, -0.2271, -0.2271, 0.6180, 0.6180, 0.6367, 1.3914, 1.3914, 2.4142, 4.1249)

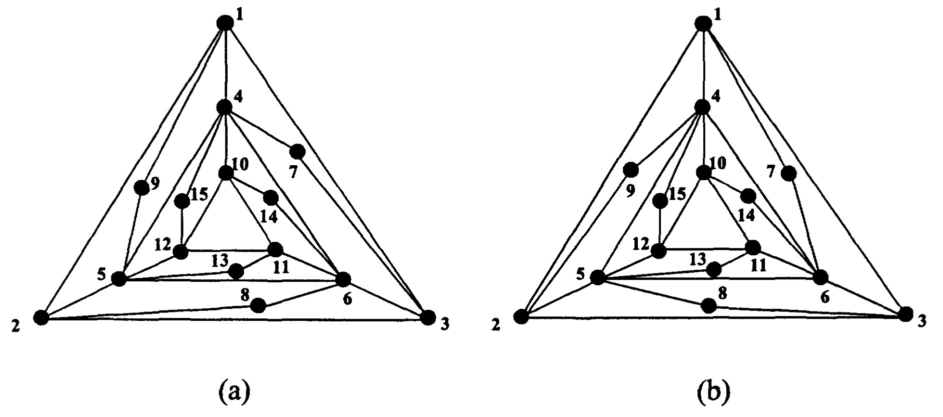


Figure 4.10 Two non-isomorphic graphs with partially coincident eigenvalues [Yan and Hall 1982].

There are 5 distinct eigenvalues, i.e., λ_5=-0.7616, λ_6=-0.4142, λ_{11}=0.6367, λ_{14}=2.4142, and λ_{15}=4.1249. The eigenvectors corresponding to these distinct eigenvalues are listed as follows:

$$X_a=\begin{pmatrix}
 & 0.0501 & 0.1562 & & 0.3131 & & 0.3772 & 0.2572 \\
 & 0.0501 & 0.1562 & & 0.3131 & & 0.3772 & 0.2572 \\
 & 0.0501 & 0.1562 & & 0.3131 & & 0.3772 & 0.2572 \\
 & 0.2320 & 0.0000 & & -0.3573 & & 0.0000 & 0.3897 \\
 & 0.2320 & 0.0000 & & -0.3573 & & 0.0000 & 0.3897 \\
 & 0.2320 & 0.0000 & & -0.3573 & & 0.0000 & 0.3897 \\
 & -0.3705 & -0.3772 & & -0.0695 & & 0.1562 & 0.1568 \\
\cdots\cdots & -0.3705 & -0.3772 & \cdots\cdots & -0.0695 & \cdots\cdots & 0.1562 & 0.1568 \\
 & -0.3705 & -0.3772 & & -0.0695 & & 0.1562 & 0.1568 \\
 & 0.0501 & -0.1562 & & 0.3131 & & -0.3772 & 0.2572 \\
 & 0.0501 & -0.1562 & & 0.3131 & & -0.3772 & 0.2572 \\
 & 0.0501 & -0.1562 & & 0.3131 & & -0.3772 & 0.2572 \\
 & -0.3705 & 0.3772 & & -0.0695 & & -0.1562 & 0.1568 \\
 & -0.3705 & 0.3772 & & -0.0695 & & -0.1562 & 0.1568 \\
 & -0.3705 & 0.3772 & & -0.0695 & & -0.1562 & 0.1568
\end{pmatrix}$$

$$X_b = \begin{pmatrix} & -0.0501 & -0.1562 & & -0.3131 & & 0.3772 & -0.2572 \\ & -0.0501 & -0.1562 & & -0.3131 & & 0.3772 & -0.2572 \\ & -0.0501 & -0.1562 & & -0.3131 & & 0.3772 & -0.2572 \\ & -0.2320 & 0.0000 & & 0.3573 & & 0.0000 & -0.3897 \\ & -0.2320 & 0.0000 & & 0.3573 & & 0.0000 & -0.3897 \\ & -0.2320 & 0.0000 & & 0.3573 & & 0.0000 & -0.3897 \\ & 0.3705 & 0.3772 & & 0.0695 & & 0.1562 & -0.1568 \\ \cdots\cdots & 0.3705 & 0.3772 & \cdots\cdots & 0.0695 & \cdots\cdots & 0.1562 & -0.1568 \\ & 0.3705 & 0.3772 & & 0.0695 & & 0.1562 & -0.1568 \\ & -0.0501 & 0.1562 & & -0.3131 & & -0.3772 & -0.2572 \\ & -0.0501 & 0.1562 & & -0.3131 & & -0.3772 & -0.2572 \\ & -0.0501 & 0.1562 & & -0.3131 & & -0.3772 & -0.2572 \\ & 0.3705 & -0.3772 & & 0.0695 & & -0.1562 & -0.1568 \\ & 0.3705 & -0.3772 & & 0.0695 & & -0.1562 & -0.1568 \\ & 0.3705 & -0.3772 & & 0.0695 & & -0.1562 & -0.1568 \end{pmatrix}$$

Through algorithm I-2, two group-to-group mappings are found and listed in Table 4.7.

Table 4.7 The group-to-group mappings by comparing all the unique eigenvectors.

Φ_{15}	Graph	Group 1	Group 2	Group 3	Group 4	Group 5
Φ'_{15}	*A*	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15
	B	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15
Φ''_{15}	*A*	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15
	B	10, 11, 12	4, 5, 6	13, 14, 15	1, 2, 3	7, 8, 9

Algorithm II-3 is applied here. As each group has the same number of vertices, the first group (Group 1) is selected as a start to find a possible one-to-one mapping from group-to-group mappings Φ_{15}. Consider mapping Φ'_{15}. If it is an isomorphic mapping, each pair of groups between A and B must match. This means that the members in a group in A must have a one-to-one relation with the members in the corresponding group in B. For example, in Group 1, there must be one of the six one-to-one mappings, i.e., $A(1, 2, 3) \leftrightarrow B(1, 2, 3)$, $A(1, 2, 3) \leftrightarrow B(1, 3, 2)$, $A(1, 2, 3) \leftrightarrow B(2, 1, 3)$, $A(1, 2, 3) \leftrightarrow B(2, 3, 1)$, $A(1, 2, 3) \leftrightarrow B(3, 1, 2)$, or $A(1, 2, 3) \leftrightarrow B(3, 2, 1)$. Take member 1 in Group 1 in A. It must match one of three members in Group 1 in B, i.e., $A(1) \leftrightarrow B(1)$, $A(1) \leftrightarrow B(2)$, or $A(1) \leftrightarrow B(3)$. Suppose that there is a mapping $A(1) \leftrightarrow B(1)$. By changing the entries a_{11} and b_{11} with $a_{11}+1$ and $b_{11}+1$, respectively, two new adjacency matrices A' and B' are obtained. These two matrices have the same eigenvalues, i.e.,

$$\Lambda'_a = \Lambda'_b = \text{diag}(-3.1642,\ -3.0734,\ -1.6180,\ -1.4249,\ -0.7595,\ -0.4014,\ -0.2271,\ -0.0888,\ 0.6180,\ 0.6267,\ 0.8338,\ 1.3914,\ 1.4722,\ 2.6022,\ 4.2132)$$

It can be seen that all eigenvalues are distinct. Through algorithm I-2, their eigenvectors are compared to find whether there exists a common mapping. The eigenvectors of A' and B' corresponding to the eigenvalue $\lambda'_1 = -3.1642$ are listed as follows, respectively,

$$\mathbf{x}'^1_a = (\ 0.0000\quad 0.3010\quad -0.3010\quad -0.1242\quad -0.3929\quad 0.5170\quad 0.1344\quad -0.2585\quad 0.1242\quad 0.1556\quad -0.3469\quad 0.1913\quad 0.2338\quad -0.2126\quad -0.0212)^T$$

$\mathbf{x}_b'^1$=(0.0000 0.3010 -0.3010 0.1242 -0.5170 0.3929 -0.1242 0.2585 -0.1344 0.0000 -0.3010 0.3010 0.2585 -0.1242 -0.1344$)^T$

There is not any mapping between $\mathbf{x}_a'^1$ and $\mathbf{x}_b'^1$ because of different eigenvector components. This means that the hypothesis $A(1) \leftrightarrow B(1)$ is not true for isomorphism. Similarly, mappings $A(1) \leftrightarrow B(2)$ and $A(1) \leftrightarrow B(3)$ are also found not true for isomorphism. Hence, mapping Φ_{15}' between A and B is not true for isomorphism. The same procedure used for Φ_{15}' can be applied to Φ_{15}''. One can lead to the conclusion that mapping Φ_{15}'' is not true for isomorphism as well. Therefore, the two graphs shown in Figure 4.10 are not isomorphic.

4.4 Adjusted Adjacency Matrix

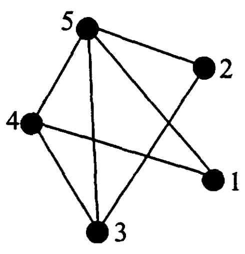
As discussed above, once two graphs are represented by two adjacency matrices, their eigenvalues and eigenvectors corresponding to distinct eigenvalues can be used for determining whether these two graphs are isomorphic. Therefore, it requires that there exist at least one distinct eigenvalue (this means that the corresponding eigenvector is unique) in the AM of a graph. However, this condition is not always satisfied with AM as the representation of a graph. The solution to this problem is to find a new matrix representation for graph; this new matrix has a one-to-one relation with a graph, and hopefully the new matrix is a symmetric one. The new matrix is called the adjusted adjacency matrix (AAM), which is discussed below.

4.4.1 Definition of Adjusted Adjacency Matrix

The adjusted adjacency matrix of a graph with n vertices is defined as

$$a_{ij} = \begin{cases} \text{same as the adjacency matrix} & ; i \neq j \\ n - \text{degree}(v_i) \neq 0 & ; i = j \end{cases}$$

In fact, the sum of entries with nonzero value at each row of the AM is equal to the degree of the corresponding vertex, i.e., degree(v_i) for vertex v_i. Figure 4.11 shows a 5-vertex graph as well as its AM and AAM. In this graph, vertices 1 and 2 have two degrees, vertices 3 and 4 have three degrees, and vertex 5 has four degrees.



graph

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

AM

$$\begin{bmatrix} 3 & 0 & 0 & 1 & 1 \\ 0 & 3 & 1 & 0 & 1 \\ 0 & 1 & 2 & 1 & 1 \\ 1 & 0 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

AAM

Figure 4.11 A 5-vertex graph, the AM and the AAM.

According to this definition, the AAM of a graph is unique and sufficient when representing the graph. Hence, any two graphs are isomorphic if and only if their AAMs are equivalent. With the AAM of a graph, there is the following theorem:

Theorem 2. *The adjusted adjacency matrix of a graph with n vertices has a unique eigenvalue n.*

Proof. Consider the matrix

$$B = \frac{1}{n} M$$

where M is the AAM of a graph. To prove that n is a unique eigenvalue of matrix M, one needs to show that matrix B has a unique eigenvalue 1.

Matrix B has two properties: (1) every element of the matrix is no less than 0 ($b_{ij} \geq 0$), and (2) the sum of every row/column of the matrix is 1 ($\sum_{i=1}^{n} b_{ij} = 1$; $j = 1,2,\cdots,n$). Matrix B with the above properties is called the *doubly stochastic matrix*, which has a largest eigenvalue 1 and its corresponding eigenvector $[1 \quad 1 \quad \cdots \quad 1]^T$ [Liu and Lai 2000]. Because the graph associated with matrix M and matrix B is strongly connected (for each entry (i, j) in matrix M, there exists an integer k such that the associated graph is connected by k edges between vertex i and vertex j), matrix B is an *irreducible matrix*. More precisely, matrix B is an irreducible nonnegative matrix due to Property (1). Furthermore, it is known that an irreducible nonnegative matrix with a nonzero principal diagonal is *primitive* [Minc 1988]. It is known that an $n \times n$ nonnegative primitive matrix, matrix B in this case, has a unique eigenvalue, i.e., 1 which is the largest among all the eigenvalues [Seneta 1981]. Matrix M thus has a unique eigenvalue n, and its corresponding eigenvector $[1 \quad 1 \quad \cdots \quad 1]^T$. ∎

With AAM, instead of AM, both Theorem 1 and Corollary 1 together with algorithms I-1, I-2, and II-3 are still applicable for GI detection without any change. This is because they are built upon the symmetry property of matrices for graphs.

4.4.2 Adjusted Adjacency Matrix versus Adjacency Matrix

AAM ensures that a graph has at least one distinct eigenvalue and thus the Eigensystem approach can be applied for solving isomorphism problems of general graphs. Besides, there is a surprising phenomenon from many tests: most pairs of non-isomorphic cospectral graphs on their AMs have different graph spectrums on their AAMs. This means that graph isomorphic identification should be conducted more effectively with AAM than with AM. The following are several examples to demonstrate this point.

Figure 2.15 illustrated a pair of trees which have the same graph spectrum on AM but are non-isomorphic. The AAMs of this pair of trees are, respectively,

$$\Lambda'_a = \text{diag}(1.9452, 4.8390, 6.4932, 7.0000, 7.0000, 7.0000, 7.7226, 8.0000)$$

$$\Lambda'_b = \text{diag}(2.3542, 4.0000, 7.0000, 7.0000, 7.0000, 7.0000, 7.6458, 8.0000)$$

It can be seen that the two trees are not isomorphic because they have different graph spectrums on AAM.

Figure 4.12 shows another example of two cospectral graphs on AM. The AMs of these two graphs have the same eigenvalues

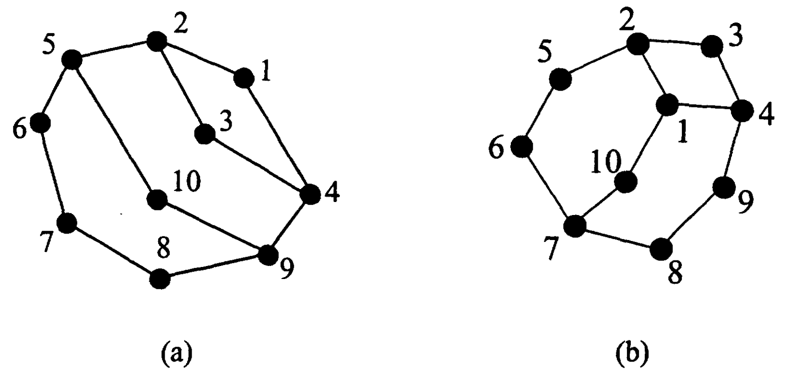


Figure 4.12 Two non-isomorphic cospectral graphs on AM.

Λ_a=Λ_b=diag(-2.4998, -1.6893, -1.4142, -0.9472, 0.0000, 0.0000, 0.9472, 1.4142, 1.6893, 2.4998)

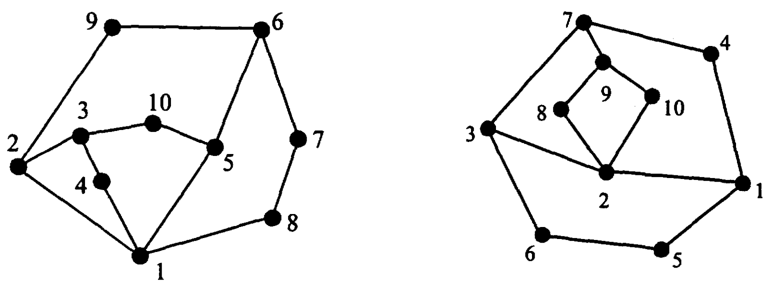
The eigenvalues of these two graphs based on AAMs are, respectively,

Λ'_a =diag(4.8441, 5.7530, 6.0000, 6.8948, 7.4450, 8.0000, 8.7681, 8.8019, 9.4930, 10.0000)

Λ'_b =diag(4.8162, 5.7119, 6.1981, 6.7568, 7.5550, 7.7564, 8.7501, 9.2086, 9.2470, 10.0000)

Since Λ'_a and Λ'_b are different, these two graphs are not cospectral on AAM. One can immediately conclude that the two graphs are not isomorphic.

Figure 4.13 shows another pair of graphs, both with 10 vertices. The two graphs have the same graph spectrum on AM:



(a) (b)

Figure 4.13 Two non-isomorphic cospectral graphs on AM.

$\Lambda_a = \Lambda_b =$diag(-2.4289, -2.0693, -1.5279, -0.9182, 0.0000, 0.4528, 1.0000, 1.1354, 1.6037, 2.7523)

The eigenvalues of these two graphs based on AAMs are, respectively,

Λ'_a =diag(4.3617, 5.1341, 5.9379, 6.7547, 7.5416, 7.6967, 8.5509, 8.7930, 9.2295, 10.0000)

Λ'_b =diag(4.3990, 5.0782, 5.8922, 6.8406, 7.4351, 8.0000, 8.2729, 8.7685, 9.3134, 10.0000)

These two graphs are not isomorphic since Λ'_a and Λ'_b are different.

More examples can be seen in Figure 4.14 where each pair of graphs (trees) has the same graph spectrum on AM but different on AAM as shown in the following. Hence, the graphs (trees) in each pair are not isomorphic.

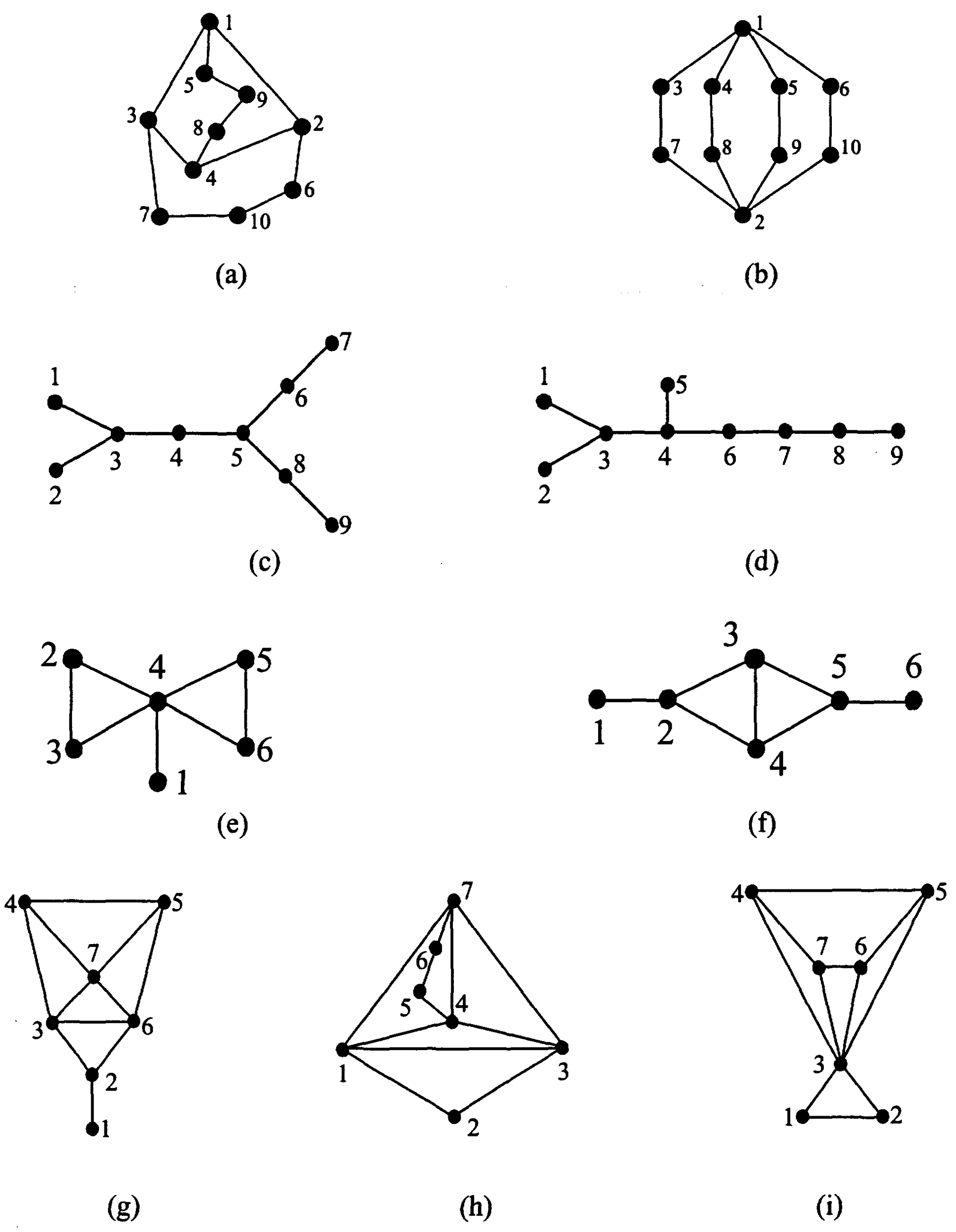
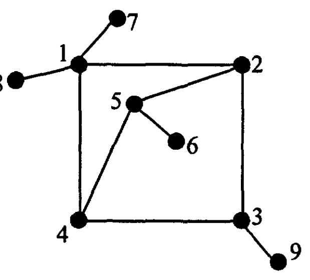
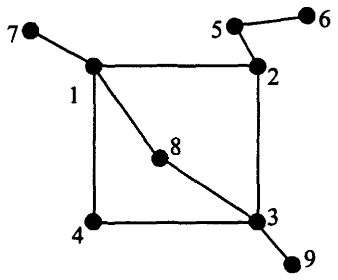


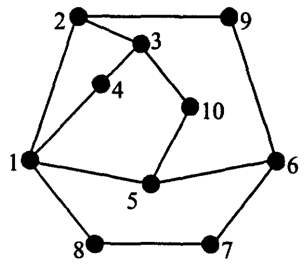
Figure 4.14 Six pairs of cospectral graphs (trees) on AM [Harary *et al.* 1971].



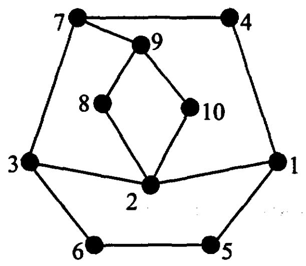
(j)



(k)



(l)



(m)

Figure 4.14 (*Continued*)

$\Lambda_a=\Lambda_b=\mathrm{diag}(-2.5616, -1.5616, -1.0000, -1.0000, -1.0000, 1.0000, 1.0000, 1.0000, 1.5616, 2.5616)$

$$\begin{cases} \Lambda'_a=\mathrm{diag}(4.6385,\ 6.3820,\ 6.3820,\ 6.3820,\ 6.8326,\ 8.6180,\ 8.6180,\ 8.6180,\ 9.5289,\ 10.0000) \\ \Lambda'_b=\mathrm{diag}(4.4384,\ 5.0000,\ 7.0000,\ 7.0000,\ 7.0000,\ 8.5616,\ 9.0000,\ 9.0000,\ 9.0000,\ 10.0000) \end{cases}$$

$\Lambda_c=\Lambda_d=\mathrm{diag}(-2.0840, -1.5718, -1.0000, -0.4317, 0.0000, 0.4317, 1.0000, 1.5718, 2.0840)$

$$\begin{cases} \Lambda'_c=\mathrm{diag}(4.4574, 5.0936, 6.3820, 6.8551, 7.7892, 8.0000, 8.6180, 8.8047, 9.0000) \\ \Lambda'_d=\mathrm{diag}(4.3563, 5.4277, 6.0000, 6.9108, 8.0000, 8.0000, 8.4679, 8.8373, 9.0000) \end{cases}$$

$\Lambda_e=\Lambda_f=\text{diag}(-1.9032, -1.0000, -1.0000, 0.1939, 1.0000, 2.7093)$

$$\begin{cases} \Lambda'_e=\text{diag}(0.0000, 3.0000, 3.0000, 5.0000, 5.0000, 6.0000) \\ \Lambda'_f=\text{diag}(1.2679, 2.0000, 2.5858, 4.7321, 5.4142, 6.0000) \end{cases}$$

$\Lambda_g=\Lambda_h=\Lambda_i=\text{diag}(-2.0000, -1.7785, -1.0000, 0.0000, 0.0000, 1.2892, 3.4893)$

$$\begin{cases} \Lambda'_g=\text{diag}(1.3820, 1.5858, 2.6972, 3.6180, 4.4142, 6.3028, 7.0000) \\ \Lambda'_h=\text{diag}(1.5858, 1.5858, 2.0000, 4.4142, 4.4142, 6.0000, 7.0000) \\ \Lambda'_i=\text{diag}(0.0000, 2.0000, 4.0000, 4.0000, 4.0000, 6.0000, 7.0000) \end{cases}$$

$\Lambda_j=\Lambda_k=\text{diag}(-2.7152, -1.2758, -1.0000, 0.0000, 0.0000, 0.0000, 1.0000, 1.2758, 2.7152)$

$$\begin{cases} \Lambda'_j=\text{diag}(3.1294, 4.8773, 5.5858, 6.0000, 7.4838, 8.0000, 8.4142, 8.5095, 9.0000) \\ \Lambda'_k=\text{diag}(3.0437, 4.6972, 5.5612, 7.0000, 7.0000, 7.7454, 8.3028, 8.6498, 9.0000) \end{cases}$$

$\Lambda_l=\Lambda_m=\text{diag}(-2.4289, -2.0693, -1.5279, -0.9182, 0.0000, 0.4528, 1.0000, 1.1354, 1.6037, 2.7523)$

$$\begin{cases} \Lambda'_l=\text{diag}(4.3617, 5.1341, 5.9379, 6.7547, 7.5416, 7.6967, 8.5509, 8.7930, 9.2295, 10.0000) \\ \Lambda'_m=\text{diag}(4.3990, 5.0782, 5.8922, 6.8406, 7.4351, 8.0000, 8.2729, 8.7685, 9.3134, 10.0000) \end{cases}$$

This phenomenon is present in all the cases of the cospectral graphs mentioned in [Collatz and Sinogowitz 1957; Harary *et al.* 1971]. It is interesting to note that the converse, i.e., two non-cospectral graphs based on AMs become cospectral graphs based on AAMs, has not yet happened. However, not all cospectral graphs on AM, which are not isomorphic, have different graph spectrums on AAM. So far, two examples have been found to be true. The first example is the one shown in Figure 4.2 where two cospectral graphs are not isomorphic, but their eigenvalues on AAM are the same, i.e.,

$$\Lambda'_a = \Lambda'_b = \text{diag}(6.4779, 7.2414, 7.7530, 7.7701, 8.4181, 9.4450, 9.7365, 10.1955, 10.8019, 10.8953, 11.2652, 12.0000)$$

The second example is the one shown in Figure 4.10. As discussed before, these two graphs are not isomorphic, but are cospectral on AM. They are also cospectral on AAM.

$$\Lambda'_a = \Lambda'_b = \text{diag}(6.8713, 6.8713, 9.6972, 9.6972, 10.0000, 10.5334, 10.5334, 12.0000, 12.0000, 13.3028, 13.3028, 13.5953, 13.5953, 14.0000, 15.0000)$$

Despite this fact, it is more effective to have the algorithms based on AAM than to have the algorithms based on AM for graph isomorphic identification. Besides, AAM ensures the Eigensystem approach works. The most important advantage of AAM is that it ensures at least one distinct eigenvalue, which meets the condition of algorithms I-1, I-2, and II-3.

4.5 Graph Isomorphism for Digraphs

The graph isomorphism problem for a directed graph is discussed here. Both AM and AAM are symmetrical for an undirected graph, and therefore their eigenvalues and eigenvectors are real. For a digraph, however, the AM (or AAM) is not symmetrical and thus the complex number may present in eigenvalues and eigenvectors of the AM (or AAM). Therefore, the algorithms (I-1, I-2, II-3) developed previously may not be directly applied for GI detection for digraphs. For an unsymmetrical matrix, there is still the eigenvalue issue. It can be easily proved that when eigenvalues are distinct, their corresponding eigenvectors are unique. Therefore, it appears that the three algorithms developed for the undirected graphs may be adapted to be useful to the directed graph.

For an eigenvalue λ (real or complex number), $\mathbf{x}$ (real or complex number) is the eigenvector corresponding to λ when the equation $A\mathbf{x}=\lambda\mathbf{x}$ is valid. For Λ, which is a diagonal matrix containing all eigenvalues (real and complex number), there is $AX=X\Lambda$, i.e.,

$$A=X\Lambda X^{-1} \tag{4.10}$$

where X contains the eigenvectors (real or complex number) corresponding to Λ. It is easy to understand that the definition of graph isomorphism for the undirected graph is equally applicable to the directed graph. This means that Equation (4.1) should be satisfied for two isomorphic directed graphs. Now let A and B be the AAMs of two directed graphs, respectively. Suppose that these two graphs have the same graph spectrum Λ. This implies the following equation:

$$B=Y\Lambda Y^{-1}. \tag{4.11}$$

Then substituting Equation (4.10) and Equation (4.11) into Equation (4.1) leads to

$$XAX^{-1}=PYAY^{-1}P^{T} \qquad (4.12)$$

Noticing $P^T=P^{-1}$ and $(PY)^{-1}=Y^{-1}P^{-1}$, Equation (4.12) can be further written as

$$XAX^{-1}=(PY)A(PY)^{-1} \qquad (4.13)$$

If each eigenvalue of A is distinct from other eigenvalues of A, similar with Theorem 1, Equation (4.13) satisfies if and only if there is

$$X=PYS \qquad (4.14)$$

where S is a sign matrix

$$S=\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_n \end{bmatrix} \qquad (4.15)$$

where the diagonal entries of S are either ±1, or ±i. In fact, substituting both Equation (4.14) and Equation (4.15) into Equation (4.10) leads to

$$A = PYS\Lambda(PYS)^{-1} = PYS\Lambda S^{-1}Y^{-1}P^{-1} = PY(S\Lambda S^{-1})Y^{-1}P^{-1}$$

$$= PY\begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_n \end{bmatrix}\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}\begin{bmatrix} 1/s_1 & 0 & \cdots & 0 \\ 0 & 1/s_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/s_n \end{bmatrix}Y^{-1}P^{-1}$$

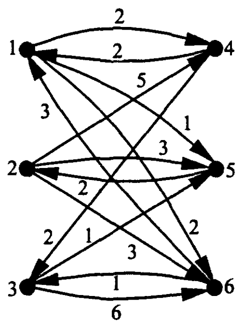
$$= PY\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}Y^{-1}P^{-1} = P(Y\Lambda Y^{-1})P^{-1} = PBP^T$$

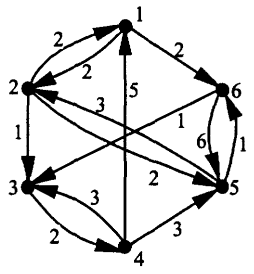
The above discussion actually implies: Theorem 1 is valid to the directed graph subject to the condition that the S matrix may contain ±1, or ±i. It can be further verified that the three algorithms (I-1, I-2, II-3) which are applicable to the undirected graph are valid to the directed graph provided that the following changes are made:

(1) Sorting eigenvalues or eigenvectors in an ascending order with (first) their real part and (then) their complex part.

(2) For each eigenvector pair $(\mathbf{x}_i^a, \mathbf{x}_i^b)$, finding the possible mappings by comparing not only $(\mathbf{x}_i^a, \mathbf{x}_i^b)$ and $(\mathbf{x}_i^a, -\mathbf{x}_i^b)$, but also $(\mathbf{x}_i^a, \mathbf{x}_i^b \cdot \mathrm{i})$ and $(\mathbf{x}_i^a, -\mathbf{x}_i^b \cdot \mathrm{i})$.

For example, Figure 4.15 describes two weighted digraphs both with 6 vertices. Their AAM are defined as





(a) (b)

Figure 4.15 Two weighted digraphs both with 6 vertices.

$$A=\begin{bmatrix} 3 & 0 & 0 & 2 & 1 & 2 \\ 0 & 3 & 0 & 5 & 3 & 3 \\ 0 & 0 & 4 & 0 & 1 & 6 \\ 2 & 0 & 2 & 4 & 0 & 0 \\ 0 & 2 & 0 & 0 & 5 & 0 \\ 3 & 0 & 1 & 0 & 0 & 4 \end{bmatrix} \qquad B=\begin{bmatrix} 4 & 2 & 0 & 0 & 0 & 2 \\ 2 & 3 & 1 & 0 & 2 & 0 \\ 0 & 0 & 5 & 2 & 0 & 0 \\ 5 & 0 & 3 & 3 & 3 & 0 \\ 0 & 3 & 0 & 0 & 4 & 1 \\ 0 & 0 & 1 & 0 & 6 & 4 \end{bmatrix}$$

The eigenvalues and the corresponding eigenvectors are, respectively,

$$\Lambda_a=\Lambda_b=\text{diag}(-1.0383, 2.3589, 3.7915\text{-}1.4728i, 3.7915\text{+}1.4728i, 5.5333, 8.5631)$$

$$X_a=\begin{pmatrix} 0.3349 & -0.0330 & -0.2122-0.0279i & -0.2122+0.0279i & 0.0372 & -0.2822 \\ 0.7316 & 0.7727 & -0.4289-0.1391i & -0.4289+0.1391i & -0.2359 & -0.6353 \\ 0.3739 & -0.1109 & 0.5545 & 0.5545 & 0.1882 & -0.4525 \\ -0.2814 & 0.1753 & -0.1017-0.4504i & -0.1017+0.4504i & 0.2940 & -0.3221 \\ -0.2423 & -0.5852 & 0.1727+0.4407i & 0.1727-0.4407i & -0.8848 & -0.3566 \\ -0.2736 & 0.1279 & -0.0481+0.0627i & -0.0481-0.0627i & 0.1956 & -0.2847 \end{pmatrix}$$

$$X_b=\begin{pmatrix} -0.2814 & 0.1753 & -0.1017-0.4504i & -0.1017+0.4504i & 0.2940 & -0.3221 \\ 0.3349 & -0.0330 & -0.2122-0.0279i & -0.2122+0.0279i & 0.0372 & -0.2822 \\ -0.2423 & -0.5852 & 0.1727+0.4407i & 0.1727-0.4407i & -0.8848 & -0.3566 \\ 0.7316 & 0.7727 & -0.4289-0.1391i & -0.4289+0.1391i & -0.2359 & -0.6353 \\ -0.2736 & 0.1279 & -0.0481+0.0627i & -0.0481-0.0627i & 0.1956 & -0.2847 \\ 0.3739 & -0.1109 & 0.5545 & 0.5545 & 0.1882 & -0.4525 \end{pmatrix}$$

It can be found that these two digraphs have the same graph spectrum, and then eigenvalues are distinct. Further, their corresponding eigenvectors (X_a and X_b) are equivalent. The following mapping can be found by algorithm I-2, as shown in Table 4.8.

Table 4.8 The one-to-one mapping between two weighted digraphs.

Graph	$\phi_{1+} \cap \phi_{2+} \cap \phi_{3+} \cap \phi_{4+} \cap \phi_{5+} \cap \phi_{6+}$					
A	1	2	3	4	5	6
B	2	4	6	1	3	5

Therefore, these two weighted digraphs are isomorphic.

4.6 Complexity Analysis

From algorithms I-1 and I-2, the time complexity for a single detecting cycle is dominated by (1) the eigendecomposition of AAM, (2) comparison of unique eigenvectors between graphs, and (3) the validations of vertex mappings. The time cost for the comparison of unique eigenpairs of two graphs is mainly for sorting eigenvalues and sorting each eigenvector. The time complexity of algorithms I-1 and I-2 can be analyzed, as shown in Table 4.9 and 4.10, respectively. It is noted that for an n-size sorting problem, $O(n\lg n)$

runs are needed [Cormen *et al.* 2001]. For an n-size eigendecomposition problem, $O(n^3)$ runs are needed [Bai *et al.* 2000].

Table 4.9 The time complexity of algorithm I-1.

Step No.	Time Complexity	Notes
1	$T_1 = O(n^3)$	Eigendecomposing AAMs
2	$T_2 = O(n\lg n)$	Sorting eigenvalues
4 – 8	$T_{4\text{-}8} = O(n)$	Comparing eigenvalues
Total	$T_{I\text{-}1} = O(n^3 + n\lg n + n)$	

Table 4.10 The time complexity of algorithm I-2.

Step No.	Time Complexity	Notes
1	$T_1 = O(n^3)$	Eigendecomposing AAMs
3	$m_1(T_{4\text{-}5} + T_{7\text{-}15} + T_{17})$	m_1 is n in the worst case
4 – 5	$T_{4\text{-}5} = O(n\lg n)$	Sorting eigenvectors
7 –15	$T_{7\text{-}15} = O(n)$	Comparing eigenvectors
17	$T_{17} = O(m_2)$	m_2 is 2^n in the worst case
Total	$T_{I\text{-}2} = O(n^3 + m_1(n\lg n + n + m_2))$	

Hence, in the worst case, the computational complexities for algorithms I-1 and I-2 are $O(n^3)$ and $O(n^3 + n2^n)$, respectively. However, the practical computational complexity of these algorithms in most cases is much better than in the worst case. Actually, the application of some rules, which will be discussed in Chapter 5, can prune the searching of possible mappings and thus speed the performance of these algorithms.

When there are group-to-group vertex mappings for the graphs having coincident eigenvalues, a recursive process has to be done, which may consist of several detecting cycles, as described in algorithm II-3. Hence, the time complexity of algorithm II-3 can be written as

$$T_{II\text{-}3} = m_3\,(T_{I\text{-}1} + T_{I\text{-}2})$$

where m_3 is an integer without any reasonable bound, i.e., $T_{II\text{-}3} = O(m_3n^3 + m_3m_1(n\lg n + n + m_2))$.

4.7 Discussion and Concluding Remarks

The algorithms in the Eigensystem approach for graph isomorphism were introduced in this chapter. Since the eigenvector corresponding to a distinct eigenvalue is unique, both eigenvalues and unique eigenvectors can be applied for determining if two graphs are isomorphic. It was proven that two graphs, each of which has all distinct eigenvalues, are isomorphic if and only if they have the same graph spectrum and their corresponding eigenvector matrices are equivalent; algorithms I-1 and I-2 were developed for this case. If coincident eigenvalues exist in graph spectrums, only the eigenvectors corresponding to distinct eigenvalues can be used for finding possible one-to-one mappings of the vertices between two graphs. If not one-to-one but group-to-group mappings exist, a checking program has to be performed to find possible one-to-one mappings; this program is algorithm II-3. The implementation of these algorithms can be seen in Applendix A.

The Eigensystem approach requires that the adjacency matrix of a graph must have at least one distinct eigenvalue. A new matrix called adjusted adjacency matrix (AAM) was proposed for representing a graph. AAM ensures at least one distinct eigenvalue, which meets the condition of Corollary 1. It was also found that use of AAM for representing graphs and conducting graph isomorphism detections would be more efficient than use of AM. In particular, it has been found that cospectral graphs in terms of AM are likely not cospectral in terms of AAM. This interesting point has further led to the finding that cospectral in terms of AM but non-isomorphic graphs are not cospectral in terms of AAM. This means that the graph isomorphic detection based on AAM would need less computation time.

In general, the AAM of a diagraph is an unsymmetrical matrix. Complex numbers may exist in eigenvalues and in the corresponding eigenvectors. It has been shown that with modifications, algorithms (I-1, I-2, II-3) developed for the undirected graph can be applied to the directed graph. It is noted that most of the algorithms for graph isomorphism published for engineering and science application have not dealt with the digraph.

The computational complexity of the Eigensystem approach has been analyzed. It has been shown that this approach does not render to an algorithm with a polynomial time. In the worst case, the approach reaches the complexity of exponential time. This result is the same as that achieved by the Nauty program. However, in practice, the use of AAM could greatly reduce the computational time, as the co-spectra on AM is (highly) likely not the co-spectra on AAM. Interestly, it is noted from the present study that such a likelihood could go with 80-90%.

CHAPTER 5
GRAPH COUNTING AND STRUCTURE ENUMERATION

5.1 Introduction

The Eigensystem approach (Chapter 3) with its algorithms (Chapter 4) has so far provided a tool for solving the second fundamental problem raised in Chapter 1. The first and third fundamental problems raised in Chapter 1 are related to the graph counting problem, in particular (1) the counting of two isomorphic graphs and (2) the counting of automorphisms (for one graph). The graph counting essentially explores the property of the symmetry of a graph. In this chapter, the Eigensystem approach to the graph counting problem is studied. In particular, Section 5.2 discusses the graph counting problem. Section 5.3 discusses how to uniquely label the graph based on the solution proposed for the graph counting problem. It is important to note that the canonical labeling is a key step towards an effective and efficient method for computer storage of graphs, and thus it is a foundation for addressing the first and third fundamental problems (see Chapter 1). Section 5.4 discusses the structure enumeration under certain constraints. Finally, a concluding remark is given in Section 5.5.

5.2 The Graph Counting Problem

The basic notion of the graph counting problem and the basic idea to solve this problem are first discussed here. It is known from Chapter 4 that two graphs, represented by their adjacency matrices A and B, are isomorphic if there is a row permutation matrix P such that Equation (4.1) is satisfied. The graph counting problem is to find all different P matrices that satisfy Equation (4.1). The number of different P matrices presents the degree of symmetry of two graphs. For example, the two graphs shown in Figure 4.7 are isomorphic, as discussed in Chapter 4, and they have two isomorphic mappings. It can be intuitively observed from the graph shown in Figure 4.7b that the graph is left-right symmetrical along the pattern enclosed by the vertex set (1, 2, 3, 4, 5, 7). Another pair of isomorphic graphs shown in Figure 4.9 has 14 isomorphisms; the number of isomorphisms for this graph is much higher than that of the graph shown in Figure 4.7 (2 in this case). The reason is easy to understand; i.e., the former has a higher degree of the symmetry than the latter.

The counting of automorphisms describes the symmetry of a graph. Its mathematical definition can be stated as follows (see also Chapter 2):

$$A = PAP^T \tag{5.1}$$

where A is the adjacency matrix (or the adjusted adjacency matrix) of a graph and P is a row permutation matrix. There could be more than one P matrix that satisfies the above equation. The number of such P matrices is the number of automorphisms of a graph.

The Eigensystem approach (i.e., algorithms I-1, I-2, and II-3 discussed in Chapter 4) can be applied for solving the graph counting problem if the control flow of algorithm II-3 is changed to: the search for a one-to-one mapping continues until all possible mappings between two graphs are examined. Algorithm III-4 modifies algorithm II-3, as mentioned, and is described in the following:

III-4 ***counting permutation matrices of two graphs (A, B, Φ)***

```
1.  for each mapping $\phi \in \Phi$, do
2.      if mapping $\phi$ is a one-to-one mapping
3.          if $A=\phi B\phi^T$ then $\Psi \leftarrow \phi$
4.      else
5.          finding the mapping group g in $\phi$ having the least number of vertices
6.          $u_x$ = randomly select one member from g on $A$ side
7.          for each member $v_y$ in group g on $B$ side, do
8.              suppose that $A$ and $B$ are isomorphic with $u_x \leftrightarrow v_y$ by modifying both the
                elements of $A$ at $a_{xx}$ and B at $b_{yy}$ with $a_{xx}+x$ and $b_{yy}+x$ into $A'$ and $B'$,
                respectively
9.              comparison of the spectrums of two graphs ($A'$, $B'$)
10.             $\Phi'$ = finding permutation matrix of two graphs ($A'$, $B'$)
11.             $\Psi$= counting permutation matrices of two graphs ($A'$, $B'$, $\Phi'$)
```

5.2.1 Counting of Isomorphisms

Counting of isomorphisms can be illustrated with the following example using algorithms I-1, I-2, and III-4. Two isomorphic graphs are constructed as follows. Figure 5.1a shows a 30-vertex graph which is a "master" graph. Figures 5.1b and 5.1c show two 28-vertex graphs, which are derived from the master graph by removing two respective vertices and their edges, respectively. The two 28-vertex graphs have the same graph spectrum on AAMs, i.e.,

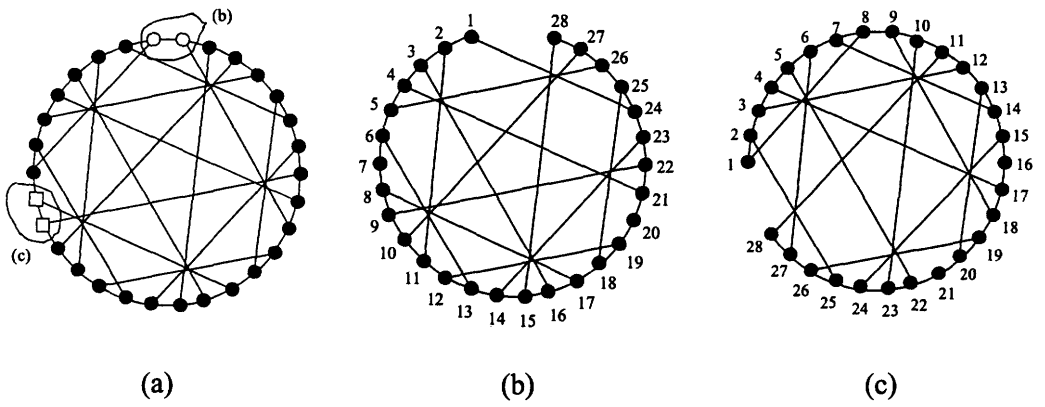


Figure 5.1 An example of counting of isomorphisms.

Λ_b=Λ_c=diag(22.1442, 23.0000, 23.0000, 23.0000, 23.0000, 23.0000, 23.1864, 23.1864, 23.5858, 24.6784, 25.0000, 25.0000, 25.0000, 25.0000, 25.0000, 25.0000, 25.4707, 25.4707, 26.4142, 27.0000, 27.0000, 27.0000, 27.0000, 27.0000, 27.1774, 27.3429, 27.3429, 28.0000)

There are six distinct eigenvalues for each of the two graphs shown in Figure 5.1b and 5.1c, i.e., λ_1, λ_9, λ_{10}, λ_{19}, λ_{25}, and λ_{28}. The eigenvectors of the two graphs corresponding to these distinct eigenvalues are shown in Figure 5.2. The mappings for each pair of unique eigenvectors are created and listed in Figure 5.3. Based on the mappings shown in Figure 5.3, the common mappings, i.e., Φ_{28}, are created and listed in Table 5.1.

$$X_b=\begin{pmatrix}
-0.0933 & & 0.2380 & -0.3166 & & -0.3971 & & -0.3755 & & 0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & -0.0822 & & -0.2211 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & -0.0822 & & -0.2211 & & 0.1890 \\
-0.0933 & & 0.2380 & -0.3166 & & -0.3971 & & -0.3755 & & 0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & -0.0822 & & -0.2211 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & \cdots & 0.0841 & -0.1247 & \cdots & 0.1404 & \cdots & 0.0529 & \cdots & 0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & -0.0822 & & 0.2211 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & -0.0822 & & 0.2211 & & 0.1890 \\
0.0933 & & 0.2380 & 0.3166 & & -0.3971 & & 0.3755 & & 0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & -0.0822 & & 0.2211 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & -0.0822 & & -0.2211 & & 0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & 0.1404 & & -0.0529 & & 0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & 0.1404 & & 0.0529 & & 0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & -0.0822 & & 0.2211 & & 0.1890 \\
0.0933 & & 0.2380 & 0.3166 & & -0.3971 & & 0.3755 & & 0.1890
\end{pmatrix}$$

Figure 5.2 The eigenvectors corresponding to the distinct eigenvalues.

$$X_c=\begin{pmatrix}
-0.0933 & & 0.2380 & -0.3166 & & 0.3971 & & -0.3755 & & -0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & 0.0822 & & -0.2211 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & 0.0822 & & -0.2211 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & 0.0822 & & 0.2211 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & \cdots & 0.0841 & -0.1247 & \cdots & -0.1404 & \cdots & 0.0529 & \cdots & -0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & 0.0822 & & 0.2211 & & -0.1890 \\
0.0933 & & 0.2380 & 0.3166 & & 0.3971 & & 0.3755 & & -0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & 0.0822 & & 0.2211 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & 0.0822 & & -0.2211 & & -0.1890 \\
-0.0933 & & 0.2380 & -0.3166 & & 0.3971 & & -0.3755 & & -0.1890 \\
0.1798 & & -0.2873 & 0.2092 & & 0.0822 & & -0.2211 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.2101 & & 0.0841 & 0.1247 & & -0.1404 & & -0.0529 & & -0.1890 \\
0.2101 & & 0.0841 & -0.1247 & & -0.1404 & & 0.0529 & & -0.1890 \\
-0.1798 & & -0.2873 & -0.2092 & & 0.0822 & & 0.2211 & & -0.1890 \\
0.0933 & & 0.2380 & 0.3166 & & 0.3971 & & 0.3755 & & -0.1890
\end{pmatrix}$$

Figure 5.2 (*Continued*)

ϕ_{1+}		ϕ_{1-}		ϕ_{9+}		ϕ_{9-}	ϕ_{10+}		ϕ_{10-}		ϕ_{19+}	ϕ_{19-}		ϕ_{25+}		ϕ_{25-}		ϕ_{28}	
$B \leftrightarrow C$		$B \leftrightarrow C$		$B \leftrightarrow C$			$B \leftrightarrow C$		$B \leftrightarrow C$			$B \leftrightarrow C$		$B \leftrightarrow C$		$B \leftrightarrow C$		$B \leftrightarrow C$	
3	3	3	4	2	2		1	1	1	16		1	1	1	1	1	16	1	1
5	5	5	6	6	8		7	21	7	28		7	16	7	21	7	28	2	2
9	7	9	10	8	11		15	11	15	2		20	21	2	2	2	11	3	3
11	9	11	12	15	15		19	15	19	8		28	28	6	8	6	15	4	4
13	13	13	14	19	17		21	17	21	20		2	2	8	20	8	17	5	5
17	19	17	18	21	20		27	27	27	22		6	8	24	22	24	27	6	6
23	23	23	24	24	22		4	4	4	3		8	11	3	3	3	4	7	7
25	25	25	26	27	27		10	6	10	5		15	15	5	5	5	6	8	8
15	11	15	2	3	3		12	10	12	7		19	17	9	7	9	10	9	9
19	15	19	8	4	4		14	12	14	9		21	20	11	9	11	12	10	10
21	17	21	20	5	5		16	14	16	13		24	22	13	13	13	14	11	11
27	27	27	22	9	6		18	18	18	19		27	27	17	19	17	18	12	12
1	1	1	16	10	7		22	24	22	23		3	3	23	23	23	24	13	13
7	21	7	28	11	9		26	26	26	25		4	4	25	25	25	26	14	14
20	16	20	1	12	10	∅	3	3	3	4	∅	5	5	4	4	4	3	15	15
28	28	28	21	13	12		5	5	5	6		9	6	10	10	10	5	16	16
2	2	2	11	14	13		9	7	9	10		10	7	12	12	12	7	17	17
6	8	6	15	16	14		11	9	11	12		11	9	14	14	14	9	18	18
8	20	8	17	17	18		13	13	13	14		12	10	16	16	16	13	19	19
24	22	24	27	18	19		17	19	17	18		13	12	18	18	18	19	20	20
4	4	4	3	22	23		23	23	23	24		14	13	22	22	22	23	21	21
10	6	10	5	23	24		25	25	25	26		16	14	26	26	26	25	22	22
12	10	12	7	25	25		2	2	2	11		17	18	15	15	15	2	23	23
14	12	14	9	26	26		6	8	6	15		18	19	19	19	19	8	24	24
16	14	16	13	1	1		8	20	8	17		22	23	21	21	21	20	25	25
18	18	18	19	7	16		24	22	24	27		23	24	27	27	27	22	26	26
22	24	22	23	20	21		20	16	20	1		25	25	20	20	20	1	27	27
26	26	26	25	28	28		28	28	28	21		26	26	28	28	28	21	28	28

Figure 5.3 The mappings created for each pair of unique eigenvectors.

Table 5.1 Common mappings Φ_{28} to the unique eigenvectors.

Φ	Operation	Result
Φ_1	$\phi_{1+} \cup \phi_{1-}$	$\phi_{1+} \cup \phi_{1-}$
Φ_9	$(\Phi_1 \cap \phi_{9+}) \cup (\Phi_1 \cap \phi_{9-}) = \Phi_1 \cap \phi_{9+} = (\phi_{1+} \cap \phi_{9+}) \cup (\phi_{1-} \cap \phi_{9+})$	$\phi_{1+} \cup \phi_{1-}$
Φ_{10}	$(\Phi_9 \cap \phi_{10+}) \cup (\Phi_9 \cap \phi_{10-}) = (\phi_{1+} \cap \phi_{10+}) \cup (\phi_{1-} \cap \phi_{10-})$	$\phi_{1+} \cup \phi_{1-}$
Φ_{19}	$(\Phi_{10} \cap \phi_{19+}) \cup (\Phi_{10} \cap \phi_{19-}) = \Phi_{10} \cap \phi_{19-} = (\phi_{1+} \cap \phi_{19-}) \cup (\phi_{1-} \cap \phi_{19-})$	$\phi_{1+} \cup \phi_{1-}$
Φ_{25}	$(\Phi_{19} \cap \phi_{25+}) \cup (\Phi_{19} \cap \phi_{25-}) = (\phi_{1+} \cap \phi_{25+}) \cup (\phi_{1-} \cap \phi_{25-})$	$\phi_{1+} \cup \phi_{1-}$
Φ_{28}	$\Phi_{25} \cap \phi_{28} = (\phi_{1+} \cap \phi_{28}) \cup (\phi_{1-} \cap \phi_{28})$	$\phi_{1+} \cup \phi_{1-}$

Two common group-to-group mappings, i.e., ϕ_{1+} and ϕ_{1-}, are obtained from Table 5.1. According to the definition of the graph counting problem, one needs to find all one-to-one mappings (if any) from these two group-to-group mappings. In the following, the procedure (using algorithms I, II, and III) for identifying only one one-to-one mapping from ϕ_{1+} is demonstrated (the procedure for getting the other one-to-one mappings is similar).

Figure 5.4 illustrates the iterative procedure for searching for a one-to-one mapping from ϕ_{1+} (see the first column). Both $B(1, 7) \leftrightarrow C(1, 21)$ and $B(20, 28) \leftrightarrow C(16, 28)$ groups in ϕ_{1+} have the least number of vertices (two). One of these two groups, e.g., $B(1, 7) \leftrightarrow C(1, 21)$ in this case, is thus selected as a start. Assume that $B(1) \leftrightarrow C(1)$ and change the entries b_{11} and c_{11} of the AAMs of these two graphs into $b_{11} = b_{11} + 1$ and $c_{11} = c_{11} +1$. The two modified AAMs are regarded as two new inputs to algorithms I-1 and I-2. Then some new common mappings could be created, such as the mapping shown in the second column in Figure 5.4. Choose the group $B(19, 21) \leftrightarrow C(15, 17)$ from the second column as an example and start with $B(19) \leftrightarrow C(15)$ in particular. With a new iterative loop, a new mapping is created as shown in the third column in Figure 5.4. Likewise, choose the group $B(5, 9) \leftrightarrow C(5, 19)$ from the third column as an example and start with $B(5) \leftrightarrow C(5)$ in particular. This results in a new common mapping shown in the fourth column in Figure 5.4. Likewise, choose the group $B(2, 24) \leftrightarrow C(2, 8)$ from the fourth column as an example and start with $B(2) \leftrightarrow C(2)$ in particular. Then a new common mapping is created in the fifth column in Figure 5.4. There is only one group in this new common mapping. Two mappings are possible in this group, i.e., $\{B(10) \leftrightarrow C(12), B(16) \leftrightarrow C(26)\}$ and $\{B(10) \leftrightarrow C(26), B(16) \leftrightarrow C(12)\}$. Start with $B(10) \leftrightarrow C(12)$ in particular. No common mapping can be found in the sixth column in Figure 5.4. Then, go back to the fifth column in Figure 5.4 and consider $B(10) \leftrightarrow C(26)$. This results in a final common one-to-one mapping, as shown in the seventh column in Figure 5.4. The whole procedure is a depth-first backtracking search. The seventh column in Figure 5.4 lists the one-to-one mapping which was first obtained through this backtracking search. There are in total 32 isomorphic mappings between the two graphs which can be found, and they are listed in Table 5.2.

1		2		3		4		5		6	7	
ϕ_{1+}		*Let* **1 ↔ 1**		*Let* **19 ↔ 15**		*Let* **5 ↔ 5**		*Let* **2 ↔ 2**		**10↔**	*Let* **10 ↔ 26**	
B* ↔ *C		***B* ↔ *C***		***B* ↔ *C***		***B* ↔ *C***		***B* ↔ *C***		**12**	***B* ↔ *C***	
3	3	5	5	5	5	9	19	9	19		9	19
5	5	9	13	9	19	**5**	**5**	5	5		5	5
9	7	13	19	13	13	17	13	17	13		17	13
11	9	17	23	17	23	13	23	13	23		13	23
13	13	3	3	3	3	23	9	23	9		23	9
17	19	11	7	23	9	3	3	3	3		3	3
23	23	23	9	11	7	11	25	11	25		11	25
25	25	25	25	25	25	25	7	25	7		25	7
15	11	19	15	21	17	21	17	21	17		21	17
19	15	21	17	**19**	**15**	19	15	19	15		19	15
21	17	15	11	15	11	15	11	15	11		15	11
27	27	27	27	27	27	27	27	27	27		27	27
1	1	7	21	7	21	7	21	7	21		7	21
7	21	**1**	**1**	1	1	1	1	1	1		1	1
20	16	28	28	28	28	28	28	28	28	∅	28	28
28	28	20	16	20	16	20	16	20	16		20	16
2	2	2	2	2	2	2	2	**2**	**2**		2	2
6	8	24	8	24	8	24	8	24	8		24	8
8	20	6	20	6	20	6	22	6	22		6	22
24	22	8	22	8	22	8	20	8	20		8	20
4	4	10	6	10	6	26	6	26	6		26	6
10	6	14	10	14	10	10	10	10	12		**10**	**26**
12	10	16	12	16	12	14	12	16	26		16	12
14	12	26	26	26	26	16	26	14	10		14	10
16	14	4	4	12	14	12	14	12	24		12	24
18	18	12	14	18	24	18	24	18	14		18	14
22	24	18	18	4	4	4	4	4	4		4	4
26	26	22	24	22	18	22	18	22	18		22	18

Figure 5.4 The iterative procedure for search for a one-to-one isomorphic mapping.

Table 5.2 Counting of isomorphisms between two graphs shown in Figure 5.1.

Graph	Vertex Label																											
(b)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
	1	2	3	4	5	22	21	20	19	26	25	24	23	10	11	12	13	14	15	16	17	18	9	8	7	6	27	28
	1	2	25	24	23	22	21	20	13	12	3	4	5	6	27	26	19	18	17	16	15	14	7	8	9	10	11	28
	1	8	7	14	13	20	21	22	23	10	9	18	19	26	27	6	5	4	17	16	15	24	25	2	3	12	11	28
	1	8	9	18	19	20	21	22	5	6	7	14	13	12	11	10	23	24	15	16	17	4	3	2	25	26	27	28
	1	2	3	12	13	20	21	22	23	24	25	26	19	18	17	4	5	6	27	28	11	10	9	8	7	14	15	16
	1	2	25	26	19	20	21	22	5	4	3	12	13	14	15	24	23	10	11	28	27	6	7	8	9	18	17	16
	1	8	7	6	5	22	21	20	19	18	9	10	23	24	15	14	13	12	11	28	27	26	25	2	3	4	17	16
	1	8	9	10	23	22	21	20	13	14	7	6	5	4	17	18	19	26	27	28	11	12	3	2	25	24	15	16
	21	20	13	14	7	8	1	2	25	26	19	18	9	10	11	12	3	4	17	16	15	24	23	22	5	6	27	28
	21	20	19	18	9	8	1	2	3	12	13	14	7	6	27	26	25	24	15	16	17	4	5	22	23	10	11	28
	21	22	5	4	3	2	1	8	9	10	23	24	25	26	27	6	7	14	15	16	17	18	19	20	13	12	11	28
	21	22	23	24	25	2	1	8	7	6	5	4	3	12	11	10	9	18	17	16	15	14	13	20	19	26	27	28
	21	20	13	12	3	2	1	8	9	18	19	26	25	24	15	14	7	6	27	28	11	10	23	22	5	4	17	16
	21	20	19	26	25	2	1	8	7	14	13	12	3	4	17	18	9	10	11	28	27	6	5	22	23	24	15	16
	21	22	5	6	7	8	1	2	25	24	23	10	9	18	17	4	3	12	11	28	27	26	19	20	13	14	15	16
	21	22	23	10	9	8	1	2	3	4	5	6	7	14	15	24	25	26	27	28	11	12	13	20	19	18	17	16
(c)	16	15	14	13	12	11	28	27	26	25	24	23	10	9	8	7	6	5	22	21	20	19	18	17	4	3	2	1
	16	15	24	23	10	11	28	27	6	7	14	13	12	3	2	25	26	19	20	21	22	5	4	17	18	9	8	1
	16	17	4	5	6	27	28	11	10	9	18	19	26	25	2	3	12	13	20	21	22	23	24	15	14	7	8	1
	16	17	18	19	26	27	28	11	12	3	4	5	6	7	8	9	10	23	22	21	20	13	14	15	24	25	2	1
	16	15	14	7	6	27	28	11	10	23	24	25	26	19	20	13	12	3	2	1	8	9	18	17	4	5	22	21
	16	15	24	25	26	27	28	11	12	13	14	7	6	5	22	23	10	9	8	1	2	3	4	17	18	19	20	21
	16	17	4	3	12	11	28	27	26	19	18	9	10	23	22	5	6	7	8	1	2	25	24	15	14	13	20	21
	16	17	18	9	10	11	28	27	6	5	4	3	12	13	20	19	26	25	2	1	8	7	14	15	24	23	22	21
	28	11	10	23	24	15	16	17	4	3	12	13	14	7	8	9	18	19	20	21	22	5	6	27	26	25	2	1
	28	11	12	13	14	15	16	17	18	9	10	23	24	25	2	3	4	5	22	21	20	19	26	27	6	7	8	1
	28	27	6	5	4	17	16	15	24	25	26	19	18	9	8	7	14	13	20	21	22	23	10	11	12	3	2	1
	28	27	26	19	18	17	16	15	14	7	6	5	4	3	2	25	24	23	22	21	20	13	12	11	10	9	8	1
	28	11	10	9	18	17	16	15	14	13	12	3	4	5	22	23	24	25	2	1	8	7	6	27	26	19	20	21
	28	11	12	3	4	17	16	15	24	23	10	9	18	19	20	13	14	7	8	1	2	25	26	27	6	5	22	21
	28	27	6	7	14	15	16	17	18	19	26	25	24	23	22	5	4	3	2	1	8	9	10	11	12	13	20	21
	28	27	26	25	24	15	16	17	4	5	6	7	14	13	20	19	18	9	8	1	2	3	12	11	10	23	22	21

5.2.2 Counting of Automorphisms

The difference between Equation (4.1) and Equation (5.1) is that matrix B in Equation (4.1) is replaced by matrix A in Equation (5.1). Hence, the counting problem of automorphisms can be readily solved with algorithms I-1, I-2, and III-4 by replacing B with A in Equation (4.1). Figure 5.5 shows a 14-vertex graph; its symmetry is easily observed. The eigenvalues of this graph, based on its AAM, are

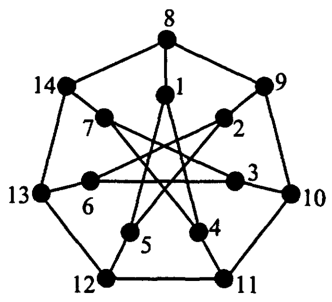


Figure 5.5 A 14-vertex graph.

Λ=diag(8.6681, 8.6681, 8.8993, 8.8993, 10.0911, 10.0911, 11.0849, 11.0849, 12.0000, 12.5457, 12.5457, 12.7108, 12.7108, 14.0000)

There are two distinct eigenvalues, i.e., λ_9 and λ_{14}. The eigenvectors $\mathbf{x}_9$ and $\mathbf{x}_{14}$ that correspond the eigenvalues λ_9 and λ_{14}, respectively, are:

$\mathbf{x}_9$ = [0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 -0.2673 -0.2673 -0.2673 -0.2673 -0.2673 -0.2673 -0.2673]^T

$\mathbf{x}_{14}$ = [0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673 0.2673]^T

Two groups are created according to $\mathbf{x}_9$ and $\mathbf{x}_{14}$. One group consists of half of the vertices from label 1 to label 7, and the other group consists of the other half of vertices from label 8 to label 14. From these two groups, one can find two group-to-group mappings;

see Figure 5.6 (one mapping is shown with the solid arrow and the other with the dashed arrow).

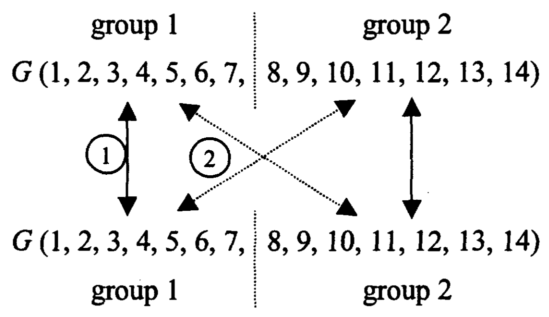


Figure 5.6 Two group-to-group mappings.

As an example, consider the group-to-group mapping ①; especially start with $G(1) \leftrightarrow G(1)$ for searching for one-to-one mappings. According to algorithm II-3 (or III-4), modify g_{11} of the AAM of G into $g_{11}+1$, and obtain a new AAM. After the new AAM is input to the algorithms (algorithms I-1 and II-2), one can find two one-to-one mappings which are listed as No. 1 and No. 2 in Table 5.3, respectively. Continuing this procedure, one can finally obtain all the one-to-one mappings for this example, which are listed in Table 5.3 (in total, 14 one-to-one mappings). This concludes that for the graph shown in Figure 5.5, there are 14 automorphisms in total.

Table 5.3 Counting of automorphisms for the graph shown in Figure 5.5.

Auto.	**Vertex Label**													
No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	7	6	5	4	3	2	8	14	13	12	11	10	9
2	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	2	1	7	6	5	4	3	9	8	14	13	12	11	10
4	2	3	4	5	6	7	1	9	10	11	12	13	14	8
5	3	4	5	6	7	1	2	10	11	12	13	14	8	9
6	3	2	1	7	6	5	4	10	9	8	14	13	12	11
7	4	3	2	1	7	6	5	11	10	9	8	14	13	12
8	4	5	6	7	1	2	3	11	12	13	14	8	9	10
9	5	6	7	1	2	3	4	12	13	14	8	9	10	11
10	5	4	3	2	1	7	6	12	11	10	9	8	14	13
11	6	7	1	2	3	4	5	13	14	8	9	10	11	12
12	6	5	4	3	2	1	7	13	12	11	10	9	8	14
13	7	6	5	4	3	2	1	14	13	12	11	10	9	8
14	7	1	2	3	4	5	6	14	8	9	10	11	12	13

Consider the graph shown in Figure 5.7, which is isomorphic to the graph shown in Figure 5.5. It can be found that there are 14 isomorphic mappings between the graph shown in Figure 5.5 and the graph shown in Figure 5.7, and they are listed in Table 5.4.

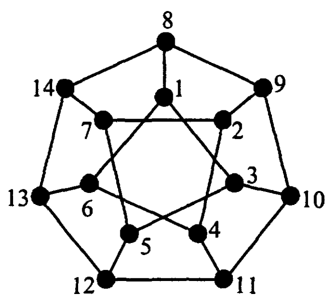


Figure 5.7 Another graph with 14 vertices.

Table 5.4 Counting of isomorphisms between the graphs shown in Figures 5.5 and 5.7.

No.	Vertex Label													
Fig. 5.5	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	8	13	11	9	14	12	10	1	6	4	2	7	5	3
	8	10	12	14	9	11	13	1	3	5	7	2	4	6
	9	11	13	8	10	12	14	2	4	6	1	3	5	7
	9	14	12	10	8	13	11	2	7	5	3	1	6	4
	10	8	13	11	9	14	12	3	1	6	4	2	7	5
	10	12	14	9	11	13	8	3	5	7	2	4	6	1
	11	13	8	10	12	14	9	4	6	1	3	5	7	2
Fig.5.7	11	9	14	12	10	8	13	4	2	7	5	3	1	6
	12	14	9	11	13	8	10	5	7	2	4	6	1	3
	12	10	8	13	11	9	14	5	3	1	6	4	2	7
	13	8	10	12	14	9	11	6	1	3	5	7	2	4
	13	11	9	14	12	10	8	6	4	2	7	5	3	1
	14	12	10	8	13	11	9	7	5	3	1	6	4	2
	14	9	11	13	8	10	12	7	2	4	6	1	3	5

Here one may come to a surprise with the number '14', because it is also the total number of automorphisms of the graph shown in Figure 5.5. Underlying this phenomenon is an important property about the relationship between isomorphisms and automorphisms. The number of isomorphisms between two graphs is the same as the number of automorphisms of each graph. The proof of this property is given below.

Suppose that two graphs, which are represented with their respective AAMs (A and B), are isomorphic with m isomorphisms. That is to say, there are P_1, P_2, ...P_m where P_i is a row permutation matrix for i=1, 2, ...m, and P_i satisfies the following equation:

$$B = P_i A P_i^T \text{ for } i = 1, 2, \ldots m \tag{5.2}$$

Equation (5.2) can be rewritten as

$$B = P_1 A P_1^T = P_2 A P_2^T = \cdots = P_i A P_i^T = \cdots = P_m A P_m^T \tag{5.3}$$

Noticing that $P_1^T P_1 = I$ (identity matrix) and $P_i^T P_1 = \left(P_1^T P_i\right)^T$, there should be

$$A = P_1^T B P_1 = IAI^T = P_1^T P_2 A\left(P_1^T P_2\right)^T = \cdots = P_1^T P_i A\left(P_1^T P_i\right)^T = \cdots = P_1^T P_m A\left(P_1^T P_m\right)^T \tag{5.4}$$

Let $P_i' = P_1^T P_i$ for *i* =2, ...*m*, Equation (5.4) is rewritten as

$$A = IAI^T = P_2' A P_2'^T = \cdots = P_i' A P_i'^T = \cdots = P_m' A P_m'^T \tag{5.5}$$

Equation (5.5) indicates that graph *A* has *m* automorphisms (let $P_1' = I$). Likewise, it can be proved that graph *B* has *m* automorphisms. ■

Suppose that graph *A* has *m* automorphisms, i.e.,

$$A = P_i A P_i^T \text{ for } i = 1, 2, \ldots m \tag{5.6}$$

It is noted that there is a $p \in P_i$ in Equation (5.6) such that *p* is an identity matrix *I*. Now, suppose that graph *B* is isomorphic to *A* with an isomorphism ϕ, i.e.,

$$B = \phi A \phi^T \tag{5.7}$$

There is

$$B = \phi A \phi^T = \phi P_i A P_i^T \phi^T = \left(\phi P_i\right) A \left(\phi P_i\right)^T \text{ for } i = 1, 2, \ldots m \tag{5.8}$$

Equation (5.8) indicates that the two graphs A and B have m isomorphisms between them.

■

With algorithms I-1, I-2, and III-4, two graphs shown in Figure 2.8 are counted, respectively. The graph shown in Figure 2.8a has 3,840 automorphisms, which are the same as the result obtained with the program Nauty [McKay 1981]. The graph shown in Figure 2.8b has 40,320 automorphisms. This result can be illustrated intuitively. Since the graph shown in Figure 2.8b is a complete regular graph (i.e., each vertex has a connection with the other vertices), each vertex in the graph can be replaced by any of the other vertices. Therefore, it has 8! (40,320) automorphisms. This has demonstrated that the Eigensystem approach can work for the worst scenario (i.e., the highest degree of symmetric or regular graph) from the viewpoint of detection of graph automorphisms.

5.3 Canonical Labeling

In order to store graphs in an efficient way, a kind of code uniquely representing a graph is described. Basically, such a kind of code can be generated by concatenating the rows of the (adjusted) adjacency matrix of a graph. For an undirected non-weighted graph, the code can be simplified as a binary number string by concatenating the up-right triangular part of the (adjusted) adjacency matrix, as illustrated in Figure 5.8. However, such a code depends on the labeling of the presented graph and thus is not invariant to its isomorphic graphs. If such a code is designed for uniquely representing a graph, the prerequisite is that the labeling of the graph must be canonical to all its isomorphic graphs. This prerequisite, i.e., the canonical labeling of a graph, can be achieved by the Eigensystem approach through three steps, which are discussed in the following. Prior to discussion of these steps, a basic labeling strategy that must be followed in the three steps is described first. This basic labeling strategy is stated as:

$$\begin{array}{c}\left[\begin{array}{cccccc} 3 & 1 & 1 & 0 & 0 & 1 \\ 1 & 3 & 1 & 0 & 1 & 0 \\ 1 & 1 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 1 & 1 \\ 0 & 1 & 0 & 1 & 3 & 1 \\ 1 & 0 & 0 & 1 & 1 & 3 \end{array}\right]\end{array}\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array}$$

$$\underbrace{1\;1\;0\,\vdots\,0\;1}_{r_1}\;\underbrace{1\;0\,\vdots\,1\;0}_{r_2}\;\underbrace{1\;0\,\vdots\,0}_{r_3}\;\underbrace{1\;1}_{r_4}\;\underbrace{1}_{r_5}$$

$= 66A7$ (hex)

Adjusted adjacency matrix The code

Figure 5.8 Adjusted adjacency matrix and its code.

At anytime when relabeling a graph by comparing the components of the unique eigenvectors of the graph, the vertex corresponding to the minimum component is reassigned the lowest label.

An example following this rule can be seen in Figure 4.5 where the unique eigenvector $\mathbf{x}_1^a\,(\mathbf{x}_1^b)$ was sorted in an ascending order. Vertex 15 has the minimum component (–0.4380) in $\mathbf{x}_1^a$ and thus is reassigned the lowest label (label 1) when relabeling the graph shown in Figure 4.4a, vertex 9 has the second minimum component (-0.2828) in $\mathbf{x}_1^a$ and thus is reassigned the second lowest label (label 2), and so on. Similarly, vertex 11 has the minimum component (–0.4380) in $\mathbf{x}_1^b$ and thus is reassigned the lowest label (label 1) when relabeling the graph shown in Figure 4.4b, vertex 13 has the second minimum component (-0.2828) in $\mathbf{x}_1^b$ and thus is reassigned the second lowest label (label 2), and so on. In the following discussion, this basic strategy is applied without any further elucidation.

5.3.1 Step 1: The Basic Expression

Take the graphs shown in Figure 4.4 as an example to illustrate the steps towards a canonical labeling of a graph. For the two graphs shown in Figure 4.4, the sorting of the first unique eigenvectors $\mathbf{x}_1^a$ and $\mathbf{x}_1^b$ in an ascending order, results in eigenvectors $\mathbf{x}'^a_1$ and $\mathbf{x}'^b_1$, respectively, as shown in Figure 4.5. The ascending orders of $\mathbf{x}'^a_1$ and $\mathbf{x}'^b_1$ are unique because the eigenvectors $\mathbf{x}_1^a$ and $\mathbf{x}_1^b$ are unique. Therefore, the labeling corresponding to $\mathbf{x}'^a_1$ ($\mathbf{x}'^b_1$) is unique in this case and is, hereafter, called the basic expression. Given an arbitrary labeling of a graph, the basic expression can be written out. For the graph with an initial labeling shown in Figure 4.5, the following relabeling leads to the basic expression: vertices {15, 9, 17, 12, 16, 8, 13, 11, 7, 4, 14, 5, 10, 2, 1, 3, 6} of graph *A* are relabeled into vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}. For the graph with an initial labeling shown in Figure 4.5b, the relabeling for the basic expression is such that vertices {11, 13, 17, 9, 16, 5, 4, 8, 6, 3, 14, 15, 10, 12, 2, 7, 1} of graph *B* are relabeled into vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}.

It is clear that the basic expression so far discussed is the unique expression for a graph and all its isomorphic graphs. Therefore, the code generated from the basic expression of the graph shown in Figure 4.4 is canonical, i.e., 00017026220703A0931909142211 A020000. It is noted that the canonical code is reversible to the canonical labeling, the AAM (or AM) matrix, and the graph.

As previously discussed in Chapters 3 and 4, however, both an eigenvector and its negative have the same effect on finding isomorphic (automorphic) mappings in the Eigensystem approach. This situation could result in two different basic expressions for a set of isomorphic graphs when sorting the unique eigenvector corresponding to the minimum distinct eigenvalue. For instance, for the two graphs shown in Figure 4.4, the negative vector $-\mathbf{x}_1^a$ of the eigenvector $\mathbf{x}_1^a$ results in the relabeling as follows: vertices {3, 1, 2, 10, 5, 14, 4, 7, 11, 13, 8, 16, 12, 17, 9, 15} into vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}. Furthermore, the canonical code of the graph in this case is 00123018A20A8170C0C5543048226020000 which is different from the one based on $\mathbf{x}_1^a$. Therefore, a rule is needed to resolve this ambiguity. This rule is hereafter called the Vector Choose Rule I and is described below.

Vector Choose Rule I:

```
(1) sorting the unique eigenvector $\mathbf{x}$ and its negative vector $-\mathbf{x}$ in an ascending order, say
    $\mathbf{x}'$ and $-\mathbf{x}'$, respectively
(2) if $\mathbf{x}'$ is the same as $-\mathbf{x}'$ then go to Vector Choose Rule II; otherwise go to (3)
(3) for each component $x_i' \in \mathbf{x}'$, $i$=1, 2, ...$n$
(4)     if $|x_i'| \neq |x_{n-i+1}'|$ then break
```

(5) if $|x'_i| < |x'_{n-i+1}|$ then return $-\mathbf{x}$ else return $\mathbf{x}$

In Vector Choose Rule I, if Step (2) yields the same result with $\mathbf{x}'$ and $-\mathbf{x}'$ then the next rule (Vector Choose Rule II) is applied.

Vector Choose Rule II:

(1) Construct the canonical codes based on $\mathbf{x}'$ and $-\mathbf{x}'$, respectively, denoted by Code$^+$ and Code$^-$

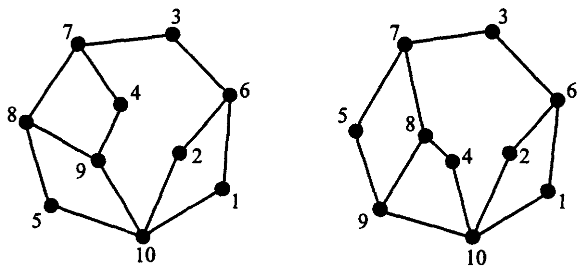
(2) The canonical code (final) is min{Code$^+$, Code$^-$}

Revisit the graphs shown in Figure 4.4. One can find that the ambiguity is resolved after Vector Choose Rule I is applied. The final result, the canonical code of the graph, is 00017026220703A0931909142211A020000.

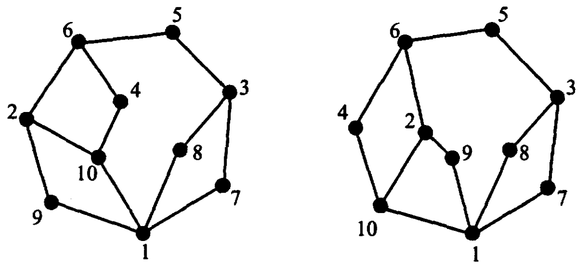
5.3.2 Step 2: Coping with Ambiguity in Labeling

It may be possible that not all graphs have a unique basic expression after only considering the first unique eigenvector. In other words, after Step 1, there is still ambiguity in unique labeling. For instance, the two graphs shown in Figure 5.9a have the same spectrum on their AAMs:

$\Lambda_a=\Lambda_b=\mathrm{diag}(4.2907, 5.1088, 6.2954, 6.8061, 7.0000, 8.0000, 8.3174, 8.9032, 9.2784, 10.0000)$



(a) two graphs



(b) the respective smallest automorphic graphs

Figure 5.9 Two isomorphic graphs and their respective smallest automorphic graphs.

The basic expressions of these two graphs (λ_1=4.2907) are shown in Figure 5.10. Note that the vectors $-\mathbf{x}_1^a$ and $-\mathbf{x}_1^b$ have the same basic expression for graphs A and B, respectively, (see Figure 5.10). It can be seen from Figure 5.10 that two groups exist in the basic expression (after Step 1). In this case, the second unique eigenvector should be considered for relabeling the vertices within these two groups to reach the canonical label. The relabeling gets back to Step 1 but with $\mathbf{x}_2^a\,(\mathbf{x}_2^b)$, and this results in Figure 5.11 for the graphs shown in Figure 5.9a. It can be seen from Figure 5.11 that vertex 6 (-0.6936) should be labeled lower than vertex 4(-0.1773) according to the basic labeling strategy. It should be noted that the relabeling procedure at this time is based on the basic expression which was generated the first time. Therefore relabeling here is just focused on the uncertain part, which are the two shadowed groups shown in Figure 5.10.

A	$\mathbf{x}_1^a$	$-\mathbf{x}_1^a$	$-\mathbf{x}_1'^a$	Relabel *A*	$\leftrightarrow$ *B*	$-\mathbf{x}_1'^b$	$-\mathbf{x}_1^b$	$\mathbf{x}_1^b$	*B*
1	-0.2253	0.2253	-0.6712	10	10	-0.6712	0.2253	-0.2253	1
2	-0.2253	0.2253	-0.3244	8	8	-0.3244	0.2253	-0.2253	2
3	0.0048	-0.0048	-0.1646	4	5	-0.1646	-0.0048	0.0048	3
4	0.1646	-0.1646	-0.1646	6	6	-0.1646	0.2684	-0.2684	4
5	-0.2684	0.2684	-0.0048	3	3	-0.0048	-0.1646	0.1646	5
6	0.1646	-0.1646	0.1822	7	7	0.1822	-0.1646	0.1646	6
7	-0.1822	0.1822	0.2253	1	1	0.2253	0.1822	-0.1822	7
8	0.3244	-0.3244	0.2253	2	2	0.2253	-0.3244	0.3244	8
9	-0.4282	0.4282	0.2684	5	4	0.2684	0.4282	-0.4282	9
10	0.6712	-0.6712	0.4282	9	9	0.4282	-0.6712	0.6712	10

Figure 5.10 The basic expression after sorting the first unique eigenvector.

A	$\mathbf{x}_2^a$	$-\mathbf{x}_2^a$	The 1st relabel	The 2nd *A*	$\leftrightarrow$ *B*	The 1st relabel	$-\mathbf{x}_2^b$	$\mathbf{x}_2^b$	*B*
1	-0.1363	0.1363	10	10	10	10	0.1363	-0.1363	1
2	-0.1363	0.1363	8	8	8	8	0.1363	-0.1363	2
3	-0.2162	0.2162	4	6	6	5	0.2162	-0.2162	3
4	0.1773	-0.1773	6	4	5	6	-0.1134	0.1134	4
5	0.1134	-0.1134	3	3	3	3	-0.1773	0.1773	5
6	0.6936	-0.6936	7	7	7	7	-0.6936	0.6936	6
7	-0.3250	0.3250	1	1	1	1	0.3250	-0.3250	7
8	0.2679	-0.2679	2	2	2	2	-0.2679	0.2679	8
9	0.0228	-0.0228	5	5	4	4	-0.0228	0.0228	9
10	-0.4613	0.4613	9	9	9	9	0.4613	-0.4613	10

Figure 5.11 The basic expression after sorting the second unique eigenvector.

Similarly, vertex 6 of graph *B* is assigned a lower label number than vertex 5 of graph *B* when considering vector $-\mathbf{x}_2^b$. However, at this point, vertices 1 and 2 have produced an ambiguity in determining that labels, because a switch of vertex 1 and vertex 2 does not affect anything. Note that this ambiguity stems from the first unique eigenvector. Thus, one needs to consider the third unique eigenvector and so forth until the last unique eigenvector. Similarly, the next relabeling procedure is based on the basic expression generated so far and only focused on the shadowed group as indicated in Figure 5.11. In this case, there is still a group-to-group mapping in the common mapping. Therefore, the process finds one-to-one mappings according to algorithm III-4 and so on until all possible mappings are examined. For this particular example, the search for one-to-one mappings was coincidently done before when the graph counting problem was discussed; see Figure 5.4.

5.3.3 Step 3: The Smallest Automorphism

After running Step 1 and Step 2 above, the ambiguity in labeling should be resolved. However, it is known that a graph may have automorphisms besides the trivial one (i.e., itself). All automorphic graphs are representatives of that graph. The canonical codes of automorphic graphs, as generated by the above two steps, could be different. A question then arises of which code is chosen as the canonical code of the graph. The idea is to find the 'smallest' automorphism given a graph. The smallest automorphic graph is one of the automorphic graphs whose canonical code is the smallest one. For example, two automorphisms exist for graphs *A* and *B* shown in Figure 5.9a, respectively. Applying the procedure in Step 1 and Step 2, one can find a unique labeling for each of the automorphic graphs. The smallest automorphic graphs are shown in Figure 5.9b. From these, the canonical codes of the two graphs shown in Figure 5.9a are generated, respectively, and they are the same, i.e., 00F13588C000. Therefore, the two graphs are isomorphic. In this case, one can see that the canonical code is also used for the detection of isomorphisms for two graphs.

5.3.4 Summary of Algorithm for Unique Labeling of a Graph

The process of unique labeling of a graph aims to formulate a canonical code representing the graph. This code represents the structural information of a graph uniquely. Therefore, graph isomorphism detection can also benefit this code system because if the codes of two graphs are the same, the two graphs are isomorphic; otherwise, they are not. This point has been demonstrated by bringing together two graphs (whether or not they are isomorphic has been already known) in the above discussion for unique labeling of a graph.

There are three steps to reach a unique labeling and thus a canonical code of a graph. The first step is to sort the unique eigenvector in a predefined order, which leads to the basic expression of a graph. The basic expression may not be unique after the first step, because some components in the eigenvector may be the same, which implies that the labeling for the vertices corresponding to these components is uncertain. The second step is basically the same as the first by seeking the second unique eigenvector, the third, etc., until the labeling for all the vertices are made certain. The third step considers the automorphic graph of the graph under labeling because different automorphic graphs may have different codes. In this step, the unique labeling of automorphic graphs is considered, which makes use of the procedure of Step 1 and Step 2 to find the smallest automorphic graph. The final canonical code corresponds to the smallest automorphic graph.

It is noted that the canonical label for a digraph or a weighted graph (undirected or directed) can also be created with these steps. For the case of digraph (weighted or not), instead of concatenating the upper-right triangular part of the rows of the (adjusted) adjacency matrix of the graph, concatenating the whole rows of AAM is necessary to formulate the canonical code of a digraph. The weighted undirected graph follows the same procedure as the non-weighted undirected graph.

5.3.5 Examples

Figure 5.12 shows three graphs. Each graph has 12 vertices. Through the Eigensystem approach, the canonical codes of these graphs are formed, i.e., 00582A261955A0000, 003816152C65C0000, and 00582A261955A0000, respectively. Since the canonical codes of the first and third graphs are the same but differ from the canonical code of the second graph, the first and third graphs are isomorphic to each other, but they are not isomorphic to the second graph.

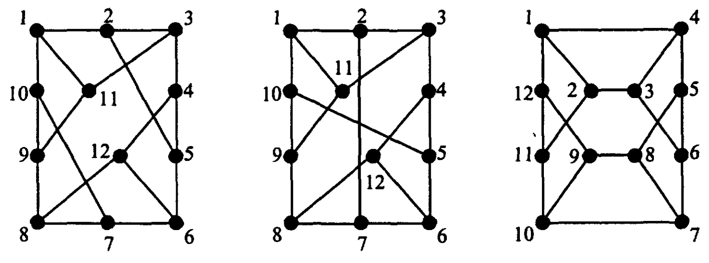


Figure 5.12 Three graphs with 12 vertices.

Figure 5.13 shows 7 graphs each with 10 vertices. The Eigensystem approach gives the following conclusion for these graphs: each of these graphs has 120 automorphisms and has the same canonical code, i.e., 00729A254920. Hence, these graphs are isomorphic to each other.

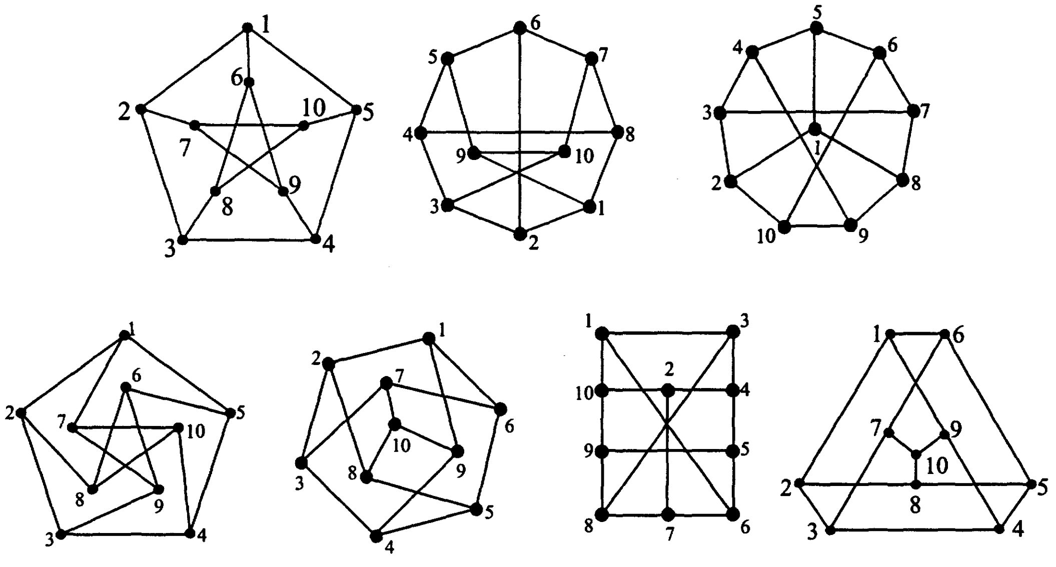


Figure 5.13 Seven isomorphic graphs each with 10 vertices.

Table 5.5 lists the canonical codes of the graphs that were previously discussed as the examples. A note is given on the table to indicate whether the graphs are isomorphic.

Table 5.5 The canonical codes of the graphs discussed previously as the examples.

Figure	**Canonical Code**	**Note**
2.5	(a) 00E6CEA0, (b) 00E6CEA0	Isomorphic graphs
2.15	(a) 03F08000, (b) 01E08440	Non-isomorphic graphs
4.1	(a) 08E5F780, (b) 01EACE70	Non-isomorphic graphs

Table 5.5 (*Continued*)

4.2	(a) 0038162C1128C0800, (a) 003886192A6820000	Non-isomorphic graphs
4.4	(a) 00017026220703A0931909142211A020000, (b) 00017026220703A0931909142211A020000	Isomorphic graphs
4.7	(a) 007092255200, (b) 007092255200	Isomorphic graphs
4.9	(a) 07DEEC, (b) 07DEEC	Isomorphic graphs
4.10	(a) 1861CC330C0E020040400003458, (b) 1861CC330C0E0200404000001D0A	Non-isomorphic graphs
4.12	(a) 00D0E068C800, (b) 00703384C800	Non-isomorphic graphs
4.13	(a) 00F158312080, (b) 00F1C2308880	Non-isomorphic graphs

4.14	(a) 0070B2293000, (b) 01E030A4C400;
	(c) 0073211000, (d) 007020AA00;
	(e) 7C21, (f) 1EC1;
	(g) 0DCCE8, (h) 07D22B, (i) 1F8CE0;
	(j) 00F2718000, (k) 00F2F08000;
	(l) 00F158312080, (m) 00F1C2308880
		Non-isomorphic graphs

5.1	(b) 000008880005040001280002440010820021100105002220080182018206020900600600000000000000000000000 (c) 000008880005040001280002440010820021100105002220080182018206020900600600000000000000000000000	Isomorphic graphs
5.5 5.7	0604A0A824101020806484C, 0604A0A824101020806484C	Isomorphic graphs
5.9	(a) 00F13588C000, (b) 00F13588C000	Isomorphic graphs

5.4 Enumeration of Structures

The third fundamental problem in structure synthesis is the enumeration of structures. Since the semantics of structures can be represented by graphs or networks, the enumeration of structures becomes the enumeration of graphs. As discussed in Chapter 1, the nature of enumeration is to find all non-isomorphic structures given a set of conditions or constraints. The constraints must come from applications. The two key techniques for the enumeration of structures are now (1) detection of isomorphic graphs and (2) conversion of constraints defined on the structure from a particular application domain into constraints on a graph. The constraints on a graph could be, for example, the total number of vertices of a graph, the degrees of vertices, etc. The current strategy taken in the Eigensystem approach is to count all graphs meeting the constraints and then remove the isomorphic ones.

5.4.1 Graphs Meeting Constraints

The two basic constraints on the vertex of a graph are (1) the total number of the vertices and (2) the degree of each vertex. Extra constraints can be added depending on the application problem under investigation. A method is developed for enumerating all graphs subject to the two types of constraints as mentioned.

This method includes the following basic points. *First*, divide the vertices into different groups V_i (i=1, 2, ...m) according to their degrees, where m is the total number of such groups. Each vertex in group V_i has the same degree d_i. Let n_i be the total number of the vertices in each V_i and let c_i be the current capacity of each vertex in V_i to be connected in the constructed graph. It is clear that $c_i = d_i$ at an initial time, and at any time

$$\sum_{i=1}^{m} n_i = N \tag{5.9}$$

and

$$T_i = n_i c_i \tag{5.10}$$

where N is the total number of vertices in a graph and T_i is the total capacity of the vertices in V_i, which could be connected to the constructed graph. T_i is called the number of tokens in V_i.

Second, the graph construction runs until one of the following conditions is satisfied: (1) no group exists, (2) no token can be offered, and (3) no vertex can accept any token. When the construction process stops and all the above conditions are satisfied, this indicates a valid configuration or graph; otherwise, the construction process goes back to the step where the present process is initiated.

Third, the so-called valid structure is defined by two integrity rules. *Integrity Rule 1*: for any vertex in a vertex group, only one token can be offered at any time. *Integrity Rule 2*: a valid structure is the one that satisfies the three conditions mentioned before simultaneously.

The sketch of the algorithm for the graph construction process is described as follows:

Step 1: Take out a vertex $u \in V_i$ (i=[1, m]), where c_i in V_i is the highest among those in other vertex groups. Give vertex u a label l where l is a number ordered from 1 to N. Note that now the number of the vertices in V_i changes to n_i-1, and the number of the tokens in V_i changes to T_i-c_i. Since u has a capacity of c_i to connect with other vertices, c_i tokens need to be taken from vertex groups (including its own vertex group) to connect with vertex u. There are many possible ways to draw tokens, and they manifest different graphs.

Step 2: Once a vertex $v \in V_j$ provides a token to vertex u, it has to be taken out from V_j and is assigned a label l+1. A connection (i.e., an edge) between v and u is established and represented by the adjacency matrix. At this time, the resource (i.e., n_j and T_j) in V_j has to be changed. If V_j is empty after vertex v is left, remove V_j from the group list and decrease m by one. If v still has capacity after a token of it is contributed to vertex u, i.e., c_j>0, create a new vertex group V_{m+1}, put v as well as its current capacity c_j into V_{m+1}, and increase m by one. Likewise, all vertices which provide tokens are dealt with.

Step 3: After Step 2, all vertices connected with vertex u are found and labeled. Some of them still have capacities and thus are put into new vertex groups (but they are labeled). Therefore, two kinds of vertex groups exist: labeled vertex groups and unlabeled vertex groups (the vertex groups have not yet been connected in the constructed graph). Take out a vertex u from the labeled vertex group having the smallest label. Vertex u has capacity and thus tokens are needed from other vertex groups (both labeled and unlabeled). There are many possible ways to draw tokens, and they manifest different graphs. The construction procedure goes back to Step 2.

As an example, if a graph is subject to the following constraints: (1) the total number of vertices is 8, (2) 4 vertices have 2 degrees, and (3) the other 4 vertices have 3 degrees, the method described above gives 108 graphs meeting these constraints. This method is also used for the application cases that will be introduced in Chapters 6 and 7 where some more constraints (derived application) are added in addition to the two constraints mentioned above. The method described above needs to be extended accordingly to deal with the extra constraints.

5.4.2 Distinct Graph Enumeration

The graphs created by the proposed method may have isomorphic graphs. Distinct graph enumeration requires to remove these isomorphic graphs. At this time, the canonical labeling of graphs is applied for a graph isomorphism test.

Continue the discussion of the example previously mentioned. In this example, the graph has 8 vertices, in which 4 vertices have 2 degrees and the other 4 vertices have 3 degrees. It is known that there are 108 graphs that meet the constraints. Applying the canonical labeling algorithm to these graphs leads to 25 distinct canonical codes, which imply that 25 graphs are distinct among 108 graphs. These 25 distinct graphs with their canonical codes are shown in Figure 5.14.

It may be clear by now that the enumeration of structures is a task depending on applications. The next two chapters are devoted to the demonstration of the structure enumeration for two applications: machine design and molecule design, respectively.

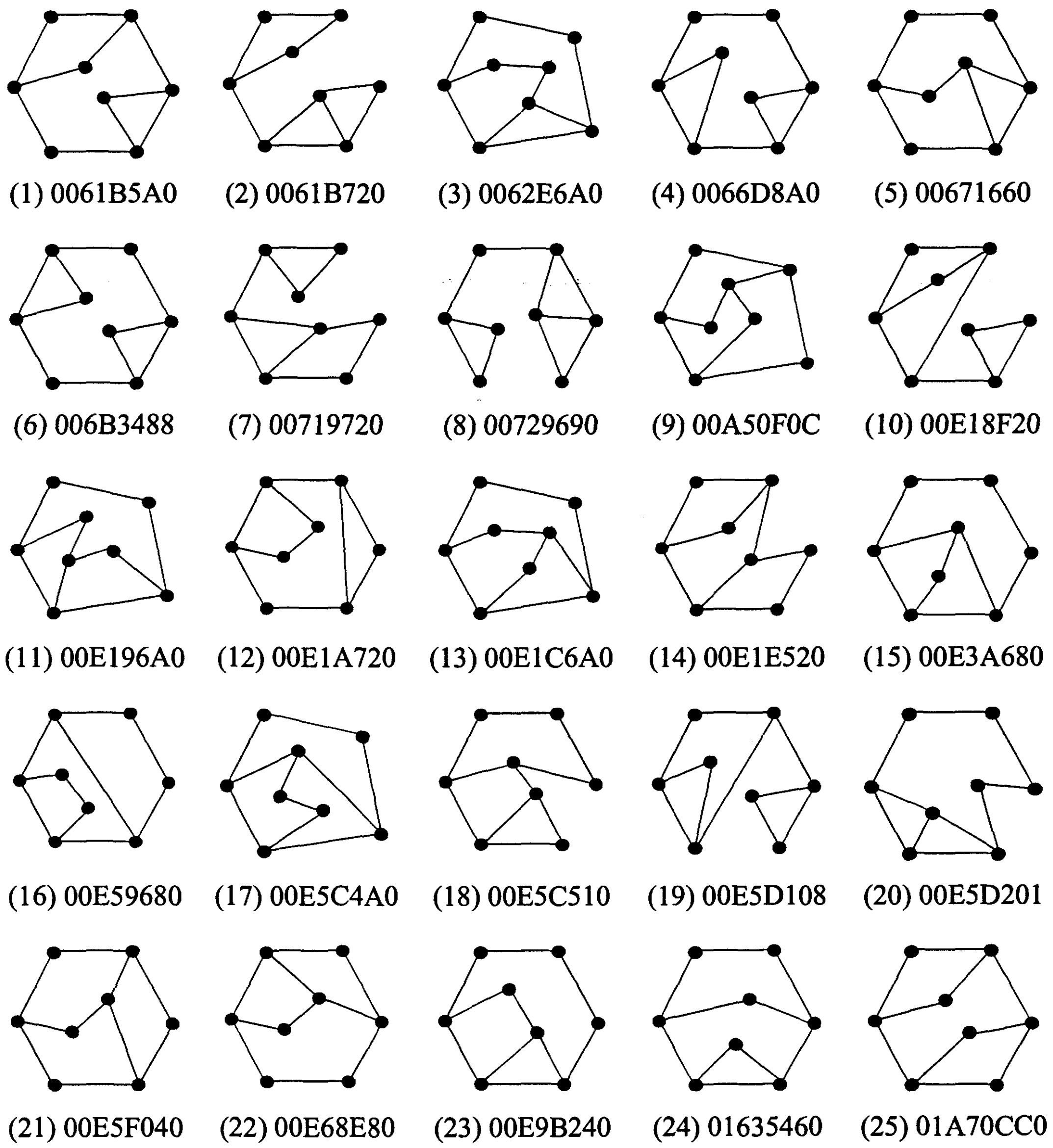


Figure 5.14 Enumeration of 8-vertex graphs (4 vertices for 3 degrees and other 4 for 2).

5.5 Concluding Remarks

The Eigensystem approach has been further extended to solving the graph counting problem and the graph canonical labeling or coding problem. These two problems are the basis for developing algorithms for the structure/graph enumeration, which is the third fundamental problem as mentioned in Chapter 1. One of the main purposes of the canonical labeling of a graph is to have a compact code which uniquely represents the graph and all graphs isomorphic to it. Therefore, the computer storage and retrieval of graphs are very efficient. It is important to note that the Eigensystem approach has provided a complete solution to graph isomorphism, graph counting, and graph labeling or coding in the sense that the solution is applicable to (1) non-weighted undirected graphs, (2) weighted undirected graphs, (3) non-weighted directed graphs, and (4) weighted directed graphs. The program Nauty has not shown the solution to the graph counting and graph labeling problems for the weighted graphs. The method with its algorithm developed in this thesis study to enumerate all graphs to the constraints is novel; however, a more general method may be developed along with the constraint satisfaction paradigm [Tsang 1993]. This is because the issue addressed here is conceptually a constraint satisfaction problem. There is a wealth of solving tools available to find all the solutions to a set of predefined constraints.

The implementation of the algorithm for canonical labeling of graphs can be seen in Appendix B.

CHAPTER 6
APPLICATION I: SYNTHESIS IN MECHANICAL DESIGN

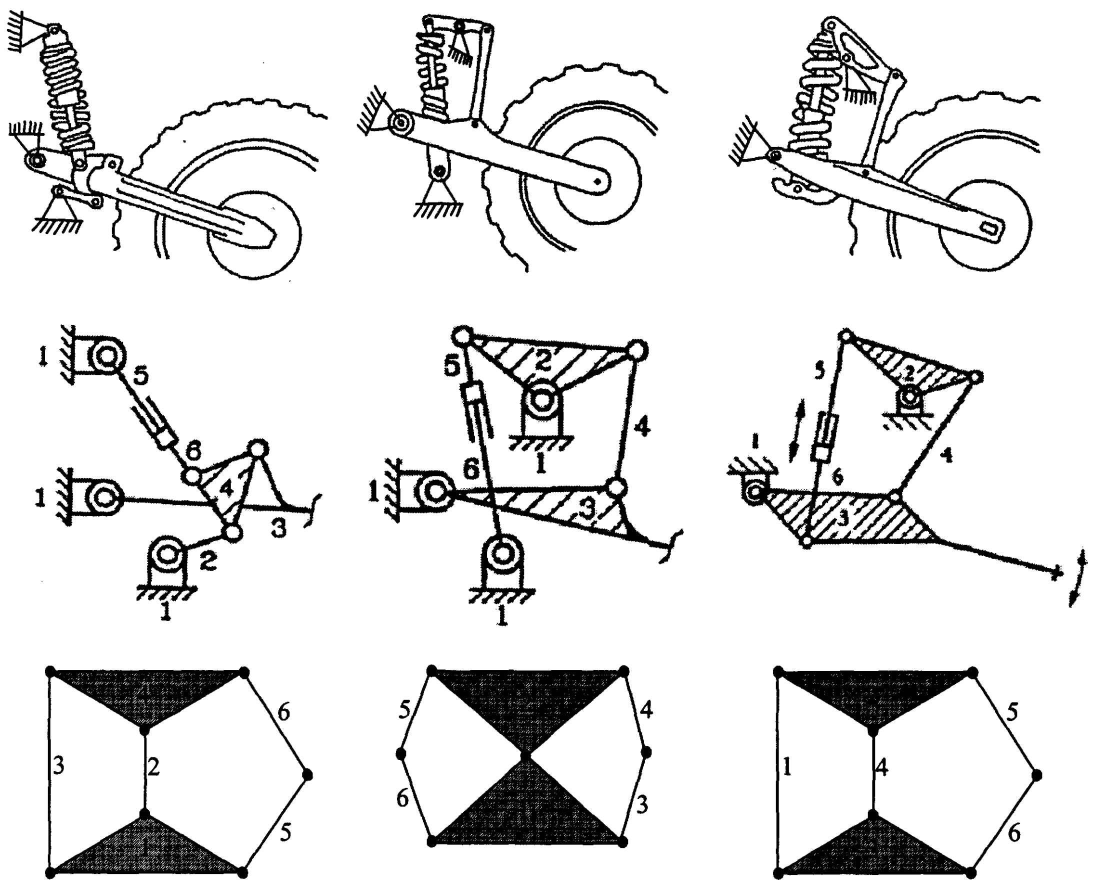
6.1 Introduction

At several early phases for designing machines or mechanisms, decisions must be made as to where the parts must be placed, how far and in what directions the parts must move, which parts must be connected to which other parts, and how they must be connected, and what the critical dimensions of the parts must be. Obviously, this phase involves the interactions between geometry and motions, and it will result in drawings or sketches of the general layout of the machine and will indicate how it will operate. These design tasks (the studies of position, displacement, rotation, speed, velocity, and acceleration) are referred to as the *kinematic design* of the machine [Uicker *et al.* 2003]. *Kinematics* is the base of any design which concerns motion and is further divided into two complementary fields: kinematic analysis and kinematic synthesis.

Kinematic analysis refers to the analysis of kinematic motion behavior, e.g., given the motion of one link, find the motion of other links. *Kinematic synthesis* involves three steps: type synthesis, number synthesis, and dimensional synthesis. *Type synthesis* refers to the selection of types of mechanisms: a linkage, a geared system, belts and pulleys, or even a cam system. *Number synthesis* deals with the number of *links* (parts of a mechanism) and the number of *joints* or *kinematic pairs* or just *pairs* (the connections, joint between the links) that are required to obtain a certain mobility. It relates to the problem of the enumeration of structures, which is concerned in this thesis. *Dimensional synthesis* refers to the determination of the dimensions of the individual links.

Highly simplified schematic diagrams and kinematic chains are used for describing a machine or mechanism when studying its kinematics. For examples, Figure 6.1 shows three suspension topologies used in automobiles, their schematic diagrams, and their related kinematic chains, respectively. Figure 6.1 also shows that there are two results of a number synthesis for a six bar and one degree of freedom linkages, i.e., the six bars shown in each of Figures 6.1a and 6.1b, because the one shown in Figure 6.1c is the same as the one in Figure 6.1a (to be revisited later).

Kinematic synthesis is an important topic in the design, in particular in the creative design, of a machine [Johnson 1978]. This chapter discusses the application of the Eigensystem approach for mechanical design synthesis, in particular for number synthesis. Section 6.2 gives the concepts on mechanisms/kinematic chains. Section 6.3 reviews others' studies on the enumeration of mechanisms. Section 6.4 presents the Eigensystem approach, which was described in Chapter 5, as applied to the enumeration of structures (mechanisms/kinematic chains). Finally, Section 6.5 gives a concluding remark.



(a) Honda CR250 Pro-link (b) Kawasaki KX250 Uni-trak (c) Suzuki RM250 Full-floater

Figure 6.1 Three suspension topologies and their kinematic diagrams [Yan and Chen 1985].

6.2 Basic Concepts

When several links are movably connected together by joints, they are said to form a *kinematic chain*. Links containing only two pair element connections are called *binary* links; those having three, four, and five are called *ternary*, *quaternary*, and *pentagonal* links, respectively. If every link in the chain is connected to at least two other links, the chain forms one or more closed loops, and is called a *closed* kinematic chain; otherwise, the chain is referred to as an *open* kinematic chain.

Prior to kinematic synthesis, the number of *degrees of freedom* (DOF) of the mechanism must be determined. When a link moves freely in a plane, it has three DOFs: the freedom to translate along two independent directions and the freedom to rotate about an axis perpendicular to the link. When two links are connected by a turning pair, the pair provides a constraint of order 2 and thus these two links have four DOFs. For a mechanism with the ground considered as one of the links, the relation between F (degrees of freedom), N (number of links) and J (number of joints) is given by Grubler's equation [Uicker *et al.* 2003]

$$F = 3(N-1) - 2J \tag{6.1}$$

Furthermore, let p be the highest connectivity of the link that can be used to form a chain and n_i be the number of links with connectivity i, $2 \leq i \leq p$. Two constraint equations exist among links and joints, i.e.,

$$n_2 + \cdots + n_i + \cdots + n_p = N \tag{6.2}$$

and

$$2n_2 + \cdots + in_i + \cdots + pn_p = 2J \tag{6.3}$$

For a given N and F, the highest connectivity p is easily determined using the following formula [Rao and Deshmukh 2001]

$$p = \begin{cases} \dfrac{N-F+1}{2} & ;F = 0,1 \\ \min\left(N-F-1, \dfrac{N+F-1}{2}\right) & ;F \geq 2 \end{cases} \tag{6.4}$$

Once p is known by Equation (6.4) for a given N and F, all n_i can be obtained by solving the linear equations from Equation (6.1) to Equation (6.3). For example, if $F = 1$ and $N = 6$, by Equation (6.4), the highest connectivity $p = 3$. According to Equation (6.1) through (6.3), one combination of links is obtained, i.e., (n_3=2, n_2=4). If $F = 3$ and $N = 8$, then $p = 4$ by Equation (6.4). There are two combinations of links n_i ($2 \leq i \leq p$) which meet Equations (6.1), (6.2), and (6.3). They are (n_4=0, n_3=2, n_2=6) and (n_4=1, n_3=0, n_2=7), respectively. Table 6.1 lists the number of combinations of links for a given N up to 12 and F up to 9.

Table 6.1 Number of link combinations for a given N up to 12 and F up to 9.

N	**Degrees of freedom (DOF)**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
4	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
5	0	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a
6	1	1	1	n/a	n/a	n/a	n/a	n/a	n/a
7	1	2	1	1	n/a	n/a	n/a	n/a	n/a
8	3	2	2	1	1	n/a	n/a	n/a	n/a
9	3	4	3	2	1	1	n/a	n/a	n/a
10	7	5	5	3	2	1	1	n/a	n/a
11	8	9	6	5	3	2	1	1	n/a
12	15	11	10	7	5	3	2	1	1

Number synthesis of kinematics enumerates all the distinct possible kinematic chains with the specified number of links and degrees of freedom. In general, this includes three tasks: (1) for a given N and F, determining the highest connectivity p and the possible combinations of n_i ($2 \leq i \leq p$), as discussed above, (2) enumerating all possible kinematic chains for each combination of links, and (3) identifying distinct kinematic chains from them.

6.3 Others' Studies

Many studies were reported in the past using different approaches to enumerate all the distinct kinematic chains with the given number of links and degrees of freedom. These methods reported so far are based on intuition [Crossley 1965, 1966], Franke's notation [Davies and Crossley 1966; Haas and Crossley 1969; Soni 1971], graph theory [Crossley 1965; Dobrjanskyj and Freudenstein 1967; Woo 1967; Freudenstein 1967], and transformation of binary chains [Manolescu 1973; Mruthyunjaya 1979]. Mruthyunjaya [1984] developed a computer program based on the method of transformation of binary chains for the structural synthesis of kinematic chains with up 10 links and 3 DOFs. Butcher and Hartman [2002] have enumerated 6856 single DOF chains with 12 links. Sohn and Freudenstein [1986] used the concept of dual graphs and generated chains with up to 11 links and two DOFs. Tuttle *et al.* [1989] used the theory of symmetric groups to generate planar kinematic chains by performing contraction and expansion operations on a base structure and obtained 2 DOFs chains with 12 links. Hwang and Hwang [1992] have reported a computer-aided method to generate planar kinematic chains using the concept of contracted link adjacency matrix. Rao and Raju [1991] reported the Hamming number technique for the synthesis of chains, but the results still need to be tested for isomorphism. Tischler *et al.* [1995] presented the method for generating the chains with the idea of avoiding tests for isomorphism but concluded that it may not be possible to synthesize distinct chains without testing for isomorphism. This situation occurred in the report of [Rao and Deshmukh 2001], where they used the concept of loop formation to construct kinematic chains without isomorphism. However, this method still cannot guarantee that no isomorphic chain exists in the results generated.

By collecting the literature, Table 6.2 lists the number of distinct kinematic chains, which is known as a correct record, with up to 12 links and 3 DOF. It is noted that these kinematic chains consider the same types of links and joints.

Table 6.2 The number of distinct kinematic chains with up to 12 links for known cases.

Number of links	DOF=1	DOF=2	DOF=3
4	1		
5		1	
6	2		1
7		4	
8	16		
9		40	
10	230		98
11			
12	6856		

6.4 The Enumeration of Kinematic Chains

6.4.1 Graph Representation

The Eigensystem approach can be used for structure synthesis of mechanisms/kinematic chains. The canonical codes of graphs can identify distinct graphs without an isomorphism test. Prior to enumerating kinematic chains, however, the schematic diagrams of kinematic chains must be converted into graph representations (see Figure 6.1). The conversion between a schematic diagram and a graph representation is done by replacing each link in the schematic diagram with a vertex in a graph representation and replacing each joint between two links in the schematic diagram with an edge between two vertices in the graph. Figure 6.2 shows the graph representations of the kinematic chains shown in Figure 6.1. Note that the graph representation shown in Figure 6.2 is different from the graph representation shown at the bottom of Figure 6.1. Figure 6.3 shows other six kinematic chains and their graph representations.

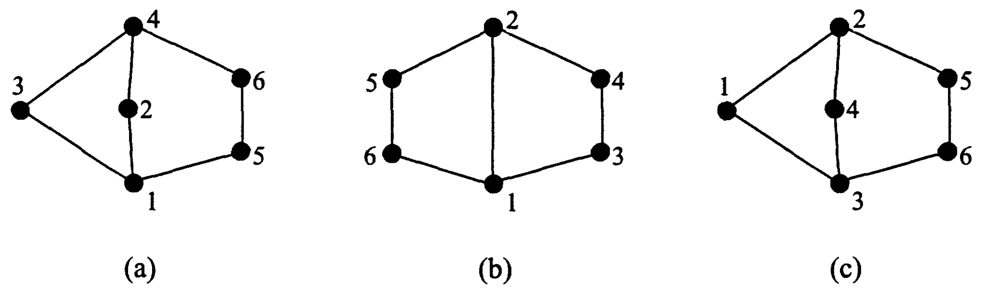


Figure 6.2 The graph representations of the kinematic chains shown in Figure 6.1.

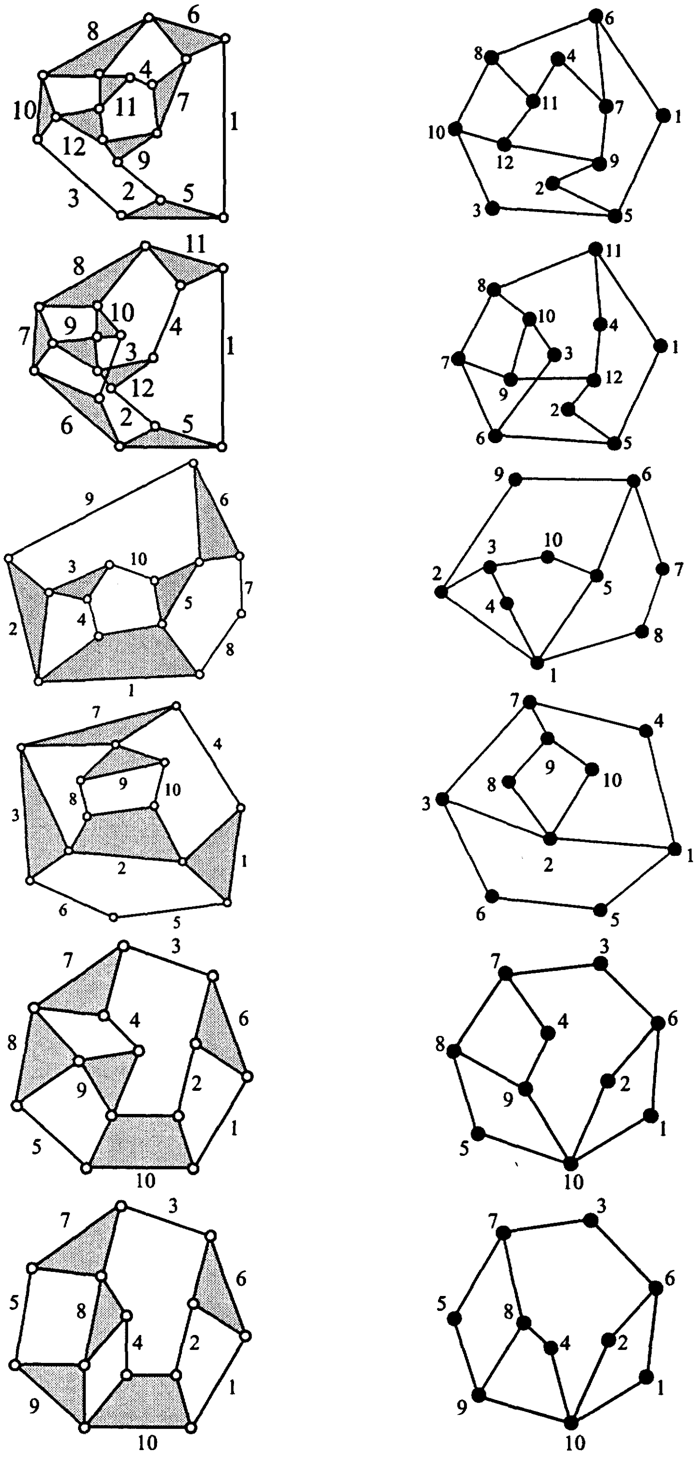


Figure 6.3 Six kinematic chains and their graph representations.

6.4.2 Enumeration of Kinematic Chains: Step 1

After the combinations of links are obtained for a given link number *N* and degrees of freedom *F*, the next objective is to enumerate kinematic chains. Two steps are followed here for this objective. The first step is to enumerate kinematic chains meeting a set of constraints. The second step is to remove isomorphic ones from the kinematic chains meeting the constraints.

To enumerate kinematic chains meeting the given constraints is to enumerate all possible kinematic chains for each combination of links. There are two types of information included in a combination of links: the number of links and the degrees of freedom for each link. When kinematic chains are converted into graph representations, these two types of information are changed into the number of vertices and the degrees of every vertex because links and DOFs of links in kinematic chains are converted into vertices and degrees of vertices in graph representations, respectively, as discussed in Section 6.4.1. A general method was developed and discussed in Chapter 5 to enumerate graphs meeting these constraints, i.e., a given number of vertices and degrees of each vertex. Therefore, the method can be applied here for enumerating the kinematic chains meeting a given combination of links. Table 6.3 lists the numbers of these kinematic chains for a given *N* up to 12 and *F* up to 9.

However, one must notice that among kinematic chains generated based on the above step, there may be cases in which the overall DOF of the kinematic chain is greater than zero, but some of the features have DOF ≤ 0. An example of this kind of case is shown in

Figure 6.4, where the links marked by a cycle form a zero DOF chain. For this reason, the kinematic chains listed in Table 6.3 are called the incomplete kinematic chains. The kinematic chains shown in Figure 6.4 are incomplete, and they are actually degraded into four links with one DOF. This thus calls for the elimination of such degraded cases. The solution is to define new constraints and to incorporate them into the general pool of constraints. In particular, the following constraint is established, i.e., any closed loop or combination of loops created during the construction of a graph must greater than zero DOF. This constraint is incorporated into the general method developed in Chapter 5.

Table 6.3 Number of kinematic chains meeting a given *N* up to 12 and *F* up to 9.

N	**Degrees of freedom (DOF)**								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
4	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
5	0	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a
6	7	0	1	n/a	n/a	n/a	n/a	n/a	n/a
7	0	20	0	1	n/a	n/a	n/a	n/a	n/a
8	222	0	34	0	1	n/a	n/a	n/a	n/a
9	0	681	0	52	0	1	n/a	n/a	n/a
10	9863	0	1565	0	75	0	1	n/a	n/a
11	0	31235	0	3184	0	103	0	1	n/a
12	609582	0	82243	0	6030	0	136	0	1

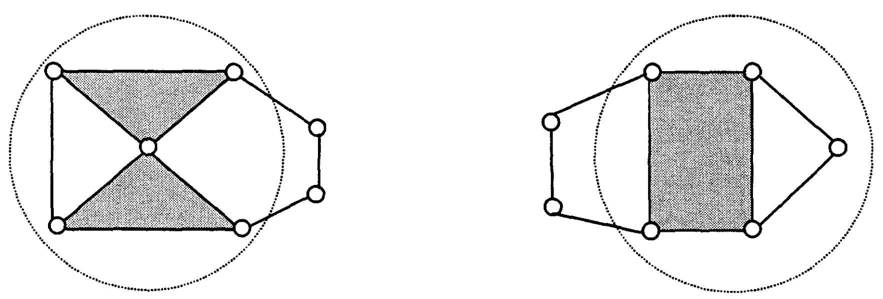


Figure 6.4 Two six-link kinematic chains each with totally 1 DOF but partly 0 DOF.

Figure 6.5 graphically illustrates how this constraint works for a kinematic chain with four binary links and four ternary links (links and joints correspond to vertices and edges, respectively). In Figure 6.5, the label in a cycle represents the label of the vertex, and the label on an edge represents the step of a graph construction process. In Figure 6.5a, vertex 2 accepts a token from vertex 5 at the fourth step and then accepts a token from vertex 3 at the fifth step. Once vertex 2 accepts this token from vertex 3, a closed loop is formed among vertices 1, 2, and 3. The sub-chain corresponding to this closed loop has zero DOF. Therefore, such a configuration is eliminated by the program. In Figure 6.5b, when vertex 3 accepts a token from vertex 5 at the fifth step, one closed loop is formed among vertices 1, 2, 3, and 5, and the corresponding sub-chain has one DOF. The construction procedure continues until the seventh step where vertex 4 accepts a token from vertex 5. In total, three loops are obtained, i.e., loop (1, 2, 4, 5), loop (1, 2, 3, 5), and loop (1, 3, 4, 5). It is interesting to note that a compound loop (1, 2, 3, 4, 5) formed from these three loops has zero DOF. Such a configuration (i.e., the compound loop with the zero DOF) also has to be eliminated. Figure 6.5c shows closed loops and that their compound loops formed at any step have greater than zero DOF. Therefore, such a configuration is accessible.

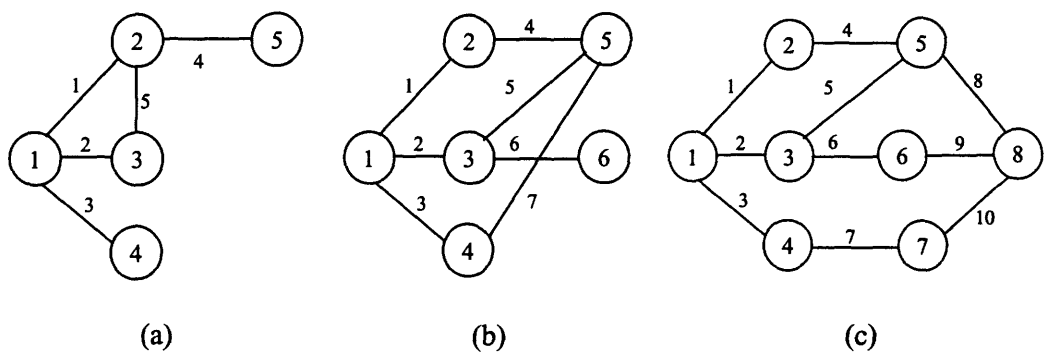


Figure 6.5 Illustration of configuring kinematic chains.

Table 6.4 lists the numbers of kinematic chains with up to 12 links and up to 9 DOFs using the program developed with this thesis study. The program system which finds all kinematic chains meeting the constraints is called 'E_mechanism' (see Appendix C).

Table 6.4 Number of possible kinematic chains with up to 12 links and 9 DOFs.

N	Degrees of freedom (DOF)								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
4	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
5	0	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a
6	4	0	1	n/a	n/a	n/a	n/a	n/a	n/a
7	0	10	0	1	n/a	n/a	n/a	n/a	n/a
8	63	0	20	0	1	n/a	n/a	n/a	n/a
9	0	232	0	32	0	1	n/a	n/a	n/a
10	1775	0	668	0	51	0	1	n/a	n/a
11	0	7885	0	1534	0	74	0	1	n/a
12	75440	0	26637	0	3210	0	102	0	1

6.4.3 Distinct Kinematic Chains: Step 2

The kinematic chains generated in Table 6.4 may include isomorphs, which must be removed. At this point, the Eigensystem approach is employed for this purpose. The procedure is such that the canonical codes for all the kinematic chains listed in Table 6.4 are found. Among the kinematic chains with the same canonical code, only one is kept. Table 6.5 lists the total number of distinct kinematic chains with up to 12 links and up to 9 DOFs. It is noted that the numbers marked with a star are the same as the numbers shown in Table 6.1. The others listed in Table 6.4 are first reported to the author's knowledge. The result obtained thus is remarkable in having proved the validity of the Eigensystem approach for the enumeration of kinematic chains or mechanisms.

Table 6.5 Number of all distinct kinematic chains with up to 12 links and 9 DOFs.

N	DOF	$n_{7\text{-}6\text{-}5\text{-}4\text{-}3\text{-}2}$	Number	Σ	N	DOF	$n_{7\text{-}6\text{-}5\text{-}4\text{-}3\text{-}2}$	Number	Σ
4	1	0-0-0-0-0-4	1	1^*	9	2	0-0-0-0-4-5	19	40^*
5	2	0-0-0-0-0-5	1	1^*			0-0-0-1-2-6	16	
6	1	0-0-0-0-2-4	2	2^*			0-0-0-2-0-7	3	
	3	0-0-0-0-0-6	1	1^*			0-0-1-0-1-7	2	
7	2	0-0-0-0-2-5	3	4^*		4	0-0-0-0-2-7	8	10
		0-0-0-1-0-6	1				0-0-0-1-0-8	2	
	4	0-0-0-0-0-7	1	1		6	0-0-0-0-0-9	1	1
8	1	0-0-0-0-4-4	9	16^*	10	1	0-0-0-0-6-4	50	230^*
		0-0-0-1-2-5	5				0-0-0-1-4-5	95	
		0-0-0-2-0-6	2				0-0-0-2-2-6	57	
	3	0-0-0-0-2-6	6	7			0-0-0-3-0-7	3	
		0-0-0-1-0-7	1				0-0-1-0-3-6	15	
	5	0-0-0-0-0-8	1	1			0-0-1-1-1-7	8	

Table 6.5 (*Continued*)

10	1	0-0-2-0-0-8	2	
	3	0-0-0-0-4-6	46	98*
		0-0-0-1-2-7	38	
		0-0-0-2-0-8	8	
		0-0-1-0-1-8	5	
		0-1-0-0-0-9	1	
	5	0-0-0-0-2-8	12	14
		0-0-0-1-0-9	2	
	7	0-0-0-0-0-10	1	1
11	2	0-0-0-0-6-5	153	839
		0-0-0-1-4-6	359	
		0-0-0-2-2-7	193	
		0-0-0-3-0-8	13	
		0-0-1-0-3-7	74	
		0-0-1-1-1-8	34	
		0-0-2-0-0-9	3	
		0-1-0-0-2-8	8	
		0-1-0-1-0-9	2	
	4	0-0-0-0-4-7	89	190
		0-0-0-1-2-8	77	
		0-0-0-2-0-9	12	
		0-0-1-0-1-9	11	
		0-1-0-0-0-10	1	
	6	0-0-0-0-2-9	16	19
		0-0-0-1-0-10	3	
	8	0-0-0-0-0-11	1	1
12	1	0-0-0-0-8-4	410	6856*
		0-0-0-1-6-5	1873	
		0-0-0-2-4-6	2339	
		0-0-0-3-2-7	648	
12	1	0-0-0-4-0-8	37	
		0-0-1-0-5-6	506	
		0-0-1-1-3-7	716	
		0-0-1-2-1-8	147	
		0-0-2-0-2-8	63	
		0-0-2-1-0-9	7	
		0-1-0-0-4-7	49	
		0-1-0-1-2-8	46	
		0-1-0-2-0-9	5	
		0-1-1-0-1-9	8	
		0-2-0-0-0-10	2	
	3	0-0-0-0-6-6	463	2424
		0-0-0-1-4-7	1029	
		0-0-0-2-2-8	530	
		0-0-0-3-0-9	32	
		0-0-1-0-3-8	226	
		0-0-1-1-1-9	102	
		0-0-2-0-0-10	8	
		0-1-0-0-2-9	27	
		0-1-0-1-0-10	5	
		1-0-0-0-1-10	2	
	5	0-0-0-0-4-8	171	354
		0-0-0-1-2-9	141	
		0-0-0-2-0-10	21	
		0-0-1-0-1-10	19	
		0-1-0-0-0-11	2	
	7	0-0-0-0-2-10	21	24
		0-0-0-1-0-11	3	
	9	0-0-0-0-0-12	1	1

6.5 Concluding Remarks

It must be pointed out that in the practical design of mechanisms, structure synthesis may not be only bar linkage systems but also other structural schemes, such as spring mechanisms, belt-pulley mechanisms, cam-linkage mechanisms, gear-linkage mechanisms, chain-sprocket mechanisms, and hydraulic piston-cylinder mechanisms. Besides, they can be connected into a planar system or a spatial system. All of these mechanisms can be represented into a graph by 'coloring' its edges and vertices (although it was not mentioned before, the Eigensystem approach can handle the situation when both edges and vertices are weighted). For example, in Figure 6.1, the kinematic pair between link 5 and link 6 differs from the other kinematic pairs. In order to represent this difference, a different weight can be applied onto the edges between vertices 5 and 6 of the graphs, as shown in Figure 6.2. Therefore, a weighted graph is created. The enumeration of such kinematic chains becomes the enumeration of a weighted graph. As discussed in Chapter 4, the Eigensystem approach can also deal with the isomorphism problems of weighted and directed graphs. Therefore, the Eigensystem approach can efficiently solve this kind of problem of the enumeration of structures. Evidently, the Eigensystem approach is applicable to the enumeration of all four structures/graphs: (1) non-weighted undirected graph, (2) weighted undirected graph, (3) non-weighted digraph, and (4) weighted digraph.

CHAPTER 7
APPLICATION II: SYNTHESIS IN MOLECULE DESIGN

7.1 Introduction

Synthesis of molecular structures plays a very important role in predicting molecular properties, discovering new materials, and designing novel drugs [Basak and Niemi 1990; Gute and Basak 2001]. Synthesis of molecular structures is also a necessary tool for chemical documentation, which refers to the unambiguous naming and the indexed system for chemical substances (single molecules and molecule complexes). The application of the Eigensystem approach for synthesis in molecular design is discussed in this chapter. Section 7.2 gives the basic concepts of molecular structures and their representations. Section 7.3 discusses the isomer enumeration of the Alkane Series. Section 7.4 gives a concluding remark.

7.2 Basic Concepts

7.2.1 Chemical Graph

A molecule is a combination of atoms held together by valence bonds and is represented by a chemical constitutional formula. The constitutional formula for the chemist is a special kind of graph called the *chemical graph*, which provides a representation of the topological structure of a species, with the vertices representing the individual atoms and the edges representing the valence bonds between pairs of atoms. Figure 7.1 illustrates a chemical graph. Figure 7.1a shows a chemical constitutional formula of a compound that consists of carbon and hydrogen atoms. Figure 7.1b illustrates its chemical graph, where closed vertices represent carbon atoms, and open vertices represent hydrogen atoms. Every carbon atom must be tetravalent (a chemical valence of four), while every hydrogen atom is univalent. All closed vertices must therefore be of valence 4 in the graph, and all open vertices must be of valence 1. Sometimes, hydrogen atoms are omitted in a chemical graph for simplicity. Figure 7.1c shows the chemical graph with the hydrogen atoms omitted.

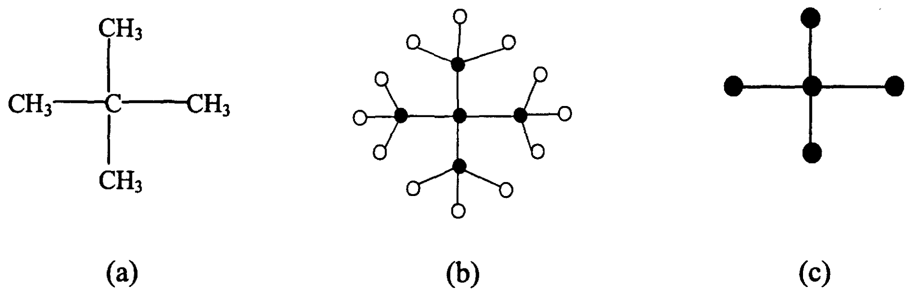
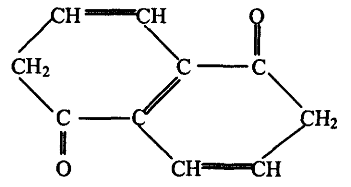
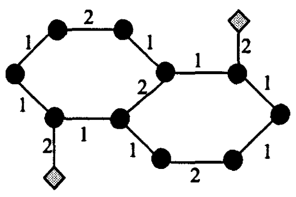


Figure 7.1 Chemical formula, graph, and the graph with omitted hydrogen atoms.

However, in many situations, the chemical graphs have to be ‘colored’ because there are sometimes of various kinds of atoms and more than one valence bond between two atoms. Different weights can be used to describe the different colors. Figure 7.2 shows two chemical constitutional formulas and the chemical graphs, where the number of valence bonds is used as weights of edges and the different shapes (weights) are used for representing different atoms (hydrogen atoms are omitted).



(a) Chemical constitutional formula



(b) Chemical graph

Figure 7.2 Chemical constitutional formula and 'colored' chemical graph.

Although in most cases it is sufficient to know only the constitution of the molecule [Bohanec and Perdih 1993], it should be noted that sometimes it is necessary to know the geometry of the structures [Wilson and Beineke 1979]. In this situation, 'coloring' the chemical graph would be a very complicated task.

7.2.2 Chemical Isomers

Clearly, not all conceivable graphs correspond to molecules, for each of the atoms forming a given structure has a given valence which must be satisfied in the structure into which it is incorporated. It is evident that, in general, the same set of atoms chemically united to yield a single molecule may be connected together in different ways; the resulting structures are known as *chemical isomers*. For example, graphite is a chemical isomer of diamond because they have the same atom set but different atom patterns; see Figure 7.3. Determining the number of isomers for a specific molecule involves the application of graph enumeration. In fact, isomer enumeration has been one of the major applications of graph theory to chemistry [Wilson and Beineke 1979]. There are many reports which focused on determining isomers or enumerating isomers [Henze and Blair 1931; Randic 1974; Bohanec and Perdih 1993; Faulon 1998].

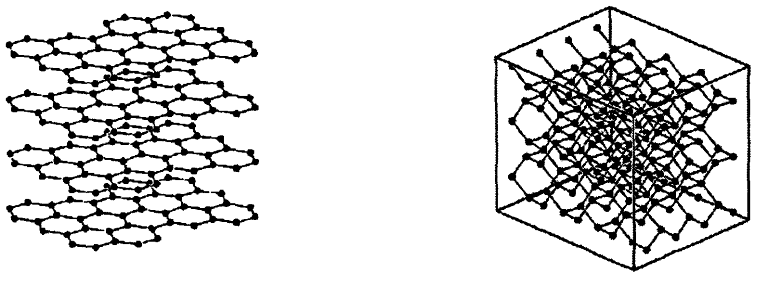


Figure 7.3 Chemical isomers: graphite and diamond.

7.3 Isomer Enumeration of the Alkane Series

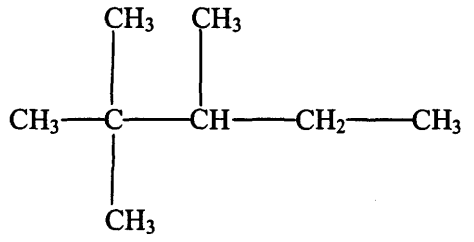
A hydrocarbon is any chemical compound containing only carbon and hydrogen. Hydrocarbons can be separated into acyclic and cyclic types. The acyclic hydrocarbons are characterized by a branched tree structure and can be separated into three categories:

(1) Alkanes contain only single bonds and have the general formula C_nH_{2n+2}.

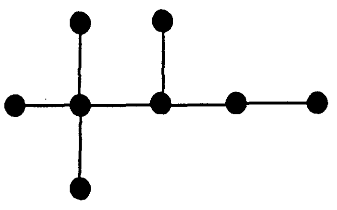
(2) Alkenes contain a double bond and have the general formula C_nH_{2n}.

(3) Alkynes contain a triple bond and have the general formula C_nH_{2n-2}.

According to the definition of alkanes, one can configure its chemical graph with carbon atoms regarded as vertices and chemical bonds regarded as edges by following three constraints: (1) meeting the general formula, (2) only single bonds between two carbons, and (3) no cycle is permitted. Figure 7.4 illustrates the chemical constitutional formula of octane (C_8H_{18}) and its chemical graph where hydrogen atoms are omitted. It is noted that in Figure 7.4b there are four kinds of vertices in terms of the degrees of these vertices from 1 to 4. Indeed, for the chemical graphs of alkanes, the highest degree of vertices is 4, because a carbon has 4 chemical bonds. If given a number of carbons at this point, one can calculate the combinations of such kinds of vertices such that the general formula is satisfied. The following discusses the method of calculating these combinations.



(a) chemical formula C_8H_{18}



(b) chemical graph without hydrogen

Figure 7.4 The chemical constitutional formula of octane (C_8H_{18}) and its chemical graph.

Suppose that n_i ($1 \leq i \leq 4$) represents the number of the carbons having *i* bonds connected with other carbons. For a carbon having *i* bonds connected with other carbons (i.e., a vertex having *i* degrees in the chemical graph), it needs to connect another 4-*i* hydrogen atoms (every carbon must have 4 chemical bonds). For a given number of carbons *N* (hydrogen atoms 2*N*+2), there is

$$n_1 + n_2 + n_3 + n_4 = N \tag{7.1}$$

and

$$3n_1 + 2n_2 + n_3 = 2N + 2 \tag{7.2}$$

By combining Equations (7.1) and (7.2) and solving these linear equations, one can achieve all possible combinations of carbons for a given *N*. For example, if *N*=5, three solutions can be achieved by Equations (7.1) and (7.2), i.e., (n_1=2, n_2=3, n_3=0, n_4=0),

(n_1=3, n_2=1, n_3=1, n_4=0), and (n_1=4, n_2=0, n_3=0, n_4=1). Table 7.1 lists the relationship between the number of carbons up to 16 and its carbon-carbon bond combinations.

Table 7.1 The number of combinations of carbons having different carbon-carbon bonds.

N	Comb.	N	Comb.	N	Comb.	N	Comb.
1	1	5	3	9	8	13	16
2	1	6	4	10	10	14	19
3	1	7	5	11	12	15	21
4	2	8	7	12	14	16	24

With these combinations of carbons, the chemical graphs meeting the general formula C_nH_{2n+2} can be enumerated by following the method discussed in Chapter 5. However, an additional constraint has to be employed when configuring the chemical graphs of alkanes using this method; i.e., no cycle is permitted during configurations of graphs, as mentioned above as the third constraint. This constraint can easily be incorporated into the general constraints. Figure 7.5 shows the illustration of how this constraint works, where the label on the edge indicates the step of the search. In Figure 7.5a, vertex 2 (carbon) accepts a token from vertex 5 (carbon) at the fourth step and then accepts a token from vertex 3 (carbon) at the fifth step. Once vertex 2 accepts this token from vertex 3, a closed loop is formed among vertices 1, 2, and 3. This closed loop results in a cycle of carbons. Therefore, such a configuration must be eliminated. The same situation happens in Figure 7.5b when vertex 4 (carbon) accepts a token from vertex 6 (carbon) at the sixth step. In this case, the closed loop is formed among vertices 1, 3, 4, and 6. Therefore, the configuration must also be eliminated. However, in Figure 7.5c, there is no closed loop formed and thus this configuration is accessible and one chemical graph is generated. Table 7.2 lists the number of the chemical graphs with up to 16 carbon atoms meeting all three constraints of alkanes.

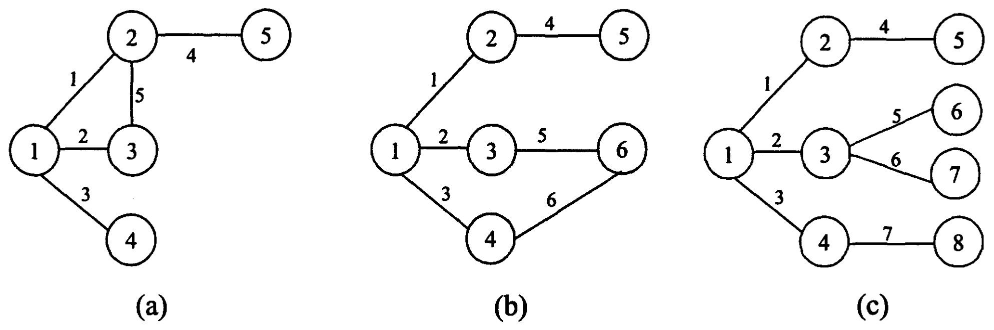


Figure 7.5 Illustration of configuring chemical graphs of alkanes.

Table 7.2 The number of chemical graphs meeting the constraints configuring alkanes.

Carbons	Graphs	Carbons	Graphs	Carbons	Graphs	Carbons	Graphs
1	1	5	4	9	71	13	2746
2	1	6	7	10	172	14	7078
3	1	7	14	11	426	15	18374
4	2	8	31	12	1076	16	48050

The chemical graphs of alkanes generated in Table 7.2 may include isomorphs. These isomorphs must be removed. At this point, the Eigensystem approach is employed for the isomer enumeration. The procedure is such that the canonical codes for all the chemical graphs listed in Table 7.2 are found. Among the chemical graphs of alkanes with the same canonical code, only one is kept. Table 7.3 lists the total number of isomers of the Alkane Series with up to 16 carbons. It should be noted that the results achieved here are the same as the records in chemistry. Appendix D lists the program codes.

Table 7.3 Number of all isomers of the Alkanes Series up to 16 carbons.

N	n_4-n_3-n_2-n_1	Number	Σ
1	0-0-0-0	1	1
2	0-0-0-2	1	1
3	0-0-1-2	1	1
4	0-0-2-2	1	2
	0-1-0-3	1	
5	0-0-3-2	1	3
	0-1-1-3	1	
	1-0-0-4	1	
6	0-0-4-2	1	5
	0-1-2-3	2	
	0-2-0-4	1	
	1-0-1-4	1	
7	0-0-5-2	1	9
	0-1-3-3	3	
	0-2-1-4	2	
	1-0-2-4	2	
	1-1-0-5	1	
8	0-0-6-2	1	18
	0-1-4-3	4	
	0-2-2-4	5	
	0-3-0-5	1	
	1-0-3-4	3	
	1-1-1-5	3	
	2-0-0-6	1	
9	0-0-7-2	1	35
	0-1-5-1	5	
	0-2-3-4	9	
	0-3-1-5	3	
	1-0-4-4	5	
9	1-1-2-5	8	
	1-2-0-6	2	
	2-0-1-6	2	
10	0-0-8-2	1	75
	0-1-6-3	7	
	0-2-4-4	17	
	0-3-2-5	10	
	0-4-0-6	2	
	1-0-5-4	6	
	1-1-3-5	17	
	1-2-1-6	8	
	2-0-2-6	5	
	2-1-0-7	2	
11	0-0-9-2	1	159
	0-1-7-3	8	
	0-2-5-4	27	
	0-3-3-5	24	
	0-4-1-6	6	
	1-0-6-4	9	
	1-1-4-5	33	
	1-2-2-6	28	
	1-3-0-7	4	
	2-0-3-6	10	
	2-1-1-7	8	
	3-0-0-8	1	
12	0-0-10-2	1	355
	0-1-8-3	10	
	0-2-6-4	43	
	0-3-4-5	55	

Table 7.3 (*Continued*)

N	n_4-n_3-n_2-n_1	Number	Σ	N	n_4-n_3-n_2-n_1	Number	Σ
12	0-4-2-6	24		14	0-3-6-5	206	
	0-5-0-7	2			0-4-4-6	183	
	1-0-7-4	11			0-5-2-7	52	
	1-1-5-5	58			0-6-0-8	4	
	1-2-3-6	73			1-0-9-4	18	
	1-3-1-7	21			1-1-7-5	153	
	2-0-4-6	20			1-2-5-6	364	
	2-1-2-7	28			1-3-3-7	275	
	2-2-0-8	6			1-4-1-8	52	
	3-0-1-8	3			2-0-6-6	61	
13	0-0-11-2	1	802		2-1-4-7	186	
	0-1-9-3	12			2-2-2-8	132	
	0-2-7-4	63			2-3-0-9	14	
	0-3-5-2	109			3-0-3-8	28	
	0-4-3-6	69			3-1-1-9	21	
	0-5-1-7	11			4-0-0-10	2	
	1-0-8-2	15		15	0-0-13-2	1	4347
	1-1-6-5	97			0-1-11-3	16	
	1-2-4-6	174			0-2-9-4	127	
	1-3-2-7	86			0-3-7-5	360	
	1-4-0-8	8			0-4-5-6	423	
	2-0-5-6	35			0-5-3-7	182	
	2-1-3-7	76			0-6-1-8	23	
	2-2-1-8	31			1-0-10-4	23	
	3-0-2-8	11			1-1-8-5	233	
	3-1-0-9	4			1-2-6-6	717	
14	0-0-12-2	1	1858		1-3-4-7	759	
	0-1-10-3	14			1-4-2-8	254	
	0-2-8-4	92			1-5-0-9	15	

Table 7.3 (*Continued*)

N	n_4-n_3-n_2-n_1	Number	Σ	N	n_4-n_3-n_2-n_1	Number	Σ
15	2-0-7-6	98		16	1-0-11-4	27	
	2-1-5-7	405			1-1-9-5	342	
	2-2-3-8	428			1-2-7-6	1311	
	2-3-1-9	101			1-3-5-7	1859	
	3-0-4-8	71			1-4-3-8	942	
	3-1-2-9	90			1-5-1-9	128	
	3-2-0-10	14			2-0-8-6	155	
	4-0-1-10	7			2-1-6-7	824	
16	0-0-14-2	1	10359		2-2-4-8	1222	
	0-1-12-3	19			2-3-2-9	508	
	0-2-10-4	174			2-4-0-10	37	
	0-3-8-5	606			3-0-5-8	154	
	0-4-6-6	920			3-1-3-9	302	
	0-5-4-7	560			3-2-1-10	102	
	0-6-2-8	124			4-0-2-10	28	
	0-7-0-9	6			4-1-0-11	8	

7.4 Concluding Remarks

As examples, the isomers of the Alkane Series with up to 16 carbons have been enumerated in this chapter. For a given number of carbons, the combinations of different carbons having different chemical bonds with other carbons are first calculated so that these combinations of the carbons can meet the general formula C_nH_{2n+2} and also meet the single bond criteria. With these combinations of the carbons, all chemical graphs meeting the general formula C_nH_{2n+2} can be achieved by incorporating the constraint (no closed loop is permitted) into the method discussed in Chapter 5. The Eigensystem approach has been applied lastly to enumerate all isomers of the Alkane Series. Canonical codes of all chemical graphs of Alkane series have been created through the Eigensystem approach. Unique canonical labels have been collected to create the enumeration of distinct kinematic chains. For the known cases available in publications, the related results obtained through the Eigensystem approach have been checked and confirmed the same as those reported in chemistry literature [Wilson and Beineke 1979].

Computational synthesis of nanotube based gears was studied by Han *et al.* [1997]; see Figure 7.6 where two molecular multiple teeth gear systems are fashioned from carbon nanotubes. The fact that the Eigensystem approach works effectively in structure synthesis for moleculae design well implies that the Eigensystem approach can be used in structure synthesis for nano molecular machine design.

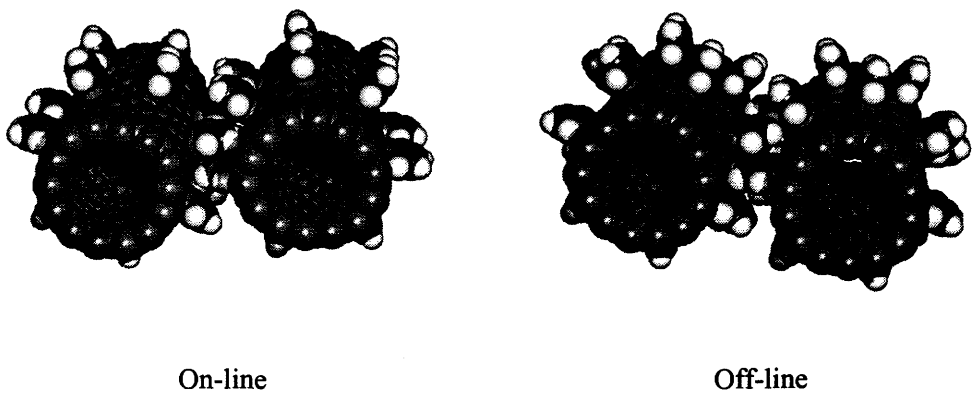


Figure 7.6 Two molecular multiple teeth gear systems [Han *et al.* 1997].

CHAPTER 8
CONCLUSIONS

8.1 Overview

Structure synthesis is a step in design that has the highest potential to result in innovative products. This thesis study was motivated by two things: (1) the lack of effective computational tools for general structure synthesis and (2) the author's preliminary finding that both eigenvalues and eigenvectors are useful to general graph (or structure) synthesis.

There are three fundamental problems in structures synthesis. Problem I: Coding and indexing of structures for efficient computer storage and retrieval of structures. Problem II: Identification of structure isomorphism (a kind of similarity). Problem III: Enumeration of structures subject to a set of constraints. The objective of this thesis study was to develop theories/methods, algorithms, and computer programs to support the solving process of these three problems.

As a first step, graphs were used to represent the structures. A graph contains a set of vertices and edges. Both the vertex and the edge can have different types. When the types are expressed in the graph, the colored graph forms. With this notion and the notion of hierarchical organization, the graph becomes a powerful tool to represent various scenarios of the structure. With the graph, structure synthesis becomes graphs synthesis. Furthermore, the graph can be represented by the matrix, in particular, the adjacency matrix (AM). This then makes it possible to computer-manipulate the graph.

Among the three fundamental problems, Problem II, the graph isomorphism problem is the basic. If there is a one-to-one mapping between two graphs, an isomorphic relationship is established between these two graphs, or these two graphs are said to be isomorphic. This problem is known to the graph theorist as the Non-deterministic Polynomial (NP) problem, which means that the computational time for solving such a kind of problem can be overwhelmingly long and, in the worst situation of some applications, it may be beyond the capacity of the modern computing facility. Problem II was first tackled in this thesis.

After a literature review (Chapter 2), the discussion was focused on the full development of the method based on both the eigenvalues and eigenvectors of the adjacency matrix of the graph – a finding generated by the author [He *et al.* 2000]. The key feature of this method was to largely make use of the information of eigenvectors, particularly their components, in seeking a one-to-one mapping between two graphs. This method was called the Eigensystem approach. It was demonstrated that the complexity of this Eigensystem approach was $O(m_3n^3 + m_3m_1(n\lg n + n + m_2))$, which is the same as the well known program called Nauty for the detection of graph isomorphisms. In addition to the completely different thought underlying the Eigensystem method compared with that underlying the program Nauty, the Eigensystem approach allows graphs with weight or non-weight and with directed edge or undirected edge to be all converted, i.e., their isomorphisms are detectable. It is noted that the current version of the program Nauty has only demonstrated the detection of isomorphic graphs with non-weight (both undirected and directed edge).

To tackle the first and third problems, this thesis further described the extension of the Eigensystem approach for the basic problem called graph counting. The graph counting problem adds one more requirement on top of the graph isomorphism, i.e., finding all the one-to-one mappings (instead of just one such a mapping, which is the case for the graph isomorphism problem). After the graph counting problem was elucidated, the canonical labeling of graphs was discussed. It was demonstrated that the Eigensystem approach can readily be extended to perform the canonical labeling, which results in a compact code for a graph. Such a code has a one-to-one correspondence to a graph and all its isomorphic graphs. Computer storage of graphs can be made possible via such a canonical code. A method was proposed to solve the third fundamental problem (i.e., the enumeration of graphs). This method was basically constraint-oriented, i.e., converting application problems into constraint equations on the graph, and then finding all the solutions which satisfy the constraints. The role of the Eigensystem approach here was a tool to screen out the isomorphic graphs that only met the constraint equations among all the graphs generated.

Two examples were made to illustrate how to apply the methods and tools developed: one is the machine design, and the other is the molecular design. These examples were also used to verify the methods and tools by comparing their results with others' results that are known to be correct. There were also some new results generated.

8.2 Contribution

This thesis has proposed a novel method and developed a suite of computer algorithms and programs for structure or graph synthesis. This new method, together with its algorithms/ programs, may be simply called the Eigensystem approach. The spectra of graphs covered by the Eigensystem approach include weighted/non-weighted, directed/undirected graphs. The complexity of the Eigensystem approach is $O(m_3n^3 + m_3m_1(n\lg n + n + m_2))$, the same as that of the program Nauty which is the best algorithm/program today for graph isomorphism, graph counting, and canonical labeling problems. However, the program Nauty has so far demonstrated its applicability to non-weighted graphs only. In the author's opinion, the extension of the program Nauty to any kind of weighted graph can be very difficult owing to the fundamental thought underlying this program.

The developed method, which was based on the constraint satisfaction problem together with the isomorphism detection for the enumeration of structures, can be very general and robust. The application of this method to the mechanism design problem has resulted in new findings for the total number of distinct structures of the kinematic chains that has 12 links and (5, 7, 9) DOFs. It is noted that, in the literature, the report for the total number of distinct structures has been given for the 12 bar linkage with 1 DOF only.

8.3 Future Work

Three issues, (1) synthesis of 3D structures, (2) enumeration of structures, and (3) optimization of the algorithms, need to be addressed in the future.

The synthesis of 3D structures is becoming important in fields such as biochemistry. In mechanical engineering, the need for 3D structure synthesis is emerging with the expected design and development of microstructures and nanostructures. Different from the synthesis of topological structures, the synthesis of 3D structures needs to consider the geometrical information of structures together with the topological information. Here, the major challenge is how the geometrical information of a 3D structure can be represented into a graph.

In the general algorithm for the graphs enumeration developed in Chapter 5, only two constraints (the total number of vertices and the degrees of vertices) were considered. When new constraints are considered, the algorithm has to be extended in an ad-hoc manner. The constraint satisfaction problem (CSP) appears to be readily applicable to the problem of the structure enumeration. The benefits of CSP include: (1) CSP allows for representing constraints in a declarative manner, and hence the adding of new constraints does not require changing the algorithm for searching solutions that satisfy the constraints, and (2) CSP has a rich suite of programs available to find all solutions given a set of constraints.

The major goal of the computer program developed in this thesis is to examine whether the proposed algorithms are correct. At this point, the implementation of the algorithms was developed in the MATLAB 6.5 environment. To improve the efficiency of the implementation of these algorithms, it should be possible to code these algorithms outside of any mathematical computation package environment. There is also some room to streamline these algorithms for faster computational rates, for example, incorporating Vector Choose Rule I discussed in Chapter 5 into algorithm I-2.

LIST OF REFERENCES

Ambekar, A. G. and Agrawal, V. P., Canonical Numbering of Kinematic Chains and Isomorphism Problem: min Code, *Mech. Mach. Theory*, v22, n5, pp. 453-461, 1987

Antonsson, E. K., and Cagan, J., eds., *Formal Engineering Design Synthesis*, Cambridge University Press, 2001

Babai, L., Automorphism Groups, Isomorphism, Reconstruction in *Handbook of Combinatorics Volume 2* edited by Graham, R. L., Grotschel, M., and Lovasz, L., North-Holland: The MIT Press, pp. 1447-1540, 1995

Babai, L., Erdos, P., and Selkows, S., Random graph isomorphism, *SIAM Journal of Computing*, vol. 9, no. 3, pp. 628-635, 1980

Babel, L., Isomorphism of Chordal (6,3) Graphs, *Computing*, vol. 54, pp. 303-316, 1995.

Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and Vorst, H. van der, *Templates for The Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, 2000

Balaban, A. T., and Harary, F., The characteristic polynomial does not uniquely determine the topology of a molecule, *J. Chem. Documentation*, vol. 11, pp. 258-259, 1971

Balcazar, J. L., Diaz, J., and Gabarro, J., *Structural Complexity I* , Berlin: Springer-Verlag, 1994

Basak, S. C., Bertelsen, S., and Grunwald, G. D., Application of Graph Theoretical Parameters in Quantifying Molecular Similarity and Structure-Activity Relationships, *Journal of Chemical Information and Computer Sciences*, vol. 34, no. 2, pp. 270-276, 1994

Basak, S. C., and Magnuson, V. R., Determining Structural Similarity of Chemicals Using Graph-Theoretic Indices, *Discrete Applied Mathematics*, vol. 19, pp. 17-44, 1988

Basak, S. C., Mills, D., Gute, B. D., Grunwald, G. D., and Balaban, A. T., Applications of Topological Indices in Predicting Property/Bioactivity/Toxicity of Chemicals, *Topology in Chemistry*, Rouvray, D. H., and King, R. B. (Eds), Ellis Howard, pp. 1-71, 2001

Basak, S. C., AND Niemi, G. J., Optimal characterization of structure for prediction of properties, *Journal of Mathematical Chemistry*, vol. 4, pp. 185-205, 1990

Berztiss, A. T., A backtrack procedure for isomorphism of directed graphs, *Journal of the ACM*, vol. 20, no. 3, pp. 365-377, 1973

Bodlaender, H. L., Polynomial Algorithms for Graph Isomorphism and Chromatic Index on Partial *k*-Trees, *Journal of Algorithms*, vol. 11, pp. 631-643, 1990.

Bohanec, S., and Perdih, M., Symmetry of chemical structures: a novel method of graph automorphism group determination, *J. Chem. Inf. Comput. Sci.*, vol. 33, pp. 719-726, 1993

Butcher, E., and Hartman, C., Enumeration and classification of 12-bar planar, simple-jointed kinematic chains using a hierarchical approach, *Proceedings of the 2002 ASME Design Engineering Technical Conferences,* DETC2002/MECH-34253, Montreal, Canada, September 2002

Carre, B., *Graphs and Networks*, Oxford: Clarendon Press, 1979

Chandrasekaran, B., and Josephson, R. J., Function in Device Representation, *Engineering with Computers*, 16:162-177, 2000

Chen, I. M., and Burdick, J. W., Enumerating the Non-Isomorphic Assembly Configurations of Modular Robotic Systems, *International Journal of Robotics Research*, vol. 17, no. 7, pp. 702-719, 1998

Collatz, L., Sinogowitz, U., Spektren endlichen Graphen, *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, vol. 21, no.1, pp. 63-77, 1957

Cormen, T. H., Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms*, Cambridge, Mass.: MIT Press, 2001

Corneil, D. G., and Gotlieb, C. C., An efficient algorithm for graph isomorphism, *J. Assoc. Comput. Mach.*, vol. 17, pp. 51-64, 1970

Corneil, D. G., and Kirkpatrick, D. G., A theoretical analysis of various heuristics for the graph isomorphism problem, *SIAM J. Comput.*, vol. 9, pp. 281-297, 1980

Crossley, F. E., A contribution to Gruebler's theory in the number synthesis of plane mechanisms, *Trans. ASME J. Eng. Ind.*, pp. 1-8, 1964

Crossley, F. E., The permutations of Kinematic Chains of 8 Members or Less from the Graph-Theoretic Viewpoint, *Developments in Theoretical and Applied Mechanics*, Vol. 2, Pregamon Press: Oxford, pp. 467-486, 1965

Crossley, F. E., On an Unpublished Work of Alt, *Journal of Mechanisms,* vol. 1, pp. 165-170, 1966

Davies, T. H., and Crossley, F. E., Structural Analysis of Plane Linkages by Franke's Condensed Notation, *Journal of Mechanisms*, vol. 1, pp. 171-183, 1966

Dobrjanskyi, L., and Freudenstein, F., Some applications of graph theory to the structural analysis of mechanisms, *Trans. ASME J. Eng. Ind.*, pp. 153-158, 1967

Faulon J. L., Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs, *J. Chem. Inf. Comput. Sci.*, vol. 38, pp. 432-444, 1998

Fogel, D. B., An Introduction to Simulated Evolutionary Optimization, *IEEE Trans. On Neural Networks*, vol. 5, no. 1, pp.3-14, 1994

Fortin, S., The Graph Isomorphism Problem, *Technical Report TR 96-20*, University of Alberta, July 1996.

Freudenstein, F., The basic concepts of Polya's theory of enumeration, with application to the structural classification of mechanisms, *J. Mech.*, vol. 3, pp. 275-290, 1967

Garey, M. R., and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W. H. Freeman and Company, 1979

Gero, J. S., Design Prototypes: A Knowledge Representation Schema for Design, *AI Magazine*, 11(4): 26-36, 1990

Gero, J., Computational Models of Creative Design Processes in T. Dartnall (eds.), *AI in Creativity*, Kluwer, Dordrecht, pp. 269-281, 1994

Giovanni, D., Synthesis and Optimization of Digital Circuit, New York: McGraw-Hill Inc., 1992

Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989

Guillemin, E. A., *The Mathematics of Circuit Analysis*, New York: John Wiley and Sons, Inc., 1949

Gute, B. D., and Basak, S. C., Molecular similarity-based estimation of properties: a comparison of three structure spaces, *Journal of Molecular Graphics and Modeling*, vol. 20, pp. 95-109, 2001

Haas, S. L., and Crossley, F. R. E., Structural synthesis of a four-bit binary adding mechanism, *Trans. ASME J. Eng. Ind.*, vol. 91B, pp. 240-250, 1969

Han, J., Globus, A., Jaffe, R., and Deardorff, G., Molecular dynamics simulations of carbon nanotube-based gears, *Nanotechnology*, vol. 8, pp. 95-102, 1997

Harary, F., The Determinant of the Adjacency Matrix of a Graph, *SIAM Review*, vol. 4, no. 3, pp. 202-210, 1962

Harary, F., *Graph Theory*, Reading, Mass. : Addison-Wesley Pub. Co., 1969

Harary, F., King, C., Mowshowitz, A., Read, R. C., Cospectral graphs and digraphs, *Bulletin of the London Mathematical Society*, vol. 3, pp. 321-328, 1971

He, P. R., Zhang, W. J., and Li, Q., Comparison of Design Knowledge, *technical report no. 2000-01*, Advanced Engineering Design Lab, Department of Mechanical Engineering, University of Saskatchewan, 2000.

He, P. R., Zhang, W. J., and Li, Q., A quadratic form-based approach to identification of kinematic chains in structural analysis and synthesis of mechanisms, *Proceedings of the 2001 ASME Design Engineering Technical Conferences*, DETC2001/DAC-21067, Pittsburgh, PA, USA, September 2001.

He, P. R., Zhang, W. J., and Li, Q., 2002a, An Efficient Approach to Structure Identification in Engineering and Science, *Proceedings of the 2002 CSME Forum*, Kingston, Canada, ISBN: 0-9730900-1-4, May 2002.

He, P. R., Zhang, W. J., and Li, Q., 2002b, A proposed eigenvalue-eigenvector approach for graph isomorphism detection, submitted to *Journal of The Franklin Institute* on June 18, 2002

He, P. R., Zhang, W. J., and Li, Q., 2002c, Eigenvalue and eigenvector information of graphs and their validity in detection of graph isomorphism, *Proceedings of the 2002 ASME Design Engineering Technical Conferences,* DETC2002/MECH-34247, Montreal, Canada, September 2002

He, P. R., Zhang, W. J., Li, Q., and Wu, F. X., A New Method for Detection of Graph Isomorphism Based on the Quadratic Form, ASME *Journal of Mechanical Design*, vol. 125, pp. 640-642, September 2003.

Heisserman, J., Comparing Designs: A Graphical Diff for Digital Mockups, *Proceedings of DETC'99 1999 ASME Design Engineering Technical Conferences*, DETC99/DFM-8929, Las Vegas, Nevada, September 12-15, 1999

Henze, H. R., and Blair, C. M., The number of isomeric hydrocarbons of the methane series, *Journal of American Chemistry Society*, vol. 53, pp. 3077-3085, 1931

Hoffmann, C. M., *Group-Theoretic Algorithms and Graph Isomorphism*, New York: Springer-Verlag, 1982

Hwang, W. M., and Hwang, Y. W., Computer-aided structural synthesis of planar kinematic chains with simple joints, Mechanism and Machine Theory, vol. 27, no. 2, pp. 189-199, 1992

Johnson, R. C., and Towfigh, K., *Mechanical Design Synthesis – Creative Design and Optimization*, Krieger, Huntington, NY, 1978

Kim, J. T., and Kwak, B. M., An Algorithm of Topological Ordering for Unique Representation of Graphs, *Trans. ASME J. Mech. Design*, vol. 114, pp. 103-108, 1992

Kobler, J., Schoning, U., and Toran, J., *Graph Isomorphism Problem: its Structural Complexity*, Birkhauser, Boston, 1993

Kudo, Y., Yamasaki, T., and Sasaki, S., The characteristic polynomial uniquely represents the topology of a molecule, *J. Chem. Documentation*, vol. 13, pp. 225-227, 1973

Li, Q. and Zhang, W.J., Computer comparison of design knowledge, *Proceedings of I.Mech.E., Part B, J. of Engineering Manufacturing*, v212, pp. 635-645, 1999

Littlewood, D. E., *The Theory of Group Characters and Matrix Representations of Groups*, Oxford: Clarendon Press, 1940

Liu, B., and Lai, H. J., *Matrices in Combinatorics and Graph Theory*, Boston: Kluwer Academic Publishers, 2000

Luks, E. M., Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time, *Journal of Computer and System Sciences*, vol. 25, no. 1, pp. 42-65, 1982

Manolescu, N. I., A method based on Baranov trusses, and using graph theory to find the set of planar jointed kinematic chains and mechanisms, *Mechanism and Machine Theory*, vol. 8, pp. 3-22, 1973

Mathon, R., A Note on the Graph Isomorphism counting Problem, *Information Processing Letters*, vol. 8, no. 3, pp. 131-132, 1979

McGregor, J., Relational consistency algorithms and their application in finding subgraph and graph isomorphism, *Information Sciences*, vol. 19, pp. 229-250, 1979

McKay, B. D., Practical Graph Isomorphism, *Congressus Numerantium*, vol. 30, pp. 45-87, 1981

Minc, H., *Nonnegative Matrices*, Wiley, New York, 1988

Mittal, H. B., A fast backtrack algorithm for graph isomorphism, *Information Processing Letters*, vol. 29, no.2, pp. 105-110, 1988

Miyazaki, T., The Complexity of McKay's Canonical Labeling Algorithm, *Groups and Computation, II* (Finkelstein, L., and Kantor, W.M., eds.), Amer. Math. Soc., Providence, RI, pp. 239-256, 1997

Mowshowitz, A., The Characteristic Polynomial of a Graph, *Journal of Combinatorial Theory*, vol. 12(B), no.2, pp. 177-193, 1972

Mruthyunjaya, T. S., Structural Synthesis by Transformation of Binary Chains, *Mechanisms and Machine Theory*, vol. 14, pp. 221-231, 1979

Mruthyunjaya, T. S., and Raghavan, Structural Analysis of Kinematic Chains and Mechanisms Based on Matrix Representation, *Transactions of the ASME Journal of Mechanical Design*, v101, pp. 488-494, 1979

Mruthyunjaya, T. S., A computerized methodology for structural synthesis of kinematic chains: part 1—formulation, *Mechanism and Machine Theory*, v19, n6, pp. 487-495, 1984

Murdock, W. J., Modeling Invention by Analogy in ACT-R, *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, Madison, WI, August 1998

Ogata, H., Fujibuchi, W., Goto, S., and Kanehisa, M. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters, *Nucleic Acids Research*, vol. 28, pp. 4021-4028, 2000

Ortega, J. M., *Matrix Theory: A Second Course*, Plenum Press, New York, 1987

Randic, M., On the recognition of identical graphs representing molecular topology, *J. Chem. Phys.*, vol. 60, pp. 3920-3928, 1974

Rao, A. C., and Raju, D. V., Application of the Hamming Number Technique to Detect Isomorphism among Kinematic Chains and Inversions, *Mech. Mach. Theory*, vol. 26, no. 1, pp. 55-75, 1991

Rao, A. C., and Deshmukh, P. B., Computer aided structural synthesis of planar kinematic chains obviating the test for isomorphism, *Mechanism and Machine Theory*, vol. 36, pp. 489-506, 2001

Read, R. C., and Corneil, D. G., The Graph Isomorphism Disease, *Journal of Graph Theory*, vol. 1, pp. 339-363, 1977.

Reich, Y., Design Knowledge Acquisition: Task Analysis and a Partial Implementation, *Knowledge Acquisition: An International Journal of Knowledge Acquisition for Knowledge-Based System*, vol. 3, no. 3, pp. 234-254, 1991

Schmidt, D. C., and Druffel, L. E., A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices, *Journal of the ACM*, vol. 23, no. 3, pp. 433-445, 1976

Schmidt, L. C., Shetty, H., and Chase, S. C., A Graph Grammar Approach for Structure Synthesis of Mechanisms, *Journal of Mechanical Design*, vol. 122, pp. 371-376, 2000

Schoning, U., Graph Isomorphism Is in the Low Hierarchy, *Journal of Computer and System Science*, vol. 37, pp. 312-323, 1988

Seneta, E., *Non-negative Matrices and Markov Chains*, Springer-Verlag, New York, 1981

Shah, Y. J., Davida, G. I., and McCarthy, M. K., Optimum Features and Graph isomorphism, *IEEE Transactions on Systems, Man, and Cybernetics SCM-4*, pp. 313-319, 1974

Sims, K., Evolving Virtual Creatures, *Computer Graphics (Proceedings of SIGGRAPH'94)*, pp. 15-22, July 1994

Sohn, W. J., and Freudenstein, F., An Application of Dual Graphs to the Automatic Generation of the Kinematic Structures of Mechanisms, *Journal of Mechanisms, Transmissions, and Automation in Design*, vol. 108, pp. 392-398, 1986

Soni, A. H., Structural analysis of two general constraint kinematic chains and their practical application, *Trans. ASME J. Eng. Ind.*, vol. 93B, no. 1, pp. 231-238, 1971

Spialter, L., The atom connectivity matrix (ACM) and its characteristic polynomial (ACMCP): A new computer-oriented chemical nomenclature, *J. Amer. Chem. Soc.*, vol. 85, pp. 2012-2013, 1963

Spialter, L., The atom connectivity matrix (ACM) and its characteristic polynomial, *J. Chem. Documentation*, vol. 4, pp. 261-269, 1964

Suh N.P., *The Principles of Design*, New York: Oxford University Press, 1990

Sussenguth, E. H., A graph-theoretic algorithm for matching chemical structures, *Journal of Chemical Documentation*, vol. 5, pp. 36-43, 1965

Tang, C. S. and Liu, T., The Degree Code – A New Mechanism Identifier, *Transactions of the ASME Journal of Mechanical Design*, v115, pp. 627-630, 1993

Tsang, E. P. K, *Foundations of Constraint Satisfaction*, Academic Press, London, 1993

Tischler, C. R., Samuel, A. E., and Hunt, K. H., Kinematic Chains for Robot Hands-I. Orderly Number-Synthesis, *Mechanism and Machine Theory*, vol. 30, pp. 1193-1215, 1995

Tomiyama, T., Yoshikawa, H., Extended general design theory, *Design Theory for CAD*, Yoshikawa, H. and Warman, E.A. (eds), North-Holland, Amsterdam, pages 95-130, 1987

Tong, C., and Sriram, D. eds., *Artificial Intelligence in Engineering Design*, Vol. I, Academic Press, 1992

Turner, J., Generalized matrix functions and the graph isomorhism problem, *SIAM J. Appl. Math.*, vol. 16, pp. 520-536, 1968

Tuttle, E. R., Peterson, J. E., and Titus, J. E., Enumeration of basic kinematic chains using the theory of finit groups, *Trans. ASME J. Mech. Trans. Automat. Design*, vol. 111, pp. 498-503, 1989

Ubar, R., Test Synthesis with Alternative Graphs, *IEEE Design and Test of Computers*, vol. 13, no. 1, pp. 48-57, 1996

Uicker, J. J., Pennock, G. R., and Shigley, J. E., *Theory of Machines and Mechanisms*, Oxford University Press, 2003

Uicker, J. J., Raicu, A., A method for the identification and recognition of equivalence of kinematic chains, *Mechanism and Machine Theory*, vol. 10, pp. 375-383, 1975

Ullman, D. G., *The Mechanical Design Process*, McGraw-Hill, New York, 1997

Umeda, Y, Ishii, M., Yoshioka, M., Shimomura, Y., and Tomiyama, T., Supporting Conceptual Design on the Function-Behavior-State Modeler, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10:275-288, 1996

Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H., Function, Behavior, and Structure, *Applications of Artificial Intelligence in Engineering, V* by Gero, J. S. eds., Computational Mechanics Publications and Springer-Verlag, pp. 177-193, 1990

Unger, S. H., GIT - a heuristic program for testing pairs of directed line graphs for isomorphism, *Communications of the ACM*, vol. 7, no. 1, pp. 26-34, 1964

Wilson, R. J., and Beineke, L. W. (eds.), *Applications of Graph Theory*, London: Academic Press, 1979

Woo, L. S., Type Synthesis of Plane Linkages, *Transactions of the ASME Journal of Engineering for Industry*, pp. 159-172, 1967

Yan, H.S. and Chen, J.J., Creative Design of a Wheel Damping Mechanism, *Mechanism and Machine Theory*, Vol. 20, pp. 597-600, 1985

Yan, H. S., and Hall, A. S., Linkage Characteristic Polynomials: Definitions, Coefficients by Inspection, *ASME Journal of Mechanical Design*, vol. 103, pp. 578-584, 1981

Yan, H. S., and Hall, A. S., Linkage Characteristic Polynomials: Assembly Theorems, Uniqueness, *ASME Journal of Mechanical Design*, vol. 104, pp. 11-20, 1982

Zhang, W. J., and Li, Q., On a New Approach to Mechanism Topology Identification, *Trans. ASME J. Mech. Design*, vol. 121, pp. 57-64, 1999

APPENDIX A
PROGRAM FOR GRAPH ISO-/AUTO-MORPHISM

The program in Appendix A is used for graph isomorphism/automorphism test. It consists of seven functions running in MATLAB 5.0 and later version, i.e., **graph**, **QuadraticForm**, **recusive**, **CompareVector**, **checkisomorphism**, **DeterminingSign**, and **NodeMatching**. The main function is **graph**(A,B) in which two augments A and B are adjacency matrices of two graphs, respectively. All possible isomorphism/ automorphism mappings are returned by running the program. A brief description for each function is introduced at the beginning of each function.

```
% Program for graph isomorphism/automorphism and their counting
% MATLAB M-file
% August 2003
% Main function graph(A,B)
% -- for graph isomorphism, A and B are adjacency matrices of two graphs
% -- for graph automorphism, enter B=A


function graph(A,B)
% Input --- adjacency matrices of graph A and B
% Output -- all possible isomorphic mappings between A and B

global FeaGro;
% FeaGro -- The matrix stored the node relationships of garph A and B
%           with constructing the same feature (component of
eigenvector)
%           into group
global group;
% number of feature groups
global n;
% n - node number of graph
global TOL;
global Bnode;
global total;

if (nargin~=2)
```

```
   disp('Input arguments are undefined!');
   error(' ');
end
i=size(A);
j=size(B);
if (i(1)~=i(2) | j(1)~=j(2))
   disp('Must be square matrices!');
   error(' ');
end
if (~isequal(A,A') | ~isequal(B,B'))
   disp('Must be symmetric matrices!');
   error(' ');
end
TOL = 0.0001;
group=1;
if (i(1)~=j(1))
   disp('They are not isomorphic graphs because of having different
nodes.');
else
   n = i(1);
   P=ones(n,1);
   Am=A*P;
   Bm=B*P;
   AAm=A;
   BBm=B;
   for k=1:n
      AAm(k,k)=n-Am(k);
      BBm(k,k)=n-Bm(k);
   end
   FeaGro(group).number=1;
   FeaGro(group).index=zeros(2,n);
   result=QuadraticForm(AAm,BBm);
   if (result==2)
      total=1;
      result=recusive(AAm,BBm,n,FeaGro,group);
      disp('Group-to-group mapping case');
      if (total>1)
         disp('These two graphs are isomorphic with ');
         disp(total-1);
         disp(' vertex mappings stored in "result.txt".');
         file=fopen('result.txt','w');
         for j=1:total-1
            for k=1:n
               fprintf(file,'%3d',Bnode(j).index(k));
            end
            fprintf(file,'\n');
         end
         fclose(file);
      else
         disp('These two graphs are not isomorphic.');
      end
   elseif (result==1)
      for j=1:group
         disp(FeaGro(j).index);
      end
      disp('These two graphs are isomorphic.');
   elseif (result==-1)
      disp('These two graphs are not isomorphic.');
   else
      disp('Oops! I can not make a decision now.');
      disp('An optional relationship of node numbers between graph A and
B:');
      [DisFea,DisInx]=sort(FeaGro(group).index');
      disp(DisInx');
   end
```

```
function result=QuadraticForm(A,B)
% A,B -- adjacency matrices of graph A and B, respectively
% result -- return 1 if having one-to-one relationship, return 0 if
%      can't make a decision, return 2 if further test is needed,
%      otherwise -1
%
% Av,Bv -- eigenvector matrices of A and B graphs
%
global FeaGro;
global Av;
global Bv;
global n;
global TOL;
ABm = A';
[Av,Ad]=eig(ABm);
ABm = B';
[Bv,Bd]=eig(ABm);
M=ones(n,1);
ABd=Ad*M;
[Ad1,IAd]=sort(ABd);
ABd=Bd*M;
[Bd1,IBd]=sort(ABd);
if (abs(Ad1-Bd1)<TOL)
   x=1;
   r=2;
   for i=1:n-1
      if (abs(Ad1(i+1)-Ad1(x))>TOL)
         if (i==x)
            r=CompareVector(IAd(i),IBd(i));
            if (r<0)
               break;
            end
         end
         x=i+1;
      end
   end
   if (x==n & r>=0)
      r=CompareVector(IAd(n),IBd(n));
   end
   if (x==1 & r>=0)
      % only one eigenvalue, I guess that they are isomorphic
      result=1;
   else
      if (r==2)
         % every eigenvalue has at least a repeated root
         result=0;
      elseif (r==1)
         % isomorphic
         result=1;
      elseif (r==-1)
         % non-isomorphic
         result=-1;
      else
```

```
            % Further test is required
            result=2;
         end
      end
   else
      % non-isomorphic: eigenvalues are different
      result=-1;
   end


   function r=recusive(A,B,n,FeaGroPre,Fgroup)
   % A,B ---- adjacency matrices of graph A and B
   % n ------ number of vertices
   % FeaGroPre ---- mapping relationships in feature groups
   % Fgroup -- number of feature groups

   global FeaGro;
   global group;
   global Bnode;
   global total;

   for k=1:Fgroup
      AA=A;
      BB=B;
      group=1;
      FeaGro(group)=FeaGroPre(k);
      [TemFeaGro,Ifg]=sort(FeaGroPre(k).index');
      j=1;
      t(1)=n;
      t(2)=n;
      for i=2:n
         if (TemFeaGro(i,1)==TemFeaGro(i-1,1))
            j=j+1;
         else
            if (j>1)
               if (t(1)>j)
                  t(1)=j; % the number of vertices in a feature group
                  t(2)=i-1; % the location of the final vertex in a feature
group
               end
               j=1;
            end
         end
      end
      % Compare the last feature group
      if (j>1)
         if (t(1)>j)
            t(1)=j; % the number of vertices in a feature group
            t(2)=i; % the location of the final vertex in a feature group
         end
      end
      mm=t(2)-t(1)+1; % Location of the first vertex in the group
      ma=Ifg(mm,1);
      % Change an element of matrix A by plusing ma at 'ma'
      AA(ma,ma)=AA(ma,ma)+ma;
      for j=1:t(1)
         mb=Ifg(mm+j-1,2);
         % Plus the element of matrix B by ma at 'mb'
         BB(mb,mb)=BB(mb,mb)+ma;
         r=QuadraticForm(AA,BB);
      if (r==2)
         r=recusive(AA,BB,n,FeaGro,group);
      else
         if (r==1)
            for l=1:group
               [DisFea,DisInx]=sort(FeaGro(l).index');
               r=checkisomorphism(A,B,DisInx);
               if (r==1)
                  Bnode(total).index=[0 1]*DisInx';
                  disp(Bnode(total).index);
                  total=total+1;
               end
            end
         end
      end
      BB(mb,mb)=BB(mb,mb)-ma;
      group=1;
      FeaGro(group)=FeaGroPre(k);
   end
end


function r=CompareVector(Ia,Ib)
% Ia, Ib -- Respective column number of A and B matrices, which have a
%           same eigenvalue of A and B graphs
% r -- return 1 if same and unique; return 0 if same and not unique;
otherwise -1
%
   global FeaGro;
   global group;
   global Av;
   global Bv;
   global n;
   global TOL;
   M=zeros(n,1);
   M(Ia)=1;
   ABi=DeterminingSign(Av*M);
   [Va,VaI]=sort(ABi);
   M=zeros(n,1);
   M(Ib)=1;
   ABi=DeterminingSign(Bv*M);
   [Vb,VbI]=sort(ABi);
   c1=0.0;
   for j=1:n
      c1=c1+abs(Va(j)-Vb(j));
   end
   TemGroup1=0;
   if (abs(c1)<TOL)
      TemGroup=group;
      TemFeaGro=FeaGro;
      NodeMatching(Va,VaI,VbI);
      TemGroup1=group;
      TemFeaGro1=FeaGro;
      group=TemGroup;
      FeaGro=TemFeaGro;
   end
   c2=0.0;
   for j=1:n
      c2=c2+abs(Va(j)+Vb(n-j+1));
   end
   if (abs(c2)<TOL)
      for j=1:n
         VBI(j)=VbI(n+1-j);
      end
      NodeMatching(Va,VaI,VBI);
      for j=1:TemGroup1
         FeaGro(group+j)=TemFeaGro1(j);
      end
      group=group+TemGroup1;
   else
      if (abs(c1)<TOL)
         group=TemGroup1;
         FeaGro=TemFeaGro1;
      end
   end
   if (group==0 | (abs(c1)>TOL & abs(c2)>TOL))
      r=-1;
   else
      j=group;
      group=1;
      for i=2:j
         k=0;
         for l=1:group
            if (isequal(FeaGro(l).index,FeaGro(i).index))
               k=1;
               break;
            end
         end
         if (k==0)
            group=group+1;
            FeaGro(group)=FeaGro(i);
         end
      end
      if (FeaGro(1).number==n)
         r=1;
      else
         r=0;
      end
   end
```

```
function r=checkisomorphism(A,B,x)

global FeaGro;
global n;

pmatrix=zeros(n,n);
for i=1:n
  m=x(i,2);
  pmatrix(i,m)=1;
end
C=pmatrix*B*pmatrix';
if (isequal(A,C))
  r=1;
  % checked;
else
  r=-1;
  % failed
end
```

```
function x=DeterminingSign(matr)
% matr -- an eigenvector
% x -- the standard expression of the eigenvector in Eigensystem
approach
% The rule: (1) if the eigenvector is symmetrical to its negative
%               then do nothing
%           (2) for each eigenvector comparing the absolute values of
between the smallest
%               component and the largest component
%           (3) let the component having the largest absolute value as a
%               negative and change the sign of the eigenvector if
needed

global n;
global TOL;
aa=sort(matr);
bb=sort(-matr);
if (abs(aa-bb)<TOL)
    x=matr;
else
    sign=1;
    for i=1:n
        y=abs(aa(i)+aa(n-i+1));
        if (y>TOL)
            if (abs(aa(i))<abs(aa(n-i+1)))
                sign=-1;
            end
            break;
        end
    end
    x=sign*matr;
end


function r=NodeMatching(matr,IaV,IbV)
% matr -- a eigenvector of graph A reordered to the compared column
% IaV,IbV -- index matrices of the eigenvector matr for A and B,
respectively
% r -- return 1 if there is a one-to-one relationship; return 0
%      if can't make a decision; otherwise return -1
%
   global FeaGro;
   global group;
   global n;
   global TOL;
   x=matr(1);
   GroupN=1;
   Tem(1,IaV(1))=GroupN;
   Tem(2,IbV(1))=GroupN;
   for i=2:n
      if (abs(matr(i)-x)>TOL)
         x=matr(i);
         GroupN=GroupN+1;
      end
      Tem(1,IaV(i))=GroupN;
      Tem(2,IbV(i))=GroupN;
   end
   for l=1:group
      r=0;
      temp=Tem;
   FeaGroT=FeaGro(l).index;
   GroupN=1;
   FeaGroMid=zeros(2,n);

   for i=1:n
      if (temp(1,i)>=0)
         FeaGroPre=zeros(2,n);
         FeaGroCur=FeaGroPre;
         for j=1:n
            if (FeaGroT(1,j)==FeaGroT(1,i))
               FeaGroPre(1,j)=1;
            end
            if (FeaGroT(2,j)==FeaGroT(1,i))
               FeaGroPre(2,j)=1;
            end
            if (temp(1,j)==temp(1,i))
               FeaGroCur(1,j)=1;
            end
            if (temp(2,j)==temp(1,i))
               FeaGroCur(2,j)=1;
            end
         end
         TemResult=FeaGroPre & FeaGroCur;
         k1=1;
         k2=1;
         for j=1:n
            if (TemResult(1,j)>0)
               FeaGroMid(1,j)=GroupN;
               FeaGroT(1,j)=-1;
               temp(1,j)=-1;
               k1=k1+1;
            end
            if (TemResult(2,j)>0)
               FeaGroMid(2,j)=GroupN;
               FeaGroT(2,j)=-1;
               temp(2,j)=-1;
               k2=k2+1;
            end
         end
         if (k1==1 | k2==1)
            r=-1;
            break;
         end
         GroupN=GroupN+1;
      end
   end
   if (r==-1)
      FeaGro(l).number=-1;
   else
      FeaGro(l).number=GroupN-1;
      FeaGro(l).index=FeaGroMid;
   end
end
GroupN=group;
group=0;
for l=1:GroupN
   if (FeaGro(l).number~=-1)
      group=group+1;
      FeaGro(group)=FeaGro(l);
   end
end
```

APPENDIX B
PROGRAM FOR CANONICAL LABELING OF GRAPHS

The program in Appendix B is used for finding unique codes of graphs by canonically labeling the graphs. It consists of eight functions running in MATLAB 5.0 and later version, i.e., **LabelingGraph**, **QuadraticForm**, **recusive**, **CompareVector**, **checkisomorphism**, **CanonicalLabel**, **DeterminingSign**, and **NodeMatching**. The main function is **LabelingGraph**(A) in which augment A is adjacency matrix of a graph. The unique code for a graph is returned by running the program. A brief description for each function is introduced at the beginning of each function.

```
% Program for canonical labeling of graphs
% MATLAB M-file
% August 2003
% Main function LabelingGraph(A)
% -- A is the adjacency matrix of a graph


function Cano=LabelingGraph(A)
% Input --- the adjacency matrix of a graph
% Output -- canonical labeling and the canonical code

global FeaGro;
% FeaGro -- The matrix stored the node relationships of garph A and B
%           with constructing the same feature (component of
eigenvector)
%           into group
global group;
% number of feature groups
global n;
% n - node number of graph
global TOL;
global Bnode;
global total;
global Cano;

countTime1=cputime;
Cano='';
```

```
B=A;
i=size(A);
j=size(B);
if (i(1)~=i(2))
   disp('Must be square matrices!');
   error(' ');
end
if (~isequal(A,A'))
   disp('Must be symmetric matrices!');
   error(' ');
end
TOL = 0.0001;
group=1;
if (i(1)~=j(1))
   disp('They are not isomorphic graphs because of having different
nodes.');
else
   n = i(1);
   P=ones(n,1);
   Am=A*P;
   Bm=B*P;
   AAm=A;
   BBm=B;
   for k=1:n
      AAm(k,k)=n-Am(k);
      BBm(k,k)=n-Bm(k);
   end
   FeaGro(group).number=1;
   FeaGro(group).index=zeros(2,n);
   result=QuadraticForm(AAm,BBm);
   if (result==2)
      result=recusive(AAm,BBm,n,FeaGro,group);
   elseif (result==1)
      for j=1:group
         [DisFea,DisInx]=sort(FeaGro(j).index');
         r=checkisomorphism(A,B,DisInx);
         if (r==1)
           disp('Canonical Labeling of the graph');
            disp(DisInx');
            disp([0 1]*DisInx');
         end
      end
      disp('Canonical Code of the graph');
      disp(Cano);
   elseif (result==-1)
      disp('These two graphs are not isomorphic.');
   else
      disp('Oops! I can not make a decision now.');
      disp('An optional relationship of node numbers between graph A and
B:');
       [DisFea,DisInx]=sort(FeaGro(group).index');
       %disp(DisInx');
   end
end
countTime2=cputime;
disp('Time cost (seconds):');
disp(countTime2-countTime1);
```

```
function result=QuadraticForm(A,B)
% A,B -- adjacency matrices of graph A and B, respectively
% result -- return 1 if having one-to-one relationship, return 0 if
%      can't make a decision, return 2 if further test is needed,
%      otherwise -1
%
% Av,Bv -- eigenvector matrices of A and B graphs
%
global FeaGro;
global Av;
global Bv;
global n;
global TOL;
ABm = A';
[Av,Ad]=eig(ABm);
ABm = B';
[Bv,Bd]=eig(ABm);
M=ones(n,1);
ABd=Ad*M;
[Ad1,IAd]=sort(ABd);
ABd=Bd*M;
[Bd1,IBd]=sort(ABd);
if (abs(Ad1-Bd1)<TOL)
   x=1;
   r=2;
   for i=1:n-1
      if (abs(Ad1(i+1)-Ad1(x))>TOL)
         if (i==x)
            r=CompareVector(IAd(i),IBd(i));
            if (r<0)
               break;
            end
         end
         x=i+1;
      end
   end
   if (x==n & r>=0)
      r=CompareVector(IAd(n),IBd(n));
   end
   if (x==1 & r>=0)
      % only one eigenvalue, I guess that they are isomorphic
      result=1;
   else
      if (r==2)
         % every eigenvalue has at least a repeated root
         result=0;
      elseif (r==1)
         % isomorphic
         result=1;
      elseif (r==-1)
         % non-isomorphic
         result=-1;
      else
         % Further test is required
         result=2;
      end
   end
else
```

```
   % non-isomorphic: eigenvalues are different
   result=-1;
end
```

```
function r=recusive(A,B,n,FeaGroPre,Fgroup)
% A,B ---- adjacency matrices of graph A and B
% n ------ number of vertices
% FeaGroPre ---- mapping relationships in feature groups
% Fgroup -- number of feature groups

global FeaGro;
global group;
global Bnode;
global total;

for k=1:Fgroup
   AA=A;
   BB=B;
   group=1;
   FeaGro(group)=FeaGroPre(k);
   [TemFeaGro,Ifg]=sort(FeaGroPre(k).index');
   j=1;
   t(1)=n;
   t(2)=n;
   for i=2:n
      if (TemFeaGro(i,1)==TemFeaGro(i-1,1))
         j=j+1;
      else
         if (j>1)
            if (t(1)>j)
               t(1)=j; % the number of vertices in a feature group
               t(2)=i-1; % the location of the final vertex in a feature
group
            end
            j=1;
         end
      end
   end
   % Compare the last feature group
   if (j>1)
      if (t(1)>j)
         t(1)=j; % the number of vertices in a feature group
         t(2)=i; % the location of the final vertex in a feature group
      end
   end
   mm=t(2)-t(1)+1; % Location of the first vertex in the group
   ma=Ifg(mm,1);
   % Change an element of matrix A by plusing ma at 'ma'
   AA(ma,ma)=AA(ma,ma)+ma;
   for j=1:t(1)
      mb=Ifg(mm+j-1,2);
      % Plus the element of matrix B by ma at 'mb'
      BB(mb,mb)=BB(mb,mb)+ma;
      r=QuadraticForm(AA,BB);
      if (r==2)
```

```
         r=recusive(AA,BB,n,FeaGro,group);
      else
         if (r==1)
            for l=1:group
               [DisFea,DisInx]=sort(FeaGro(l).index');
               r=checkisomorphism(A,B,DisInx);
               if (r==1)
                  Bnode(total).index=[0 1]*DisInx';
%                  disp(DisInx');
                  % disp(Bnode(total).index);
                  total=total+1;
               end
            end
         end
      end
      BB(mb,mb)=BB(mb,mb)-ma;
      group=1;
      FeaGro(group)=FeaGroPre(k);
   end
end
```

```
function r=checkisomorphism(A,B,x)

global FeaGro;
global n;
global Cano;

pmatrix=zeros(n,n);
pmatrix1=zeros(n,n);
for i=1:n
    columnA=x(i,1);
    columnB=x(i,2);
    pmatrix(i,columnB)=1;
    pmatrix1(i,columnA)=1;
end
Aa=pmatrix1*A*pmatrix1';
Bb=pmatrix*B*pmatrix';
if (isequal(Aa,Bb))
    r=1;
    Cano=CanonicalLabel(Aa);
else
    r=-1;
end
```

```
function r=CompareVector(Ia,Ib)
% Ia, Ib -- Respective column number of A and B matrices, which have a
%           same eigenvalue of A and B graphs
% r -- return 1 if same and unique; return 0 if same and not unique;
otherwise -1
%
```

```
global FeaGro;
global group;
global Av;
global Bv;
global n;
global TOL;
M=zeros(n,1);
M(Ia)=1;
ABa=Av*M;
sign1=DeterminingSign(ABa);
ABa=sign1*ABa;
M=zeros(n,1);
M(Ib)=1;
ABb=Bv*M;
sign2=DeterminingSign(ABb);
ABb=sign2*ABb;
[Va,VaI]=sort(ABa);
[Vb,VbI]=sort(ABb);
c1=0.0;
for j=1:n
   c1=c1+abs(Va(j)-Vb(j));
end
TemGroup1=0;
if (abs(c1)<TOL)
   TemGroup=group;
   TemFeaGro=FeaGro;
   NodeMatching(Va,VaI,VbI);
   TemGroup1=group;
   TemFeaGro1=FeaGro;
   group=TemGroup;
   FeaGro=TemFeaGro;
end
[Va,VaI]=sort(-ABa);
c2=0.0;
for j=1:n
   c2=c2+abs(Va(j)-Vb(j));
end
if (abs(c2)<TOL)
   NodeMatching(Va,VaI,VbI);
   for j=1:TemGroup1
      FeaGro(group+j)=TemFeaGro1(j);
   end
   group=group+TemGroup1;
else
   if (abs(c1)<TOL)
      group=TemGroup1;
      FeaGro=TemFeaGro1;
   end
end
if (group==0 | (abs(c1)>TOL & abs(c2)>TOL))
   r=-1;
else
   j=group;
   group=1;
   for i=2:j
      k=0;
      for l=1:group
         if (isequal(FeaGro(l).index,FeaGro(i).index))
            k=1;
            break;
         end
```

```
            end
            if (k==0)
               group=group+1;
               FeaGro(group)=FeaGro(i);
            end
         end
         if (FeaGro(1).number==n)
            r=1;
         else
            r=0;
         end
      end
```

```
function pCano=CanonicalLabel(matr)
% matr -- The adjacency matrix which is the canonical labeling
% Cano -- the canonical labeling

global n;
global Cano;

pmatrix=zeros(n,n);
for i=1:n
    pmatrix(i,n-i+1)=1;
end
matr1=pmatrix*matr*pmatrix';
r=0;
for i=1:n-1
    for j=i+1:n
        if (matr(i,j)~=matr1(i,j))
            r=1;
            break;
        end
    end
    if (r==1)
        break;
    end
end
if (r==1)
    if (matr(i,j)>matr1(i,j))
        matr=matr1;
    end
end
m=n*(n-1)/2;
r=4-mod(m,4);
sym=zeros(1,m+r);
ind=r+1;
for i=1:n-1
    for j=i+1:n
        sym(1,ind)=matr(i,j);
        ind=ind+1;
    end
end
pCano='';
for i=1:4:m+r
    x=8*sym(1,i)+4*sym(1,i+1)+2*sym(1,i+2)+sym(1,i+3);
    if (x<10)
        pCano=strcat(pCano,int2str(x));
    elseif (x==10)
        pCano=strcat(pCano,'A');
    elseif (x==11)
        pCano=strcat(pCano,'B');
    elseif (x==12)
        pCano=strcat(pCano,'C');
    elseif (x==13)
        pCano=strcat(pCano,'D');
    elseif (x==14)
        pCano=strcat(pCano,'E');
    else
        pCano=strcat(pCano,'F');
    end
end
r=fix((m+3)/4);
if (isempty(Cano))
    Cano=pCano;
else
    for i=1:r
       if (pCano(i)~=Cano(i))
          if (pCano(i)>Cano(i))
             pCano=Cano;
          end
          break;
       end
    end
end
```

```
function sign=DeterminingSign(matr)
% matr -- an eigenvector
% sign -- the sign of the standard expression of the eigenvector in
Eigensystem approach

% The rule: (1) if the eigenvector is symmetrical to its negative
%               then do nothing
%           (2) for each eigenvector comparing the absolute values of
between the smallest
%               component and the largest component
%           (3) let the component having the largest absolute value as a
%               negative and change the sign of the eigenvector if
needed

global n;
global TOL;

aa=sort(matr);
bb=sort(-matr);
if (abs(aa-bb)<TOL)
    sign=1;
else
    sign=1;
    for i=1:n
        y=abs(aa(i)+aa(n-i+1));
        if (y>TOL)
           if (abs(aa(i)-aa(n-i+1))<TOL)
              if (aa(i)>0)
                 sign=-1;
```

```
                end
             else
                if (abs(aa(i))<abs(aa(n-i+1)))
                   sign=-1;
                end
             end
             break;
          end
      end
end
```

```
function r=NodeMatching(matr,IaV,IbV)
% matr -- a eigenvector of graph A reordered to the compared column
% IaV,IbV -- index matrices of the eigenvector matr for A and B,
respectively
% r -- return 1 if there is a one-to-one relationship; return 0
%      if can't make a decision; otherwise return -1
%
   global FeaGro;
   global group;
   global n;
   global TOL;
   x=matr(1);
   GroupN=1;
   Tem(1,IaV(1))=GroupN;
   Tem(2,IbV(1))=GroupN;
   for i=2:n
      if (abs(matr(i)-x)>TOL)
         x=matr(i);
         GroupN=GroupN+1;
      end
      Tem(1,IaV(i))=GroupN;
      Tem(2,IbV(i))=GroupN;
   end
   for l=1:group
      r=0;
      temp=Tem;
      FeaGroT=FeaGro(l).index;
      GroupN=1;
      FeaGroMid=zeros(2,n);
      if (FeaGro(l).number==1)
         FeaGroT=ones(2,n);
      end
      for i=1:FeaGro(l).number
         tmax=0;
         tmin=n;
         FeaGroCur=zeros(2,n);
         for j=1:n
            if (FeaGroT(1,j)==i)
               FeaGroCur(1,j)=temp(1,j);
               if (temp(1,j)>tmax)
                  tmax=temp(1,j);
               end
               if (temp(1,j)<tmin)
                  tmin=temp(1,j);
               end
            end
            if (FeaGroT(2,j)==i)
               FeaGroCur(2,j)=temp(2,j);
            end
         end
         for ii=tmin:tmax
            k1=0;
            k2=0;
            for j=1:n
               if (FeaGroCur(1,j)==ii)
                  FeaGroMid(1,j)=GroupN;
                  k1=k1+1;
               end
               if (FeaGroCur(2,j)==ii)
                  FeaGroMid(2,j)=GroupN;
                  k2=k2+1;
               end
            end
            if (k1~=k2)
               r=-1;
               break;
            end
            GroupN=GroupN+1;
         end
         if (r==-1)
            break;
         end
      end

      if (r==-1)
         FeaGro(l).number=-1;
      else
         FeaGro(l).number=GroupN-1;
         FeaGro(l).index=FeaGroMid;
      end
   end
   GroupN=group;
   group=0;
   for l=1:GroupN
      if (FeaGro(l).number~=-1)
         group=group+1;
         FeaGro(group)=FeaGro(l);
      end
   end
```

APPENDIX C
PROGRAM FOR ENUMERATION OF MECHANISMS

The program in Appendix C is used for listing all alternative mechanisms meeting design constraints of mechanisms, i.e., the number of linkages and the number of degrees of freedom. These alternative mechanisms have been converted into graphs and are stored in output files with formats of adjacency matrices. The graphs stored in output files are then uniquely coded by using the program in Appendix B to enumerate all distinct graphs (mechanisms). The class name of the program is **E_mechanism**. It is written in Java code and must be compiled in JDK 1.2 or later version environment before running it. A brief description for each method in the class is introduced at the beginning of each method.

```
// Program for enumeration of kinematic chains meeting constraints
// JAVA
// August 2003
// Main function E_mechanism


import java.io.*;
import java.lang.*;
import java.util.*;

class E_mechanism
{
        public static int graph=0;
       public static File outputFile;
       public static FileWriter out;

         public static void main(String[] args) throws IOException
         {
                 // dof -- Degree of Freedom
                 // linkN -- number of links
```

```
                // groupN -- the maximum joint values that the link
could have
                //             2 binary; 3 ternary; 4 quaternary; ...
                // linkType[] -- number of the ith link for
i=2,3,..,groupN.
                // linkG[i][j] -- number of the jth kind of link type at
the ith possible group
                // typeN -- number of possible groups where various
types of links combine together
                int [] linkType=new int[20];
                int [][] temType=new int[50][3];
                int [][] linkG=new int [200][20];
                int [][] curType=new int [50][3];
                int [][] graphAAM=new int [50][50];
                int x,dof,linkN,groupN,typeN;
                String str;

                BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
                System.out.print("\nDegree of Freedom = ");
                str=stdin.readLine();
                dof=Integer.parseInt(str,10);
                System.out.print("\n\nNumber of links = ");
                str=stdin.readLine();
                linkN=Integer.parseInt(str,10);
                System.out.print("\n\nFilename of kinematic chains: ");
                str=stdin.readLine();
                System.out.println();
                groupN=findJointValues(dof,linkN);
                findLinkNumbers(dof,linkN,groupN,linkType);
                String s=new String(enuNumber(linkType,groupN,linkN));
                typeN=findLinkGroup(groupN,s,dof,linkG);
                System.out.println();
                System.out.print(typeN);
                System.out.println(" file(s) saving potential kinematic
chains will be created.\n");
            for (int i=1;i<=typeN;i++)
                {
                     s=new String(str);
                     s=s.concat("_");
                     s=s.concat(String.valueOf(i));
                     outputFile= new File(s.concat(".kc_"));
                  out = new FileWriter(outputFile);
                  out.write(linkN);
                     out.write(groupN+100);
                        for (int j=2;j<=groupN;j++)
                        {
                                temType[j-1][0]=j;
                                temType[j-1][1]=-linkG[i][j];
                                temType[j-1][2]=0;
                        out.write(linkG[i][j]);
                        }
                        graphAAM[0][1]/=2;
                        for (x=groupN;x>=2;x--)
                                if (temType[x-1][1]!=0) break;
                        // pick up one link with the maximum joint
values
                        temType[x-1][1]++;
                        graphAAM[0][0]=0; //counting face number of
graph
                        graphAAM[1][1]=x;
                        int
cur_groupN=waitConnectingList(temType,groupN,curType);

enumerationKC(graphAAM,1,1,x,curType,cur_groupN,dof);
                        out.close();
System.out.println(graph);
                }
            System.out.println("Total graphs (isomorphic and
nonisomorphic): "+graph);
        }

    // Determining the maximum joint values of the link that a kinematic
chain could have
        public static int findJointValues(int dof,int linkN)
        {
                int groupN;
                if (dof>=2) {
                        int i=linkN-dof-1;
                        int j=(linkN+dof-1)/2;
                        if (i>j) groupN=j;
                        else groupN=i;
                }
                else groupN=(linkN-dof+1)/2;
                return groupN;
        }

        // Determining possible number of each link in a kinematic chain
        public static void findLinkNumbers(int dof,int linkN,int
groupN,int [] linkType)
        {
                for (int i=3;i<=groupN;i++)
                        linkType[i]=(linkN-dof-3)/(i-2);
                linkType[2]=linkN;
        }

        // Recurisive procedure for enumerating all possible link groups
        public static String enuNumber(int L[],int groupN,int linkN)
        {
                int n;
                String r;

                r=" L".concat(String.valueOf(groupN));
                r=r.concat(" ");
                if (groupN==2)
                {
                        if (L[groupN]<linkN) r=r.concat(String.valueOf(-
1));

                        else r=r.concat(String.valueOf(linkN));
                        return r;
                }
                if (linkN==0)
                {
                        r=r.concat(String.valueOf(0));
                        return r;
                }
                if (L[groupN]<linkN) n=L[groupN];
                else n=linkN;
                String Num=new String();
                for (int i=0;i<=n;i++)
                {
                        Num=Num.concat(r);
                        Num=Num.concat(String.valueOf(i));
                        Num=Num.concat(enuNumber(L,groupN-1,linkN-i));
                }
                return Num;
        }

        // Determining the number of each link in a possible kinematic
chain
        public static int findLinkGroup(int groupN,String str,int
dof,int [][] linkG)
        {
                int row,col,preC;
                int [] jointV=new int[200];
                String s=new String();

                row=1;
                col=groupN;
                preC=col+1;
                StringTokenizer st = new
StringTokenizer(str.substring(1));
                while (st.hasMoreTokens()) {
                         s=st.nextToken();
                         if (s.charAt(0)=='L')
                         {
                                col=Integer.parseInt(s.substring(1),10);
                                if (col>=preC) {
                                        if (jointV[row]==dof+3) {
                                                row++;
                                                for (int
i=groupN;i>col;i--)
                                                {

linkG[row][i]=linkG[row-1][i];

jointV[row]=jointV[row]+(3-i)*linkG[row][i];
                                                }
                                        }
                                        else {
                                                jointV[row]=0;
                                                for (int
i=groupN;i>col;i--)

jointV[row]=jointV[row]+(3-i)*linkG[row][i];
                                        }
                                }
                                preC=col;
                         }
                         else {
                                linkG[row][col]=Integer.parseInt(s,10);
                                if (linkG[row][col]==-1) row--;
                                else jointV[row]=jointV[row]+(3-
col)*linkG[row][col];
                         }
                }
                if (jointV[row]!=dof+3) row--;
                return row;
        }

        // Enumerating all possible kinematic chains
        public static void enumerationKC(int[][] AAM,int aAm,int
l_num,int r_joint,int [][] preType,int cur_groupN,int dof) throws
IOException
        {
                int i,j,n,num,aam,gNumber;
                int [][] linkGroup=new int [100][20];
                int [][] curType=new int [50][3];
                int [][] ncurType=new int [50][3];
                int [][] graphAAM=new int[50][50];

                // linkGroup[i][j] -- number of the link curType[j][] in
the ith group
                String str=new String();

                str=chainEnumeration(preType,cur_groupN,r_joint);
              if (!str.equals("-1"))
              {
                gNumber=findChainGroup(cur_groupN,str,linkGroup);
                for (i=1;i<=gNumber;i++)
                  {
                      for (int ii=1;ii<=cur_groupN;ii++)
                     {
                           curType[ii][0]=preType[ii][0];
                             curType[ii][1]=preType[ii][1];
                      curType[ii][2]=preType[ii][2];
                      }
                     for (int ii=1;ii<=aAm;ii++)
                    {
                             for (int jj=1;jj<=aAm;jj++)
                                 graphAAM[ii][jj]=AAM[ii][jj];
                    }
                      num=cur_groupN;
                   aam=aAm;
                   for (j=1;j<=cur_groupN;j++)
                        {
                             n=linkGroup[i][j];
                      if (n==0) continue;
                           int y=curType[j][1];
                             if (y>0) // It already joins into the
chain
                             {
                                   curType[j][0]-=n;
                                     graphAAM[l_num][y]=n;
                                    graphAAM[y][l_num]=n;
                           curType[j][2]=0;
                           int
jj=checkZeroDOF(graphAAM,aam,l_num,y,dof);
                           if (jj==1) break;
                           jj=checkZeroDOF2(graphAAM,aam,l_num,y);
                           if (jj==1) break;
}
                             else { // It doesn't yet join into
the chain
                           int z=aam+1;
                           for (int k=1;k<=n;k++)
                                 {
                                        aam++;
                                       num++;

graphAAM[aam][aam]=curType[j][0];
graphAAM[l_num][aam]=1;
                                                    graphAAM[aam][l_num]=1;

   curType[num][0]=curType[j][0]-1;
                                                      curType[num][1]=aam;
                                           if (n>1) curType[num][2]=z;
                                           else curType[num][2]=0;
                                               }
                                             curType[j][1]+=n;
                                      }
                                 }
                     if (j<=cur_groupN) continue;
                     num=waitConnectingList(curType,num,ncurType);
                        if (num<2)
                          {
                                 if (num==0)
                          {
                          /* if (isPlanarKC(graphAAM,aam)==0)
*/saveGraphAAM(graphAAM,aam);
                          }
                        }
                          else {
                                 for (j=1;j<=num;j++)
                                       if (ncurType[j][0]>0 &&
ncurType[j][1]>0) break;
                                  if (j<=num)
                                 {
                                       int i_num=ncurType[j][1];
                                         int c_l_num=ncurType[j][1];
                                        int c_r_num=ncurType[j][0];
                                       ncurType[j][2]=0;

pickupLink(graphAAM,aam,i_num,ncurType,num);
                                        int [][] nncurType=new
int[50][3];

num=waitConnectingList(ncurType,num,nncurType);

enumerationKC(graphAAM,aam,c_l_num,c_r_num,nncurType,num,dof);
                                 }
                          }
                 }
            }
        }

        public static void pickupLink(int[][] graphAAM,int aam,int
k,int[][] curType,int num)
        {
                int i,j;
                for (j=1;j<=aam;j++)
                {
                        if (graphAAM[k][j]>0)
                        {
                                for (i=1;i<=num;i++)
                                {
                                        if (curType[i][1]==j)
                                        {
                                                curType[i][0]=0;
                                                break;
                                        }
                                }
                        }
                }
        }


        // Removing completely connected links from the waiting list for
connection
        // curType[][0] - remain of joint values; curType[][1] - link
No.
        // if (curType[][1]<0) then it means the link doesn't yet join
into the chain, and
        // -curType[][1] represents the number of this kind of link
        public static int waitConnectingList(int[][] preType,int
preNum,int[][] curType)
        {
                int curNum=0;

                for (int j=1;j<=preNum;j++)
                {
                        if (preType[j][1]!=0 && preType[j][0]!=0)
                        {
                                curNum++;
                                curType[curNum][0]=preType[j][0];
                                curType[curNum][1]=preType[j][1];
                        curType[curNum][2]=preType[j][2];
                        }
                }
                return curNum;
        }

        // Recurisive procedure for enumerating all possible chains
        public static String chainEnumeration(int[][] curType,int
indexN,int linkN)
        {
                int n,ii,jj,kk;
                int [] L=new int[20];
            int [] mark=new int[20];
            int [][] linkGroup=new int[100][20];
            String str;
            ii=1;
            kk=0;
                for (int i=1;i<=indexN;i++)
                {
                        if (curType[i][1]<0)
                  {
                     ii++;
                     L[ii]=-curType[i][1]; // nonconnected link
                     kk-=curType[i][1];
                     mark[i]=ii;
                  }
                        else if (curType[i][2]>0)
                  {
                     for (jj=1;jj<i;jj++)
                     {
                        if (curType[i][2]==curType[jj][2])
                        {
                           int k=mark[jj];
                           L[k]++;
                           kk++;
                           mark[i]=k;
```

```
                    break;
                }
            }
            if (jj==i)
            {
                ii++;
                L[ii]=1;
                kk++;
                mark[i]=ii;
            }
        }
        else {
            ii++;
            L[ii]=1;
            kk++;
            mark[i]=ii;
        }
    }
    if (kk<linkN) return ("-1");
    str=enuNumber(L,ii,linkN);
int gNumber=findChainGroup(ii-1,str,linkGroup);
int [][] tmp=new int [100][20];
for (int i=1;i<=gNumber;i++)
{
    for (int j=1;j<ii;j++)
    {
        for (jj=1;jj<=indexN;jj++)
        {
            if (mark[jj]==j+1)
            {
                if (curType[jj][2]>0)
                {
                    int x=linkGroup[i][j];
                    if (x==0) break;
                    for (kk=1;kk<=indexN;kk++)
                    {
                        if (curType[kk][2]==curType[jj][2])
                        {
                            tmp[i][kk]=1;
                            x--;
                        }
                        if (x==0) break;
                    }
                }
                else tmp[i][jj]=linkGroup[i][j];
                break;
            }
        }
    }
}
String s=new String();
     for (int i=1;i<=gNumber;i++)
     {
    for (int j=indexN;j>=1;j--)
    {
        s=s.concat(" L");
        s=s.concat(String.valueOf(j+1));
        s=s.concat(" ");
        s=s.concat(String.valueOf(tmp[i][j]));
    }
     }
```

```
            return s;
        }

        // find a kinematic chain group
        public static int findChainGroup(int cur_groupN,String
str,int[][] linkGroup)
        {
                int row,col,preC,mark;
                String s=new String();

                mark=1;
            row=1;
                col=cur_groupN;
                preC=col+1;
                StringTokenizer st = new
StringTokenizer(str.substring(1));
                while (st.hasMoreTokens()) {
                          s=st.nextToken();
                          if (s.charAt(0)=='L')
                          {
                                 col=Integer.parseInt(s.substring(1),10)-
1;
                                 if (col>=preC) {
                                         if (mark==1)
                               {
                                     row++;
                                     for (int i=cur_groupN;i>col;i--)

   linkGroup[row][i]=linkGroup[row-1][i];
                               }
                               else mark=1;
                                 }
                                 preC=col;
                          }
                          else {

linkGroup[row][col]=Integer.parseInt(s,10);
                                 if (linkGroup[row][col]==-1) mark=0;
                          }

                }
                return row;
        }

        // Check if this connection will result in a 0 DOF among links
      // return 1 -- a 0 DOF; return 0 -- no 0 DOF
      public static int checkZeroDOF(int[][] graphAAM,int aam,int
prenum,int curnum,int deg)
      {
            int i,j,x,y,tri;

            // checkining if there is a triangle among three links
            for (i=1;i<=aam;i++)
            {
                  j=graphAAM[prenum][i]+graphAAM[curnum][i];
                  if (j==2 && i!=prenum && i!=curnum) break;
            }
            if (i<=aam) return (1);
            // checkining if there are more than two connections,
            // each of which is jointed by one link, between the two
links
```

```
            for (i=1;i<=aam;i++)
            {
                  if (prenum!=i)
                  {
                        tri=0;
                        for (j=1;j<=aam;j++)
                        {
                              if (j!=prenum && j!=i)
                              {
   x=graphAAM[prenum][j]+graphAAM[i][j];
                                    if (x==2) tri++;
                              }
                        }
                        if (tri>=3) return (1);
                  }
                  if (curnum!=i)
                  {
                        tri=0;
                        for (j=1;j<=aam;j++)
                        {
                              if (j!=curnum && j!=i)
                              {
   x=graphAAM[curnum][j]+graphAAM[i][j];
                                    if (x==2) tri++;
                              }
                        }
                        if (tri>=3) return (1);
                  }

            }
            // check if there is a zero DOF for 7 linked chains
            for (i=1;i<=aam;i++)
            {
               for (j=1;j<=aam;j++)
               {
                  if (i!=j && graphAAM[i][j]==1)
                  {
                     int ii=isZeroDOF(graphAAM,aam,i,j);
                     if (ii==1) return (1);
                  }
               }
            }
            // check if the DOF of partialy binary chains is more than
the DOF of the mechanism
            for (i=1;i<=aam;i++)
            {
               if (graphAAM[i][i]==2 && graphAAM[i][0]!=-1)
               {
                  int bi=1;
                  graphAAM[i][0]=-1;
                  for (j=1;j<=aam;j++)
                     if (graphAAM[i][j]==1)
bi=binaryChains(graphAAM,aam,j,i,bi);
                    if (bi>=deg+2 && bi<aam) return (1);
               }
            }
            // check if DOF=0 for the kinematic chains with the joint
between prenum & curnum
            int [] temp=new int[50];
            temp[prenum]=1;
            temp[curnum]=1;
            for (i=1;i<=aam;i++)
            {
               if ((graphAAM[prenum][i]!=0 || graphAAM[curnum][i]!=0) &&
i!=prenum && i!=curnum)
               {
                  temp[i]=1;
                  if (i!=1) nextLink(graphAAM,aam,temp,i);
               }
            }
            int DOF=0;
            for (i=1;i<=aam;i++)
            {
               if (temp[i]==1)
               {
                  tri=0;
                  for (j=1;j<=aam;j++)
                     if (graphAAM[i][j]>0) tri++;
                  if (tri<=2) temp[i]=0; // the ith link has no closed
loop
               }
            }
            for (i=1;i<=aam;i++)
            {
               if (temp[i]==0) continue;
               int tmp=0;
               for (j=1;j<=aam;j++)
                  if (i!=j) tmp+=graphAAM[i][j]*temp[j];
               DOF+=3-tmp; // (3-tmp)*L=dof+3
            }
            if (DOF<=3) return (1);
            else return (0);
      }

      // Check if this connection will result in a 0 DOF among links
      // return 1 -- a 0 DOF; return 0 -- no 0 DOF
      public static int checkZeroDOF2(int[][] graphAAM,int aam,int
prenum,int curnum)
      {
         int[][] temp=new int[50][50];
         int[] sign=new int[50];
         int[] sign1=new int[50];
         int i,j,k,n,tmp;

         n=2;
         sign[prenum]=1;
         sign[curnum]=1;
         while (n!=aam)
         {
            for (i=1;i<=aam;i++)
            {
               for (j=1;j<=aam;j++)
               {
                  if (i!=j) temp[i][j]=graphAAM[i][j];
                  else temp[i][j]=0;
               }
            }
            for (i=1;i<=aam;i++)
               sign1[i]=sign[i];
            for (i=1;i<=aam;i++)'
```

```
{
   if (sign[i]==1)
   {
      for (j=1;j<=aam;j++)
      {
         if (temp[i][j]==1 && sign[j]==0 && sign1[j]==0)
         {
            sign1[j]=1;
            n++;
         }
      }
   }
}
for (i=1;i<=aam;i++)
   sign[i]=sign1[i];
for (i=1;i<=aam;i++)
{
   if (sign[i]==0)
   {
      for (j=1;j<=aam;j++)
      {
         temp[i][j]=0;
         temp[j][i]=0;
      }
   }
}
int r=1;
while (r==1)
{
   r=0;
   for (i=1;i<=aam;i++)
   {
      tmp=0;
      for (j=1;j<=aam;j++)
         tmp+=temp[i][j];
      if (tmp==1)
      {
         r=1;
         for (k=1;k<=aam;k++)
         {
            temp[i][k]=0;
            temp[k][i]=0;
         }
      }
   }
}
int DOF=0;
int t=1;
for (i=1;i<=aam;i++)
{
   tmp=0;
   for (j=1;j<=aam;j++)
      tmp+=temp[i][j];
   if (tmp!=0)
   {
      DOF+=3-tmp;
      t=0;
   }
}
if (DOF<=3 && t==0)
{
```

```
                n=-1;
                break;
            }
        }
        if (n==-1) return (1);
        else return (0);
    }


    public static int binaryChains(int[][] graphAAM,int aam,int
cur,int pre,int bi)
    {
        int i;

        if (graphAAM[cur][cur]!=2 || graphAAM[cur][0]==-1) return (bi);
        for (i=1;i<=aam;i++)
            if (graphAAM[cur][i]==1 && i!=pre) break;
        graphAAM[cur][0]=-1;
        bi++;
        bi=binaryChains(graphAAM,aam,i,cur,bi);
        return (bi);
    }


    public static int isZeroDOF(int[][] graphAAM,int aam,int node1,int
node2)
    {
        int i,j,k1,k2;

        for (i=1;i<=aam;i++)
        {
            k1=0;
            k2=0;
            if (i!=node1 && i!=node2)
            {
                for (j=1;j<=aam;j++)
                {
                    if (j!=node1 && j!=node2 && j!=i)
                    {
                        if (graphAAM[i][j]>0 && graphAAM[node1][j]>0) k1++;
                        if (graphAAM[i][j]>0 && graphAAM[node2][j]>0) k2++;
                    }

                }
                if (k1>=2 && k2>=2) return (1);
            }
        }
        return (0);
    }


    // finding next link
    public static void nextLink(int[][] graphAAM,int aam,int[]
temp,int k)
    {
        for (int i=1;i<=aam;i++)
        {
            if (graphAAM[k][i]!=0 && temp[i]==0)
            {
                temp[i]=1;
                if (i!=1) nextLink(graphAAM,aam,temp,i);
            }
        }
    }


    // Check if the mechanism is a planar kinematic chains
    public static int isPlanarKC(int[][] gAAM,int index)
    {
        int[][] graphAAM=new int[50][50];
        int[] number=new int[20];
        int[][] temp=new int[20][20];
        int[] L=new int[20];
        int[] kk=new int[6];
        int i,j,k,x,y,xx,yy,num,aam;
        String str=new String();

        for (i=1;i<=index;i++)
        {
            for (j=1;j<=index;j++)
                graphAAM[i][j]=gAAM[i][j];
        }
        for (i=1;i<=index;i++)
        {
            if (graphAAM[i][i]==2)
            {
                int n=0;
                for (j=1;j<=index;j++)
                {
                    if (graphAAM[i][j]==1)
                    {
                        n++;
                        kk[n]=j;
                    }
                }
                x=kk[1];
                y=kk[2];
                graphAAM[i][0]=-1;
                graphAAM[x][y]=1;
                graphAAM[y][x]=1;
                graphAAM[i][x]=0;
                graphAAM[x][i]=0;
                graphAAM[i][y]=0;
                graphAAM[y][i]=0;
            }
        }
        int col=0;
        aam=0;
        for (i=1;i<=index;i++)
        {
            if (graphAAM[i][0]==-1) continue;
            aam++;
            col=0;
            for (j=1;j<=index;j++)
            {
                if (graphAAM[j][0]==-1) continue;
                col++;
                graphAAM[aam][col]=graphAAM[i][j];
            }
        }
        for (i=0;i<20;i++)
            L[i]=1;
```

```
k=0;
for (i=1;i<=aam;i++)
{
   if (graphAAM[i][i]>3)
   {
      k++;
      number[k]=i;
   }
}
if (k>=5) // K5
{
   str=enuNumber(L,k+1,5);
   num=findChainGroup(k,str,temp);
   for (i=1;i<=num;i++)
   {
      int kkk=0;
      for (j=1;j<=k;j++)
      {
         if (temp[i][j]==1)
         {
            kk[kkk]=number[j];
            kkk++;
         }
      }
      for (x=0;x<5;x++)
      {
         xx=kk[x];
         for (y=0;y<5;y++)
         {
            yy=kk[y];
            if (graphAAM[xx][yy]==0) break;
         }
         if (y<5) break;
      }
      if (x==5) return (-1);
   }
}
for (i=1;i<=aam;i++)
{
   if (graphAAM[i][i]==3)
   {
      k++;
      number[k]=i;
   }
}
if (k>=6) // K3,3
{
   str=enuNumber(L,k+1,6);
   num=findChainGroup(k,str,temp);
   int[] k1=new int[6];
   int[] k2=new int[6];
   int kk1,kk2;
   for (i=1;i<=num;i++)
   {
      int kkk=0;
      for (j=1;j<=k;j++)
      {
         if (temp[i][j]==1)
         {
            kk[kkk]=number[j];
            kkk++;
```

```
                    }
                }
                int[][] tmp=new int[20][20];
                str=enuNumber(L,6,2);
                int nn=findChainGroup(5,str,tmp);
                for (j=1;j<=nn;j++)
                {
                   kk1=1;
                   kk2=0;
                   k1[kk1]=kk[0];
                   for (int jj=1;jj<=5;jj++)
                   {
                      if (tmp[j][jj]==1)
                      {
                         kk1++;
                         k1[kk1]=kk[jj];
                      }
                      else {
                         kk2++;
                         k2[kk2]=kk[jj];
                      }
                   }
                   int ii,jj;
                   for (ii=1;ii<=3;ii++)
                   {
                      xx=k1[ii];
                      for (jj=1;jj<=3;jj++)
                      {
                         yy=k2[jj];
                         if (graphAAM[xx][yy]==0) break;
                      }
                      if (jj<=3) break;
                   }
                   if (ii>3) return (-1);
                }
             }
          }
          return (0);
       }


       // Save AAM of a graph (kinematic chains)
         public static void saveGraphAAM(int[][] graphAAM,int aam) throws
IOException
         {
             graph++;
                 for (int i=1;i<=aam;i++)
                 {
                         for (int j=1;j<=aam;j++)
                         {
                                 if (i==j) out.write(aam-graphAAM[i][j]);
                                 else out.write(graphAAM[i][j]);
                         }
                 }
         }
}
```

APPENDIX D
PROGRAM FOR ENUMERATION OF THE ALKANE SERIES

The program in Appendix D is used for listing all alternative structures of the Alkane series for a given number of carbon atoms. These alternative structures have been converted into graphs and are stored in output files with formats of adjacency matrices. The graphs stored in output files are then uniquely coded by using the program in Appendix B to enumerate all distinct graphs (molecular structures). The class name of the program is **E_Alkane**. It is written in Java code and must be compiled in JDK 1.2 or later version environment before running it. A brief description for each method in the class is introduced at the beginning of each method.

```
// Program for enumeration of the Alkane Series meeting constraints
// JAVA
// August 2003
// Main function E_Alkane


import java.io.*;
import java.lang.*;
import java.util.*;

class E_Alkane
{
        public static int graph=0;
            public static File outputFile;
            public static FileWriter out;

        public static void main(String[] args) throws IOException
        {
                // linkN -- number of carbons
                // groupN -- the maximum carbon-carbon bonds that the
carbon could have
                // linkType[] -- number of the ith carbon for
i=2,3,..,groupN.
```

```
                // linkG[i][j] -- number of the jth kind of carbon type
at the ith possible group
                // typeN -- number of possible groups where various
types of carbons combine together
                int [] linkType=new int[20];
                int [][] temType=new int[50][3];
                int [][] linkG=new int [200][20];
                int [][] curType=new int [50][3];
                int [][] graphAAM=new int [50][50];
                int x,linkN,groupN,typeN;
                String str;

                groupN=4;
                BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
                System.out.print("\n\nNumber of Carbon Atoms = ");
                str=stdin.readLine();
                linkN=Integer.parseInt(str,10);
                System.out.print("\n\nFilename of Isomers of Alkane
Series: ");
                str=stdin.readLine();
                System.out.println();
                findChemicalBonds(linkN,linkType);
                String s=new
String(enuNumber(linkType,groupN,linkN,2*linkN+2));
                typeN=findBondGroup(groupN,s,linkG);
                        System.out.println();
                System.out.print(typeN);
                System.out.println(" file(s) saving potential isomers
are created.\n");
                        for (int i=1;i<=typeN;i++)
                {
                        s=new String(str);
                        s=s.concat("_");
                        s=s.concat(String.valueOf(i));
                        outputFile= new File(s.concat(".kc_"));
                        out = new FileWriter(outputFile);
                                out.write(linkN);
                        out.write(groupN+100);
                        for (int j=1;j<=groupN;j++)
                        {
                                temType[j][0]=j;
                                temType[j][1]=-linkG[i][j];
                                temType[j][2]=0;
                                        out.write(linkG[i][j]);
                        }
                        for (x=groupN;x>=1;x--)
                                if (temType[x][1]!=0) break;
                        // pick up one link with the maximum joint
values
                        temType[x][1]++;
                        graphAAM[1][1]=x;
                        int
cur_groupN=waitConnectingList(temType,groupN,curType);

    enumerationKC(graphAAM,1,1,x,curType,cur_groupN);
                        out.close();
                }
                        System.out.println("Total graphs (isomorphic and
nonisomorphic): "+graph);
        }
        // Determining possible H atom number of each carbon in a alkane
        public static void findChemicalBonds(int linkN,int [] linkType)
        {
            // linkType[4] -- four bonds with other Carbon atoms
            //                Max number of carbon atoms connected
without H atom
            // linkType[3] -- three bonds with other Carbon atoms
            //                Max number of carbon atoms connected with
one H atom
            // linkType[2] -- two bonds with other Carbon atoms
            //                Max number of carbon atoms connected with
two H atoms
            // linkType[1] -- one bond with other Carbon atoms
            //                max number of carbon atoms connected with
three H atoms
            linkType[4]=(linkN-2)/3;
            linkType[3]=(linkN-2)/2;
            linkType[2]=linkN+1;
            linkType[1]=2*(linkN+1)/3;
        }

        // Recurisive procedure for enumerating all possible link groups
        public static String enuNumber(int L[],int groupN,int linkN,int
bond)
        {
                int n;
                String r;

                r=" L".concat(String.valueOf(groupN));
                r=r.concat(" ");
                if (groupN==1)
                {
                        if (L[groupN]<linkN) r=r.concat(String.valueOf(-
1));
                        else {
                              int re=bond-linkN*(4-groupN);
                              if (re!=0) r=r.concat(String.valueOf(-1));
                              else r=r.concat(String.valueOf(linkN));
                        }
                        return r;
                }
                if (linkN==0)
                {
                        if (bond!=0) r=r.concat(String.valueOf(-1));
                        else r=r.concat(String.valueOf(0));
                        return r;
                }
                if (L[groupN]<linkN) n=L[groupN];
                else n=linkN;
                String Num=new String();
                for (int i=0;i<=n;i++)
                {
                        Num=Num.concat(r);
                        Num=Num.concat(String.valueOf(i));
                        Num=Num.concat(enuNumber(L,groupN-1,linkN-
i,bond-i*(4-groupN)));
                }
                return Num;
        }
        // Recurisive procedure for enumerating all possible link groups
        public static String enuBondNumber(int L[],int groupN,int linkN)
        {
                int n;
                String r;

                r=" L".concat(String.valueOf(groupN));
                r=r.concat(" ");
                if (groupN==1)
                {
                        if (L[groupN]<linkN) r=r.concat(String.valueOf(-
1));
                        else r=r.concat(String.valueOf(linkN));
                        return r;
                }
                if (linkN==0)
                {
                        r=r.concat(String.valueOf(0));
                        return r;
                }
                if (L[groupN]<linkN) n=L[groupN];
                else n=linkN;
                String Num=new String();
                for (int i=0;i<=n;i++)
                {
                        Num=Num.concat(r);
                        Num=Num.concat(String.valueOf(i));
                        Num=Num.concat(enuBondNumber(L,groupN-1,linkN-
i));
                }
                return Num;
        }

       // Determining the number of each link in a possible kinematic
chain
        public static int findBondGroup(int groupN,String str,int [][]
linkG)
        {
                int row,col,preC;
                String s=new String();

                        int tmp=0;
                row=1;
                col=groupN;
                preC=col+1;
                StringTokenizer st = new
StringTokenizer(str.substring(1));
                while (st.hasMoreTokens()) {
                         s=st.nextToken();
                         if (s.charAt(0)=='L')
                         {
                                col=Integer.parseInt(s.substring(1),10);
                                if (col>=preC)
                                {
                                                            row++;
                                    if (tmp==0)
                                                            {
                                                                  for (int
i=groupN;i>col;i--)
```

```
linkG[row][i]=linkG[row-1][i];
                                                        }
                                                        else tmp=0;
                                            }
                                          preC=col;
                                  }
                                  else {
                                          linkG[row][col]=Integer.parseInt(s,10);
                                          if (linkG[row][col]==-1) {row--; tmp=1;}
                                  }
                        }
                        return row;
        }

        // Enumerating all possible kinematic chains
        public static void enumerationKC(int[][] AAM,int aAm,int
l_num,int r_joint,int [][] preType,int cur_groupN) throws IOException
        {
                        int i,j,n,num,aam,gNumber;
                        int [][] linkGroup=new int [100][20];
                        int [][] curType=new int [50][3];
                        int [][] ncurType=new int [50][3];
                        int [][] graphAAM=new int[50][50];

                        // linkGroup[i][j] -- number of the link curType[j][] in
the ith group
                        String str=new String();

                        str=chainEnumeration(preType,cur_groupN,r_joint);
                                if (!str.equals("-1"))
                                {

      gNumber=findBondGroup(cur_groupN,str,linkGroup);
                                      for (i=1;i<=gNumber;i++)
                            {
                                      for (int ii=1;ii<=cur_groupN;ii++)
                                 {
                                           curType[ii][0]=preType[ii][0];
                                             curType[ii][1]=preType[ii][1];

      curType[ii][2]=preType[ii][2];
                                      }
                                 for (int ii=1;ii<=aAm;ii++)
                                   {
                                             for (int jj=1;jj<=aAm;jj++)
                                                 graphAAM[ii][jj]=AAM[ii][jj];
                                   }
                                     num=cur_groupN;
                                                 aam=aAm;
                                                 for (j=1;j<=cur_groupN;j++)
                                   {
                                             n=linkGroup[i][j];
                                                          if (n==0)
continue;
                                           int y=curType[j][1];
                                             if (y>0) // It already joins into.
the chain
                                         { // No loop existed
                                           break;
```

```
 /*                                                 curType[j][0]-=n;
                                                      graphAAM[l_num][y]=n;
                                                    graphAAM[y][l_num]=n;

      curType[j][2]=0;
*/                                                  }
                                                else { // It doesn't yet join into
the chain
                                                                         int
z=aam+1;
                                                                         for
(int k=1;k<=n;k++)
                                                        {
                                                            aam++;
                                                              num++;

graphAAM[aam][aam]=curType[j][0];

graphAAM[l_num][aam]=1;

graphAAM[aam][l_num]=1;

      curType[num][0]=curType[j][0]-1;

curType[num][1]=aam;

      if (n>1) curType[num][2]=z;

      else curType[num][2]=0;
                                                        }
                                                    curType[j][1]+=n;
                                              }
                                        }
                                                    if (j<=cur_groupN) continue;

      num=waitConnectingList(curType,num,ncurType);
                                      if (num<2)
                                        {
                                            if (num==0)
saveGraphAAM(graphAAM,aam);
                                            }
                                        else {
                                            for (j=1;j<=num;j++)
                                                     if (ncurType[j][0]>0 &&
ncurType[j][1]>0) break;
                                                if (j<=num)
                                            {
                                                     int i_num=ncurType[j][1];
                                                       int
c_l_num=ncurType[j][1];
                                                    int c_r_num=ncurType[j][0];
                                                      ncurType[j][2]=0;

pickupLink(graphAAM,aam,i_num,ncurType,num);
                                                    int [][] nncurType=new
int[50][3];

num=waitConnectingList(ncurType,num,nncurType);

enumerationKC(graphAAM,aam,c_l_num,c_r_num,nncurType,num);
                                                 }
```

```
                        }
                    }
                }
        }

        public static void pickupLink(int[] [] graphAAM,int aam,int
k,int[] [] curType,int num)
        {
                int i,j;
                for (j=1;j<=aam;j++)
                {
                        if (graphAAM[k] [j]>0)
                        {
                                for (i=1;i<=num;i++)
                                {
                                        if (curType[i] [1]==j)
                                        {
                                                curType[i] [0]=0;
                                                break;
                                        }
                                }
                        }
                }
        }


        // Removing completely connected links from the waiting list for
connection
        // curType[] [0] - remain of joint values; curType[] [1] - link
No.
        // if (curType[] [1]<0) then it means the link doesn't yet join
into the chain, and
        // -curType[] [1] represents the number of this kind of link
        public static int waitConnectingList(int[] [] preType,int
preNum,int[] [] curType)
        {
                int curNum=0;

                for (int j=1;j<=preNum;j++)
                {
                        if (preType[j] [1]!=0 && preType[j] [0]!=0)
                        {
                                curNum++;
                                curType[curNum] [0]=preType[j] [0];
                                curType[curNum] [1]=preType[j] [1];

      curType[curNum] [2]=preType[j] [2];
                        }
                }
                return curNum;
        }

        // Recurisive procedure for enumerating all possible chains
        public static String chainEnumeration(int[] [] curType,int
indexN,int linkN)
        {
                int n,ii,jj,kk;
                int [] L=new int[20];
                        int [] mark=new int[20];
                        int [] [] linkGroup=new int[100] [20];
                        String str;
                ii=0;
                kk=0;
        for (int i=1;i<=indexN;i++)
        {
                if (curType[i][1]<0)
                        {
                                ii++;
                                L[ii]=-curType[i][1]; //
nonconnected link
                                kk-=curType[i][1];
                                mark[i]=ii;
                        }
                else if (curType[i][2]>0)
                        {
                                for (jj=1;jj<i;jj++)
                                {
                                        if
(curType[i][2]==curType[jj][2])
                                        {
                                                int k=mark[jj];
                                                L[k]++;
                                                kk++;
                                                mark[i]=k;
                                                break;
                                        }
                                }
                                if (jj==i)
                                {
                                        ii++;
                                        L[ii]=1;
                                        kk++;
                                        mark[i]=ii;
                                }
                        }
                        else {
                                ii++;
                                L[ii]=1;
                                kk++;
                                mark[i]=ii;
                        }
        }
        if (kk<linkN) return ("-1");
        str=enuBondNumber(L,ii,linkN);
                int gNumber=findBondGroup(ii,str,linkGroup);
                int [][] tmp=new int [100][20];
                for (int i=1;i<=gNumber;i++)
                {
                        for (int j=1;j<=ii;j++)
                        {
                                for (jj=1;jj<=indexN;jj++)
                                {
                                        if (mark[jj]==j)
                                        {
                                                if (curType[jj][2]>0)
                                                {
                                                        int
x=linkGroup[i][j];
                                                        if (x==0) break;
                                                        for
(kk=1;kk<=indexN;kk++)
                                                        {
```

```
                                                                    if
(curType[kk][2]==curType[jj][2])
                                                                    {

      tmp[i][kk]=1;
                                                                          x--;
                                                                    }
                                                                    if (x==0)
break;
                                                              }
                                                        }
                                                        else

tmp[i][jj]=linkGroup[i][j];
                                                        break;
                                                  }
                                            }
                                      }
                                }
                                String s=new String();
                        for (int i=1;i<=gNumber;i++)
                        {
                                      for (int j=indexN;j>=1;j--)
                                      {
                                            s=s.concat(" L");
                                            s=s.concat(String.valueOf(j));
                                            s=s.concat(" ");

      s=s.concat(String.valueOf(tmp[i][j]));
                                      }
                        }
                                return s;
            }


                  // Save AAM of a graph (kinematic chains)
            public static void saveGraphAAM(int[][] graphAAM,int aam) throws
IOException
            {
                                graph++;
                        for (int i=1;i<=aam;i++)
                        {
                                for (int j=1;j<=aam;j++)
                                {
                                        if (i==j) out.write(aam-graphAAM[i][j]);
                                        else out.write(graphAAM[i][j]);
                                }
                        }
            }
}
```