1977

A New Algorithm, And The Evaluation Of Current Algorithms, Concerning Graph Isomorphism.

Virginia Charmane Perry May
University of Alabama at Birmingham

A NEW ALGORITHM, AND THE EVALUATION OF CURRENT ALGORITHMS, CONCERNING GRAPH ISOMORPHISM

by

VIRGINIA CHARMANE PERRY MAY

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer and Information Sciences in the Graduate School, University of Alabama in Birmingham

BIRMINGHAM, ALABAMA
1977

ACKNOWLEDGEMENTS

I thank my research advisor Dr. C. C. Yang for his inspiration and guidance in my research efforts. I acknowledge Dr. A.C.L. Barnard, Chairman of the Department of Computer and Information Sciences, for providing moral support and scientific advice during my graduate studies. Also, I acknowledge help from the following people: Ms. Susan Dean of the Computer and Information Sciences Department, Mr. Wayne Satterwhite of the Biostatistics Department, and Mr. James E. Allen of the Multiple Laboratory Computer Center. Finally, I thank Ms. Joyce Perry for her perseverance in the typing of the rough draft of this dissertation.

Financial assistance was provided by a half-time instructionship in the Department of Computer and Information Sciences and by a Graduate School Fellowship.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

CHAPTER I

INTRODUCTION

1.1 The Graph Isomorphism Problem

The graph isomorphism problem is to determine whether two directed graphs are isomorphic, by finding an isomorphism if one exists. More formally, it is the problem of determining whether an isomorphism γ exists between two directed graphs $G = (V,A)$ and $G' = (V',A')$ where V and V' are the sets of vertices and $A \subseteq V \times V$ and $A' \subseteq V' \times V'$ are the incidence relations or sets of arcs. An invertible (one-to-one and onto) mapping $\gamma: V \rightarrow V'$ is an isomorphism from G to G' (or between G and G') if and only if (iff) it preserves graph incidences, i.e., for every arc $(v_i,v_j) \in A$, there is a corresponding arc $(\gamma(v_i),\gamma(v_j)) \in A'$ and vice versa.

The graph isomorphism problem can be solved theoretically by an enumeration algorithm which lists all possible invertible mappings and checks each mapping for incidence preservation. However, for a pair of isomorphic directed graphs, each having n vertices, there are up to n! possibilities to be considered, and for a pair of non-isomorphic directed graphs, each having n vertices, there are exactly n! possibilities to be considered. As n becomes large, this type of algorithm becomes totally impractical. Berztiss (1973) states that even with today's fastest computers, the resolution of all isomorphisms between two directed graphs, each having 20 vertices, would take 75,000 years. Thus, the development of more practical algorithms,

for solving the graph isomorphism problem, is important.

1.2 Previous Mathematical Research

Mathematicians have used two primary approaches for solving the graph isomorphism problem based respectively on graph enumeration theory and adjacency matrix properties. Because an undirected graph is a special case of a directed graph, the term "graph" is used to refer to a directed graph unless otherwise stated. Note also that each element in the incidence relation on the vertex set of an undirected graph is called an edge rather than an arc.

1.2.1 Graph Enumeration

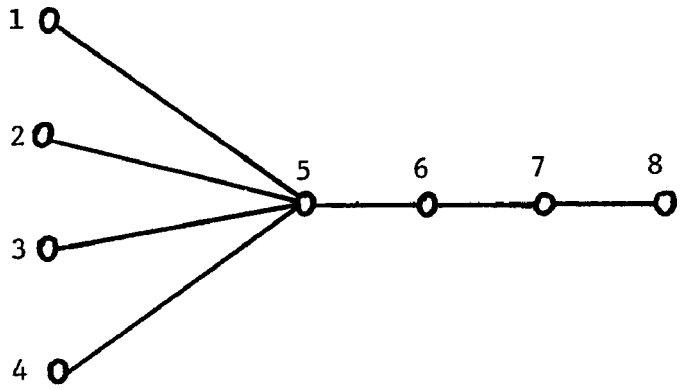
Some graph theorists tried to link the graph isomorphism problem with problems in graph enumeration. Using graph enumeration, the graph isomorphism problem became part of the unsolved enumeration problem with a given group, and was restated as the problem of counting the number of mutually nonisomorphic graphs. All of these problems from graph enumeration theory could be solved by determining the number of equivalence classes under the isomorphism relation. Much of the graph enumeration research was based on the Combinational Theorem due to Pólya (1937). Essentially, this theorem related the equivalence relation to a group of permutations of finite objects. A proof of Pólya's Theorem, in a more general form than that presented by Pólya himself, was given by de Bruijn (1964). Harary (1960, 1964) summarized his effort, and the contributions of others, in extending and applying this theorem to various types of graphs.

What is often referred to as non-Pólya enumeration turned out to be more applicable to the graph isomorphism problem, but only for the

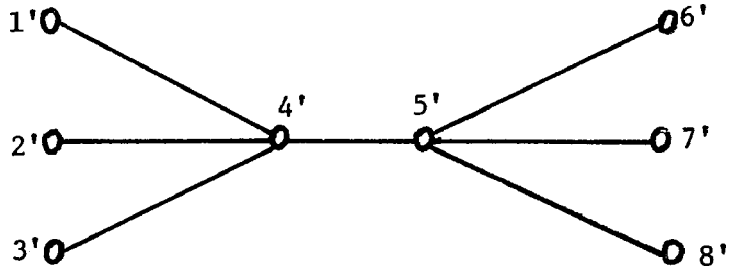
special class of planar graphs. Specifically, Tutte (1962, 1963, 1964) and Brown (1963, 1966) developed an enumeration theory for planar graphs by using the properties of these graphs to simplify the theory of Pólya. This simplified theory was successfully used by Weinberg (1966), Hopcroft and Tarjan (1972), and Hopcroft and Wong (1974) in developing isomorphism algorithms for planar graphs. Of these algorithms, the recent one developed by Hopcroft and Wong was more efficient and guaranteed a solution in linear time. However, no efficient algorithm was developed for non-planar graphs.

1.2.2 Adjacency Matrix Properties

Several theorists tried to find a necessary and sufficient condition for graph isomorphism based on the properties of the adjacency matrix of a graph. Harary (1962) conjectured that two graphs were isomorphic if their adjacency matrices had the same eigenvalue spectrum. However, Hoffman (1963) and Fisher (1966) provided numerous counter examples. Specifically, Fisher cited, as a counter example, the two 8-vertex undirected graphs which were published by Collatz and Sinogowitz (1957). Each of the Collatz and Sinogowitz graphs as shown in Figure 1.2.2.1 had the same eigenvalues, ($\pm$ 2.3027756, $\pm$ 1.3027756, 0,0,0,0), but the two graphs were not isomorphic. Turner (1967) showed that the eigenvalue spectrum for the special case of an adjacency matrix of a point symmetric graph with a prime number of vertices characterized the graphs up to an isomorphism. Using this result as a basis, Turner (1968) reported failure in finding a more powerful matrix function for characterizing graphs up to an isomorphism. Since generalized matrix functions were difficult to com-



Undirected Graph 1



Undirected Graph 2

Figure 1.2.2.1 Collatz and Sinogowitz Graphs

pute, and since they did not characterize a graph, Turner concluded that it was impractical to use such functions in devising a graph isomorphism algorithm.

1.2.3 Other Mathematical Approaches

Other researchers tried to determine mathematically whether an efficient algorithm existed for the graph isomorphism problem. A. B. Lehman communicated a conjecture to Corneil (1968) that if a graph was embedded on a surface whose genus did not exceed a fixed integer, k, then there existed an efficient solution. The genus of a surface is the largest number of simple closed curves which do not disconnect the surface. Since no non-exponential algorithm was known for calculating the genus of a graph, this conjecture was not useful as a basis for an isomorphism algorithm. However, the importance of the conjecture was its implication that no efficient solution to the general graph isomorphism problem could exist. Karp (1972) showed that a number of graph related problems for which there was no known polynomial algorithm belonged to a class called nondeterministic polynomial complete (NP-complete). If a polynomial algorithm existed for any one of these problems, then a polynomial algorithm existed for all problems in that class. However, Karp concluded that, while it was known that the graph isomorphism problem was in the class NP (computable in polynomial time by a one-tape nondeterministic Turing machine), it was still not known whether the problem belonged to the class of NP-complete.

1.3 Previous Computer Science Research

Because of the failure by mathematicians to find a necessary and

sufficient condition which could serve as a basis for an efficient graph isomorphism algorithm, computer scientists tried to develop effective, and perhaps efficient, graph isomorphism algorithms needed for applications (see Section 1.4). Most of the algorithms developed were based on conditions necessary for graph isomorphism. These graph isomorphism algorithms belonged to one of three general classes: heuristic, coding, and backtracking.

1.3.1 Heuristic Algorithms

The most popular approach for solving the graph isomorphism used heuristics as a basis. These heuristic algorithms exploited a number of conditions necessary for graph isomorphism, in an effort to reduce the number of possible invertible functions tested for isomorphism. For example, the necessary condition that two isomorphic graphs must have the same number of vertices and the same number of arcs or edges, could be used in an algorithm to decide initially if an isomorphism existed.

The graph isomorphism problem was so well suited for the use of heuristics that Unger (1964) used the problem to illustrate heuristic programming. Since his algorithm very clearly illustrated the heuristic approach, it is described in detail below.

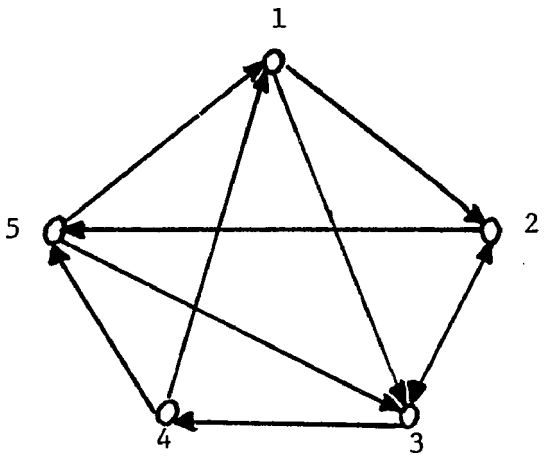
Unger first used the necessary condition for graph isomorphism that if $(v_i, v'_j) \in V \times V'$ is an element of an isomorphism, then the indegrees (id) and outdegrees (od) of the vertices were equal, i.e., $id(v_i) = id(v'_j)$ and $od(v_i) = od(v'_j)$. This necessary condition was used initially to partition the set of vertices of each graph. Both of these partitions, in some cases, reduced the possible number of

invertible functions to be checked. As an example, let graphs G and G' be graphs 3 and 4 of Figure 1.3.1.1 respectively. After applying this necessary condition to G and G' and after assigning a unique class to each subset of vertices each having the same indegree and outdegree, the initial partitioning shown in Table 1.3.1.1 was obtained.

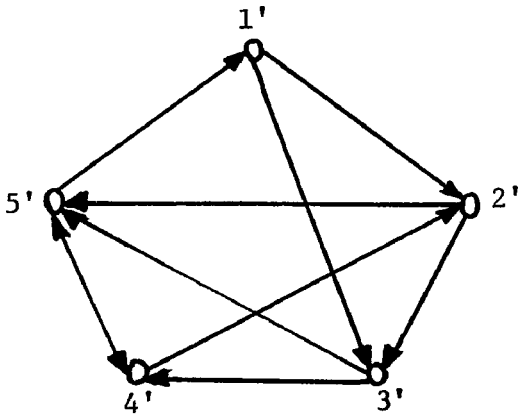
If for each unique class of G there was a corresponding unique class of G', then an isomorphism γ was defined. However, for the current example, the initial partition while reducing the number of possible vertex assignments to be checked did not completely determine an isomorphism γ.

If the mapping γ was an isomorphism from G to G', then as seen from Table 1.3.1.1, $\gamma(3) = 5'$ and $\gamma(4) = 1'$. The remaining images for vertices 1, 2, and 5 could be determined either by checking the 3! possibilities or by applying another heuristic in an effort to further reduce the possibilities.

At this point, Unger used another heuristic based on the adjacency relation. Let A_i and A'_j denote the sets of vertices adjacent to v_i and v'_j respectively. If v_i and v'_j had the same class assignment, then $\gamma(v_i) = v'_j$ belonged to an isomorphism only if the number of vertices in each class in A_i was equal to the number of vertices in the corresponding class in A'_j. Unger used a weaker form of this condition, in that he required only that the sum of the classes of vertices in A_i must equal the sum of the classes of the vertices in A'_j. The results of the application of this weaker condition to the initial partition of Table 1.3.1.1 is shown in Table 1.3.1.2. For this example, the heuristic reduced the number of possibilities to be checked from 3! to 2!. Thus, the remaining 2! possible images for vertices 1 and 5 could



Graph 3



Graph 4

Figure 1.3.1.1 Isomorphic Graphs

TABLE 1.3.1.1

INITIAL PARTITION USING UNGER'S ALGORITHM

Graph 3				Graph 4			
Vertex	id	od	Class	Vertex	id	od	Class
1	2	2	1	1'	1	2	3
2	2	2	1	2'	2	2	1
3	3	2	2	3'	2	2	1
4	1	2	3	4'	2	2	1
5	2	2	1	5'	3	2	2

TABLE 1.3.1.2

APPLICATION OF UNGER'S ADJACENCY HEURISTIC

Graph	Vertex	id	od	Old Class	Adjacency	Sum	New Class
Graph 3	1	2	2	1	(2,3,4,5)	1+2+3+1=7	1
	2	2	2	1	(1,3,5)	1+2+1=4	4
	3	3	2	2	(1,2,4,5)	1+1+3+1=6	2
	4	1	2	3	(1,3,5)	1+2+1=4	3
	5	2	2	1	(1,2,3,4)	1+1+2+3=7	1
Graph 4	1'	1	2	3	(2',3',5')	1+1+2=4	3
	2'	2	2	1	(1',3',4',5')	3+1+1+2=7	1
	3'	2	2	1	(1',2',4',5')	3+1+1+2=7	1
	4'	2	2	1	(2',3',5')	1+1+2=4	4
	5'	3	2	2	(1',2',3',4')	3+1+1+1=6	2

either be checked or further reduced.

Since the heuristic succeeded, it was repeated and the results are shown in Table 1.3.1.3. Since no further refinement was achieved, the algorithm checked the remaining 2! possibilities by enumeration and determined the isomorphism γ = (1-3',2-4',3-5',4-1',5-2').

In this example, two heuristics were used to reduce the 5! (120) possibilities to 2! (2) possibilities. However, in some cases the number of possibilities required further reduction through the use of other heuristics. Unger called these extended heuristics and listed in his paper several heuristics which could be used.

Other existing graph isomorphism heuristic algorithms, some of which were applicable only to undirected graphs, were developed by Salton and Sussenguth (1964), Sussenguth (1965), Steen (1969), Corneil (1968), Knodel (1971), Morpurgo (1971), Saucier (1971), Sirovich (1971), Levi (1974), and Yang (1975). Nearly all the algorithms obtained their initial partitions based on the vertex degree heuristic. After obtaining the initial partition, each algorithm then used various heuristics based on one or more necessary conditions for the isomorphism. A list of necessary conditions which was proposed by Corneil (1968) and Druffel (1975) is shown in Table 1.3.1.4. in which K(V) means the cardinality of the set V.

As illustrated by Unger, heuristic algorithms generated successive partitions by applying necessary conditions which reflected the relation of a vertex with its neighbors. For most pairs of graphs the heuristics worked well. However, for many highly symmetric graphs all the known heuristics failed to reduce the number of possibilities because the heuristics could not distinguish between vertices with

TABLE 1.3.1.3

ITERATION OF UNGER'S ADJACENCY HEURISTIC

Graph	Vertex	id	od	Old Class	Adjacency	Sum	New Class
Graph 3	1	2	2	1	(2,3,4,5)	4+2+3+1=10	1
	2	2	2	4	(1,3,5)	1+2+1=4	4
	3	3	2	2	(1,2,4,5)	1+4+3+1=9	2
	4	1	2	3	(1,3,5)	1+2+1=4	3
	5	2	2	1	(1,2,3,4)	1+4+2+3=10	1
Graph 4	1'	1	2	3	(2',3',5')	1+1+2=4	3
	2'	2	2	1	(1',3',4',5')	3+1+4+2=10	1
	3'	2	2	1	(1',2',4',5')	3+1+4+2=10	1
	4'	2	2	4	(2',3',5')	1+1+2=4	4
	5'	3	2	2	(1',2',3',4')	3+1+1+4=9	2

TABLE 1.3.1.4

NECESSARY CONDITIONS USED FOR HEURISTICS

Property of the Graph	The Corresponding Necessary Condition
Vertices	γ is an isomorphism only if $K(V) = K(V')$.
Arcs	γ is an isomorphism only if $K(A) = K(A')$.
Degree	γ is an isomorphism only if indegree and outdegree of a vertex $v_i \in V$ is equal to those of $\gamma(v_i) \in V'$ for all i.
Components	γ is an isomorphism only if the number of components of two graphs are equal.
Strongly Connected Component Size	γ is an isomorphism only if $v_i \in V$ and $\gamma(v_i) \in V'$ belong to strongly connected components of the same size.
Complete Subgraph	γ is an isomorphism only if the number of complete subgraphs of order k to which v_i and $\gamma(v_i)$ belong is the same for all k.
Circuit Structure	γ is an isomorphism only if the number of circuits of length k to which v_i and $\gamma(v_i)$ belong is the same for all k.
Reachability Relationships	Assume there are n paths of length k from $v_j' \in V'$ to a vertex with properties $(A_1, A_2, \ldots, A_i)$. Vertex $v_i \in V$ and v_j' may be mapped by γ only if there exists a vertex $v_k \in V$ with properties $(A_1, \ldots, A_i)$ such that there are exactly n paths of length k from v_i to v_k for all k.
Automorphism	γ is an isomorphism only if v_i and $\gamma(v_i)$ belong to similar cells of the automorphism partition.
F Relationships	γ is an isomorphism only if the graphs induced by the removal of v_i and $\gamma(v_i)$ have the same number of arcs.

similar characteristics. If a heuristic partitioned the vertices into h classes, and k(i) was the number of vertices in the i^{th} class, then the remaining possibilities were given by $k(1)!\cdot k(2)!\cdot\ldots\cdot k(h)!$. If there was some class of graphs of n vertices for which the heuristic was unable to generate refined partitions and h = 1, then k(h) = n, and the number of possibilities was still n!. A strongly regular graph was an example of a class of graphs for which existing heuristics were not very effective. In this case some other type of procedure which systematically assigned remaining vertices was required to make the algorithm more practical.

1.3.2 Coding Algorithms

Although based on heuristics, coding algorithms represented a different approach, in that the backtracking technique was used to construct a canonical code for the two graphs. Shah, Davida and McCarthy (1974) developed a coding algorithm for undirected graphs. This algorithm used the adjacency matrix of a graph to derive a canonical code for the graph. A code for a graph was defined as the binary number formed by concatenation of successive rows of the upper triangle of the adjacency matrix. A code was called "canonical" if the rows and the corresponding columns of the adjacency matrix were permuted such that the resulting binary number was maximal. Of course in order to determine if a code was maximal, all permutations which possibly could result in a larger binary number were checked. This checking was performed by a backtracking algorithm.

For an example, the two nonisomorphic undirected graphs 1 and 2 of Figure 1.2.2.1 were used. From the adjacency matrix of graph 1 as

shown in Table 1.3.2.1, the canonical code was generated by first interchanging labels of vertex 5 and vertex 1. This permutation caused row one to contain ones in columns two through six. Since vertex 6 had a degree greater than that of any other vertices adjacent to vertex 5, it was renumbered as vertex 2. This reordering given by (5-1,6-2,3-3,4-4,1-5,2-6,7-7,8-8) produced the canonical code F810001 (hexadecimal number) shown in the upper triangle of Table 1.3.2.2.

If the canonical codes for two graphs were equal then the mapping specified by the reordering was an isomorphism. Using the reordering given by (4'-1',5'-2',3'-3',1'-4',2'-5',6'-6',7'-7',8'-8') the canonical code F038000 for graph 2 was produced. Thus, it was concluded that graphs 1 and 2 were not isomorphic, a fact obvious from visual inspection of Figure 1.2.2.1.

Proskurowski (1974), using the incidence matrix of a graph, developed a similar coding scheme for undirected simple graphs called the maximal incidence matrix. Both of these coding algorithms were based on a necessary and sufficient condition, i.e., two graphs were isomorphic iff their canonical codes were equal. However, neither algorithm was efficient for graphs with large numbers of vertices (probably greater than 10), since the amount of backtracking necessary to check possible maximal codes increased rapidly as the numbers of vertices of the graphs increased.

1.3.3 Backtracking Algorithms

A third approach to solving the graph isomorphism problem was to use some necessary conditions and a backtracking technique to select possible vertex assignments to test for isomorphism. The backtracking

TABLE 1.3.2.1

ADJACENCY MATRIX FOR GRAPH 1

	1	2	3	4	5	6	7	8
1	0	0	0	0	1	0	0	0
2	0	0	0	0	1	0	0	0
3	0	0	0	0	1	0	0	0
4	0	0	0	0	1	0	0	0
5	1	1	1	1	0	1	0	0
6	0	0	0	0	1	0	1	0
7	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	1	0

TABLE 1.3.2.2

CANONICAL REORDERED MATRIX FOR GRAPH 1

	1	2	3	4	5	6	7	8
1	0	1	1	1	1	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0
7	0	1	0	0	0	0	0	1
8	0	0	0	0	0	0	1	0

The canonical code is given by the binary number

11111000000100000000000001

or the hexadecimal number

F810001

approach represented an improvement over the heuristic approach because, if the heuristic part of the backtracking algorithm failed to reduce the number of possible vertex assignments, then the backtracking technique insured a stepwise elimination of all inconsistent vertex assignments. Backtracking algorithms for finding graphs isomorphisms were recently developed by Berztiss (1973), Ullmann (1976), and Schmidt and Druffel (1976) (see Chapter III for detail reviews).

1.3.4 Performance of the Algorithms

Except in the case of Corneil, performance of each algorithm if given, was based on experimental results using various classes of graphs. Corneil, using a conjecture, stated that the upper bound of his algorithm was of the order $O(n^{5+k})$, $2 \leq k \leq n$. However, it was shown by a counter example of R. Mathon that the conjecture was not true (Corneil, 1974).

All heuristic algorithms which completely solved the graph isomorphism problem had some type of permutation procedure to check the remaining vertex assignments. Thus, reported estimates on the efficiency of these algorithms were usually based on random graphs. Furthermore, not all the authors reported the complexities of their algorithms. Corneil, using only undirected graphs, claimed the orders: $O(n^5)$ if the graph did not contain a k-strongly regular subgraph, $O(n^4)$ for polygons, and $O(n^2)$ for random graphs. Sirovich stated that his algorithm, which also was based on a conjecture, was of the order $O(n^5)$ for most graphs. Levi claimed that his algorithm performed on the order $O(n^6)$ for most undirected graphs tested. However, neither Sirovich nor Levi described the classes of graphs used.

The orders of efficiency for the coding algorithms were not reported. However, Proskurowski stated that his algorithm was efficient only for undirected graphs with a small number of vertices and edges.

For random nonisomorphic simple n/2 regular graphs, Berztiss claimed a statistical order $0((2.15 \times 10^{-5}) \exp(1.07n))$. Ullmann gave no order, but stated that his algorithm performed as well as, or better than, that of Berztiss. Schmidt and Druffel claimed an order $0(n^2)$ for non-regular random graphs, simple polygons and other special graphs. Using strongly regular graphs of order 25 (Paulus, 1973) Druffel gave an order $0(n^3)$. He also conjectured that is was reasonable to expect performance better than $0(n^4)$ for most graphs since the predicted dynamic bound never exceeded $0(n^5)$ for all graphs tested.

Other authors provided running times and storage estimates. However, since all algorithms were executed on different computers and using different languages, any direct comparison would be meaningless.

1.4 Applications of Graph Isomorphism Algorithms

Graph isomorphism algorithms were applied in such fields as information retrieval, chemistry, circuit and network theory, and pattern recognition. Most of the initial algorithms were by-products of a particular application. As the graph isomorphism problem became better known, the emphasis switched to the development of more efficient algorithms for existing applications.

1.4.1 Information Retrieval

In automatic information retrieval, comparison between a description of the stored information and the requested information was one of

the principal tasks performed. Salton and Sussenguth (1964) suggested that graph matching techniques could compare graphs representing requests for information. They developed a topological structure-matching procedure. This procedure matched the information graph and query graph in parallel by identifying certain simple properties of the vertices of the two graphs, and by equating those subsets of vertices in the two graphs that exhibited similar properties. A standard process was then used to break down matching subsets of vertices into smaller and smaller sets, until a complete correspondence was determined for all vertices of the two graphs. Salton (1968) discussed topological and other types of structure-matching procedures which could be used in an automatic document retrieval system to identify matching phrases included in documents and search requests.

1.4.2 Chemistry

There was probably no science in greater need of an automatic information retrieval than chemistry. Many compounds were known, and many new ones were produced daily. The chemist had two main problems: first, he wanted to find out whether the substance in his test tube was already known; second, given a substance, he wanted to know the properties of similar substances. Both problems reduced to a matching process between the given substance and the millions of substances already known and cataloged.

Sussenguth (1964, 1965) described a method of cataloging chemical compounds as undirected graphs. In his model, the atoms of a compound corresponded to the vertices and the interatomic bonds corresponded to

the edges. Given a compound, a search was made to determine if its graph was isomorphic to any graph or any subgraph in the library. Further applications in this area were described by Tate (1967), and Lynch et al. (1971).

1.4.3 Network and Circuit Theory

In developing an algorithm for an efficient layout of microelectronic circuits, Weinberg (1966) devised an algorithm to find the isomorphisms between two triply connected graphs. Cornog and Bryan (1966) described a search method for transistor patents which used the Seshu and Reed model (1961) of an electrical network.

1.4.4 Pattern Recognition

In the automatic recognition of printed characters, techniques had to be devised specifically for the recognition of characters printed by hand. Handprinted characters were difficult to recognize, because of the major differences that existed between a character printed by one person and the same character printed by a different person. Sherman (1960) suggested a technique for recognizing characters based on a graph representation of the character. Sherman regarded the limbs of the character as edges and the junctions and ends of the limbs as vertices of a graph. He then explored the idea that if two graphs of two characters were isomorphic, then the characters belonged to the same recognition class. Barrow and Popplestone (1971) extended the idea of using a graph to represent a character, by adding further rules to assign attributes to each vertex or edge. They suggested that if the graphs of the two characters were isomorphic, then the two characters belonged to the same recognition class. Grimsdale et al.

(1959) provided another example of this type of technique, using different rules for constructing a graph from a pattern. Ullmann (1973, 1976) further explored the idea of using a graph isomorphism algorithm as the basis for a character-recognition technique. He noted that determining whether two graphs were related by an isomorphism was very similar to determining whether two patterns were related by a distortion which conserved spatial relationships within known limits.

Another pattern recognition problem was the detection of a relationally described object embedded in a pattern. Barrow et al. (1972) and Sakai et al. (1972) suggested subgraph isomorphism be used to solve this problem. Sakai et al. developed a system to detect areas of overlap in aerial photographs taken sequentially, in order to combine the many small pictures into one large picture of the area.

1.5 Summary of Original Research Reported in this Dissertation

A new graph isomorphism algorithm is described. The algorithm is based on a necessary and sufficient condition for the existence of an isomorphism between two graphs. The condition is based on the isomorphisms between the Moore sequential machines (MSM) corresponding to the two graphs. Thus, in addition to the isomorphism algorithm, an algorithm is given which transforms any graph to a MSM. Using a method which partitions the union of the state sets of two MSM's into closed output-consistent partitions, the isomorphism algorithm finds any isomorphism between two graphs by examining any isomorphism between their MSM's. An analysis of theoretical and experimental bounds of both algorithms is made. The result is that, while the algorithms are not guaranteed to run in polynomial time, they do perform efficiently for

a large class of regular graphs.

In order to gain some insight into actual computing performance of the current graph isomorphism algorithms, an experimental evaluation and comparison of the new isomorphism algorithm and three other recent algorithms is presented. First, the three recent algorithms by Berztiss, Ullmann, and Schmidt and Druffel are reviewed. The computer implementation of the four algorithms and experiments which consist of several classes of graphs are described. Using the experimental results, the four methods are compared using performance criteria based on the execution time performance of their PL/I implementations and the performance of their algorithms measured by the number of vertex assignments required to process a pair of graphs. The evaluation and comparison leads to the conclusion that in the general case Schmidt and Druffel's algorithm is superior to the other three algorithms. However, when regular graphs are used, the proposed new graph isomorphism algorithm is superior to the isomorphism algorithm by Schmidt and Druffel.

Chapter II presents the new graph isomorphism algorithm. Chapter III reviews the other current graph isomorphism algorithms, i.e., the three backtracking algorithms by Berztiss, Ullmann, and Schmidt and Druffel. Chapter IV deals with the evaluation procedure used in comparing the graph isomorphism algorithms of Chapters II and III. Chapter V presents the experimental results and conclusions obtained from the evaluation of these four algorithms. Appendix A lists all relevant definitions and notations which are used but not defined in this dissertation. Appendices B-G contain the listings of the programs and algorithms which are used in the experimental evaluation of the four

graph isomorphism algorithms.

CHAPTER II

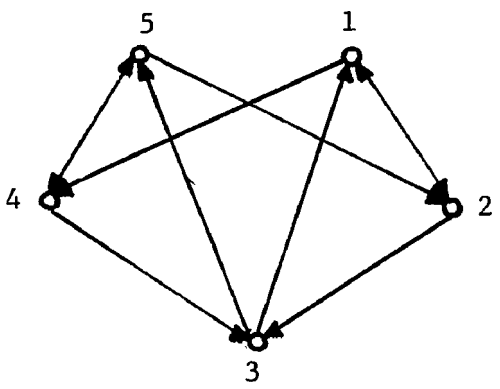
A NEW GRAPH ISOMORPHISM ALGORITHM BASED ON FINITE AUTOMATA

2.1 Introduction

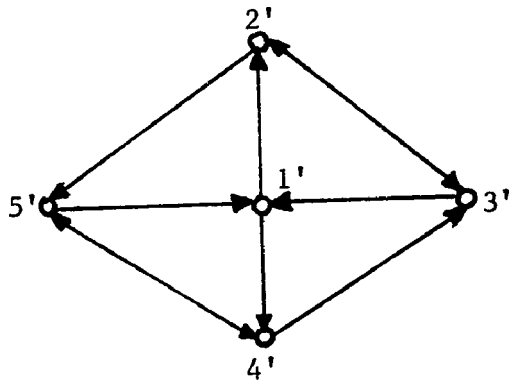
In a recently published article, Yang (1975) introduced a new method for determining the isomorphisms between two graphs based on the isomorphisms between their corresponding Moore sequential machines (MSM). Isomorphisms between two MSM's were determined by partitioning all states of both MSM's to satisfy the conditions of Corollary 1 which is restated below as Theorem 2.1.1 in which I and I' are the input alphabets of the MSM's, S and S' are the state sets of the MSM's, and ϕ is the empty set.

Theorem 2.1.1. Let $K(I) = K(I') = 1$ and $S \cap S' = \phi$. There is an isomorphism between M and M' iff there is a nontrivial closed and output-consistent partition over $S \cup S'$ such that each element of the partition contains exactly two states: one belonging to S and the other belonging to S'.

However, a partition which induced an isomorphism between two MSM's could fail to induce an isomorphism between the represented graphs as evidenced by P_{21} and γ_{21} or P_{22} and γ_{22} in Illustration 6 of Yang's paper. This counter example is shown in Figure 2.1.1. Thus, as is proved in Section 2.5, an isomorphism between two MSM's provided only a necessary condition for graph isomorphism. This proof corrected Yang's initial results and led to a new method based on



Graph 5



Graph 6

Two nontrivial closed and output-consistent partitions of MSM's representing Graphs 5 and 6

P_{21} = {1-4',2-3',3-1',4-5',5-2',6-7',7-8',8-6',9-9',10-10',11-11', 12-12'}

P_{22} = {1-2',2-5',3-1',4-3',5-4',6-7',7-9',8-6',9-8',10-10',11-11', 12-12'}

The corresponding induced isomorphisms between Graphs 5 and 6

γ_{21} = {1-4',2-3',3-1',4-5',5-2'}

γ_{22} = {1-2',2-5',3-1',4-3',5-4'}

Figure 2.1.1 A Counter Example for Yang's Conditions

finite automata for solving the graph isomorphism problem. The new method is developed in this Chapter.

2.2 Graph Representations from Finite Automata

A finite automaton without outputs known as a nondeterministic state machine (NDSM) is a triplet $N = (V,I,F)$, where the state set V and the input alphabet I are finite nonempty sets, and the next state function F is mapping from $V \times I$ into 2^V, the power set of V. A deterministic state machine (DSM) is a triplet $D = (S,I,H)$, where the state sets $S \subseteq 2^V$ and the input alphabet I are finite nonempty sets, and the next state function H is mapping from $S \times I$ into S. It is noted that $H(\phi,i) = \phi$, where $i \notin I$.

A finite automaton known as a Moore sequential machine (MSM) is a quintuple $M = (S,I,O,H,J)$, where S, I and H were previously defined, the output alphabet O is a finite nonempty set, and the output function J is mapping from S onto O.

Any graph G can be represented by a NDSM $N = (V,I,F)$ with V, its set of states, being the same as the set V of vertices of $G = (V,A)$;

$$I = \{i\} \tag{2.2.1}$$

being its input alphabet containing the single input i and $F: V \times I \rightarrow 2^V$ such that

$$F(a,i) = \{b \mid (a,b) \notin A\} \tag{2.2.2}$$

being its next state function. It is noted that if for some $a \notin V$ there is no arc $(a,b) \notin A$ for all $b \notin V$, then $F(a,i) = \phi$. Using (2.2.1) and (2.2.2), the NDSM N of graph 5 of Figure 2.1.1 is constructed and is shown in Table 2.2.1.

TABLE 2.2.1

NDSM N AND DSM D CORRESPONDING TO GRAPH 5

NDSM N

State a	Next State F(a,i)
1	{2,4}
2	{1,3}
3	{1,5}
4	{3,5}
5	{2,4}

DSM D

State s	Next State H(s,i)
{1}	{2,4}
{2}	{1,3}
{3}	{1,5}
{4}	{3,5}
{5}	{2,4}
{2,4}	{1,3,5}
{1,3}	{1,2,4,5}
{1,5}	{2,4}
{3,5}	{1,2,4,5}
{1,3,5}	{1,2,4,5}
{1,2,4,5}	{1,2,3,4,5}
{1,2,3,4,5}	{1,2,3,4,5}

In order to describe the DSM corresponding to N, F is extended as $F: V \times I^* \to 2^V$ such that

$$F(a,\lambda) = \{a\} \text{ for all } a \in V \tag{2.2.3}$$

and

$$F(a,wi) = \bigcup_{v \in F(a,w)} F(v,i) \text{ for every } (a,w) \in V \times I^* \tag{2.2.4}$$

where I^* is the free monoid generated by i and has the identity λ. The DSM corresponding to N (or G) is D = (S,I,H) with

$$S = \{s \mid s = F(a,w) \text{ for each } (a,w) \in V \times I^*\} \tag{2.2.5}$$

being its set of states; and $H: S \times I \to S$ such that

$$H(s,i) = \bigcup_{a \in s} F(a,i) \text{ for each } s \in S \tag{2.2.6}$$

being its next state function. Using (2.2.5) and (2.2.6) the DSM D of Table 2.2.1 corresponding to N of Table 2.2.1 (or graph 5) is constructed. The first five rows of D represent, as sets, the states of N (or vertices of graph 5).

A MSM corresponding to D (or N, or G) is M = (S,I,O,H,J) with O being its output alphabet, whose elements are bags called outputs, and $J: S \to O$ being its output function. It is noted that a bag is an ordered set whose elements are not necessarily distinct. A bag is enclosed by a pair of square brackets. Before defining O and J, it is noted that a graph G has, within an isomorphism, a unique DSM. However, a graph G has more than one MSM because both O and J can be defined differently for a given DSM.

In general, the output function should define outputs which can reflect information about the graph structure. There are a number of output functions which reflect different properties of the graph

structure. The outdegree of each vertex of the graph G is reflected by

$$J(s) = [K(H(s,i))] \text{ for each } s \in S. \tag{2.2.7}$$

The indegree of each vertex of the graph G is reflected when the output of each state $s \in S$ of the DSM is defined by

$$J(s) = [K([H(t,i) \mid s \subseteq H(t,i)])] \text{ for each } t \in S \tag{2.2.8}$$

This definition is defined to consider only distinct next states by

$$J(s) = [K(\{H(t,w) \mid s \subseteq H(t,w),(t,w) \in S \times (I^* - \{\lambda\})\})] \tag{2.2.9}$$

The output of each state also may reflect the output of each element of that state. For instance, the indegree of each state could be defined as the union of the indegrees of each of its elements by

$$J(s) = \bigcup_{a \in s} [K([H(t,w) \mid \{a\} \subseteq H(t,w),\ (t,w) \in S \times (I^* - \{\lambda\})])] \tag{2.2.10}$$

Finally J is defined to reflect all circuits of length two which originate from each vertex of G by

$$J(s) = \bigcup_{a \in s} [K\{t \mid t \in H(\{a\},i) \wedge a \in H(\{t\},i)\}]. \tag{2.2.11}$$

The MSM of Table 2.2.2 corresponding to D (or N) of Table 2.2.1 is constructed by using equations (2.2.7) through (2.2.11) to define the output for each state of D. In practice, only the outputs which reflect the most graph structure are used. In this example, output definition (2.2.9) can be chosen.

2.3 An Algorithm for Transforming a Graph into a Moore Sequential Machine

The process defined by (2.2.5) through (2.2.11), of transforming a graph into a unique DSM or MSM, is described by Algorithm 1. Algorithm 1, which is presented below, first transforms a graph G into a unique

TABLE 2.2.2

A MSM M CORRESPONDING TO D OF TABLE 2.2.1

State s	Next State H(s,i)	Output J(s) Defined By				
		(2.2.7)	(2.2.8)	(2.2.9)	(2.2.10)	(2.2.11)
{1}	{2,4}	[2]	[8]	[5]	[8]	[1]
{2}	{1,3}	[2]	[8]	[3]	[8]	[1]
{3}	{1,5}	[2]	[5]	[4]	[5]	[0]
{4}	{3,5}	[2]	[8]	[3]	[8]	[1]
{5}	{2,4}	[2]	[8]	[5]	[8]	[1]
{2,4}	{1,3,5}	[3]	[8]	[3]	[8,8]	[1,1]
{1,3}	{1,2,4,5}	[4]	[4]	[3]	[5,8]	[0,1]
{1,5}	{2,4}	[2]	[7]	[4]	[8,8]	[1,1]
{3,5}	{1,2,4,5}	[4]	[4]	[3]	[5,8]	[0,1]
{1,3,5}	{1,2,4,5}	[4]	[3]	[2]	[5,8,8]	[0,1,1]
{1,2,4,5}	{1,2,3,4,5}	[5]	[5]	[2]	[8,8,8,8]	[1,1,1,1]
{1,2,3,4,5}	{1,2,3,4,5}	[5]	[2]	[1]	[5,8,8,8,8]	[0,1 1 1 1]

DSM by (2.2.5) and (2.2.6). Next, the MSM is constructed by defining O and J for each state of the DSM. For each s $\in$ S, J(s) is defined to be the bag of the outputs which are defined by (2.2.9) and (2.2.11). It is noted that any combination of the equations defining J(s) can be used. First, Algorithm 1 is presented and illustrated by an example. Then, the computational complexity of the algorithm is discussed.

2.3.1 Algorithm 1

Algorithm 1 transforms a graph G into a MSM M. The graph G is represented by the adjacency matrix GA, and the MSM M is represented by the arrays H and J. The variables ns and h represent respectively the current number of states and the current number of next states implied by or contained in the constructed array H. K(V) is the cardinality of the set of vertices V of graph G. The variable si is the index for the states of the MSM. The variable KG contains the number of states which are elements of the next state of the state indexed by si. The arrays J9 and J11 contain the outputs of each state as defined by (2.2.9) and (2.2.11) respectively.

Step 1. (Initialize variables.)

ns $\leftarrow$ 0, h $\leftarrow$ K(V).

Step 2. (If all next states are states of the constructed DSM, then proceed to construct the MSM by defining outputs.)

If ns = h then go to Step 12.

Step 3. (Otherwise, determine next states of all states in the constructed DSM by using array H to contain all next states.)

si $\leftarrow$ ns + 1, ns $\leftarrow$ h.

Step 4. If si > ns then go to Step 2.

Step 5. If $KG \leftarrow K(\{j \mid GA_{si,j} = 1, 1 \leq j \leq K(V)\}) < 2$

then go to Step 10.

Step 6. (Check if next state is a state of DSM.)

If there exists a j such that $GA_{si*} = GA_{j,*}$ for $1 \leq j \leq si-1$

then $H_{si} \leftarrow H_j$, go to Step 11.

($GA_{si,*}$ represents row si of matrix GA).

Step 7. $h \leftarrow h + 1$.

Step 8. For $1 \leq k \leq K(V)$, if $GA_{si,k} = 1$ then $GA_{h,*} = GA_{h,*} \vee GA_{k,*}$.

($\vee$ is the OR operation).

Step 9. $H_{si} \leftarrow h$, go to Step 11.

(At this point, matrix GA has been expanded to include all next states for each state in S indexed by si.)

Step 10. If $KG = 1$

then $H_{si} \leftarrow j$ such that $GA_{si,j} = 1$

else $H_{si} \leftarrow -1$ (where -1 represents the empty set ϕ).

Step 11. $si \leftarrow si + 1$, go to Step 4.

Step 12. (At this point, the DSM has been defined by H. By defining the outputs of DSM, the MSM is constructed. The array J contains the bag of outputs J9 as defined by (2.2.9) and J11 as defined by (2.2.11) for each state in the DSM indexed by si.)

For $1 \leq si \leq K(V)$

$J9_{si} \leftarrow [K(\{H_j \mid GA_{j,si} = 1, 1 \leq j \leq ns\})]$.

For $K(V)+1 \leq si \leq ns$

$J9_{si} \leftarrow [K(\{H_j \mid GA_{k,*} = GA_{j,*} \wedge GA_{k,*}$

where $H_k = si$ for the least k and $1 \leq j \leq ns\})]$.

($\wedge$ is the AND operation).

Step 13. For $1 \leq si \leq K(V)$

$$J11_{si} \leftarrow [K(\{j \mid GA_{si,j} = 1 \wedge GA_{j,si} = 1,\ 1 \leq j \leq K(V)\})].$$

For $K(V)+1 \leq si \leq ns$

$$J11_{si} \leftarrow [J_j \mid GA_{k,j} = 1$$

where $H_k = si$ for the least k and $1 \leq j \leq K(V)]$.

Step 14. For $1 \leq si \leq K(V)$

$$J_{si} \leftarrow [J9_{si}, J11_{si}].$$

Step 15. Stop.

For demonstrating Algorithm 1, graph 7 in Figure 2.3.1.1 is used. The first six iterations of Steps 4-11 produce the NDSM equivalent to graph 7. The next state of each state {v}, such that $v \notin V$, is the union of all states {t}, such that $(v,t) \notin A$. For instance, the next state of state {1} is {2,5,6}, since (1,2), (1,5), and (1,6) are all arcs of graph 7. Since the next states which are generated in the first six iterations are not states of DSM, i.e., the closure condition of Step 2 is not satisfied, five more iterations of Steps 4-11 are performed. For each of these states, the next state is obtained by taking the union in Step 8 of the set of next states of its elements. For example, for state {2,5,6}, the next state {1,2,3,4,5} is obtained by {1,3,4} U {3,4,5}. These five iterations produce four next states which are not states of DSM, and thus, four more iterations of Steps 4-11 are performed. After these iterations, the closure condition of Step 2 is satisfied, and the DSM equivalent to graph 7 is obtained. Columns two and three of Table 2.3.1.1 represent the DSM.

The rightmost column of Table 2.3.1.1 represents, for each state, the outputs J(s) which are defined by Steps 12-14. For example, in Step 12, the first output [6] of state {1} is obtained by counting the

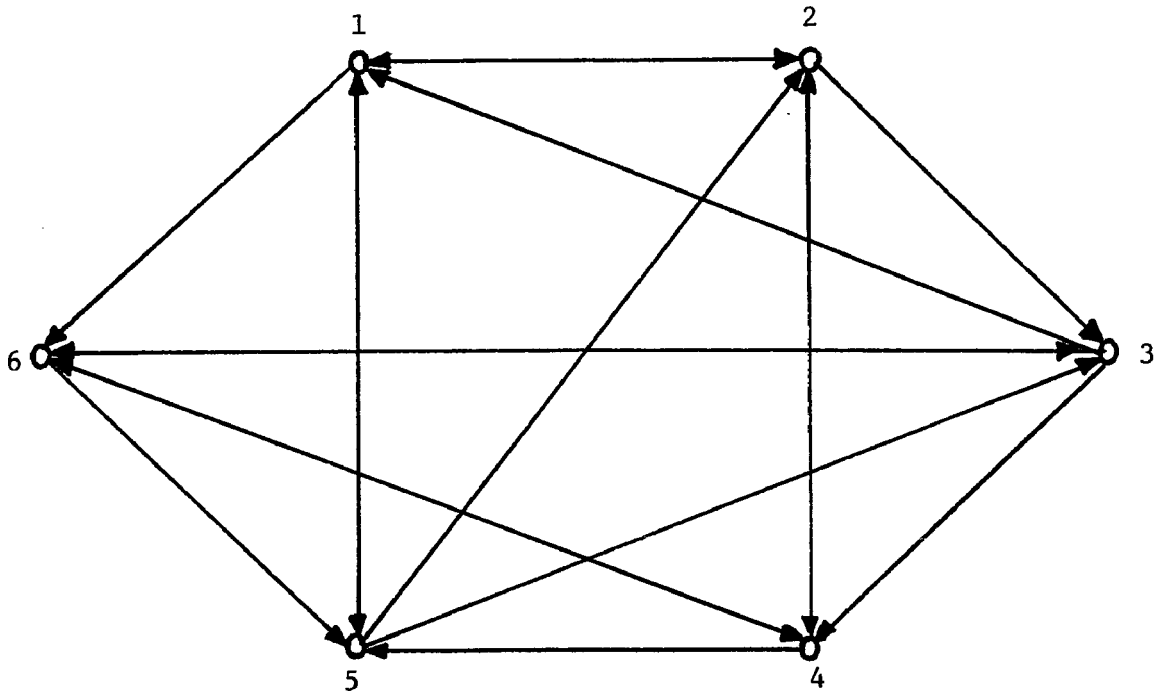


Figure 2.3.1.1 Graph 7

TABLE 2.3.1.1

MSM M CORRESPONDING TO GRAPH 7

Index si	State s	Next State H(s,i)	Output J(s)
1	{1}	{2,5,6}	[[6],[2]]
2	{2}	{1,3,4}	[[6],[2]]
3	{3}	{1,4,6}	[[6],[1]]
4	{4}	{2,5,6}	[[7],[2]]
5	{5}	{1,2,3}	[[6],[1]]
6	{6}	{3,4,5}	[[5],[2]]
7	{2,5,6}	{1,2,3,4,5}	[[4],[1,2,2]]
8	{1,3,4}	{1,2,4,5,6}	[[3],[1,2,2]]
9	{1,4,6}	{2,3,4,5,6}	[[3],[2,2,2]]
10	{1,2,3}	{1,2,3,4,5,6}	[[3],[1,2,2]]
11	{3,4,5}	{1,2,3,4,5,6}	[[4],[1,1,2]]
12	{1,2,3,4,5}	{1,2,3,4,5,6}	[[2],[1,1,2,2,2]]
13	{1,2,4,5,6}	{1,2,3,4,5,6}	[[2],[1,2,2,2,2]]
14	{2,3,4,5,6}	{1,2,3,4,5,6}	[[2],[1,1,2,2,2]]
15	{1,2,3,4,5,6}	{1,2,3,4,5,6}	[[1],[1,1,2,2,2,2]]

number of distinct next states which contain the element 1. In Step 13, the output [2] of state {1} is obtained by counting the 1's in the next state of each element contained in the next state of {1}. Since the next state of {1} is {2,5,6}, the 1's of the next states of {2}, {5} and {6} are counted, and a value of 2 is obtained. By defining in Steps 12-14 the outputs for each state of the DSM, the MSM is obtained. For ease of implementation, an index is used for each state of the MSM and is shown in column 1.

2.3.2 The Computational Complexity of Algorithm 1

The theoretical computational complexity of Algorithm 1 is difficult to analyze, since for the DSM or a MSM, the state set $S \subseteq 2^V$. However, based on the definitions of the state set S and the next state function H given respectively by (2.2.5) and (2.2.6), it is conjectured that Algorithm 1 requires less than 2^n states to represent a graph of n vertices as a MSM.

Conjecture 2.3.2.1. Algorithm 1 requires at most $1 + \sum_{i=i}^{n} (\min(n, {}_nC_i))$ states to represent a graph of n vertices as a MSM.

It is obvious that the minimum number of states, which is required to represent a graph as a MSM, is n. The minimum number is achieved when for all $a \in V$, $H(\{a\},i) = \{b\}$, where $b \in V$ or $H(\{a\},i) = \phi$.

Using Conjecture 2.3.2.1, the upper bound of Algorithm 1 is derived.

Theorem 2.3.2.1. Algorithm 1 transforms a graph of n vertices into a DSM or MSM in less than time $O(n^4)$.

Proof. The upper bound of Algorithm 1 is determined by its innermost loop which is given by Step 8. In Step 8, for each distinct next

state, up to n^2 "OR" operations can be required. By Conjecture 2.3.2.1, there are at most $1 + \sum_{i=1}^{n} (\min(n, {}_nC_i))$ next states. Since $1 + \sum_{i=1}^{n} (\min(n, {}_nC_i)) < n^2$. Thus, the upper bound of Step 8 is

$$1 + \sum_{i=1}^{n} (\min(n, {}_nC_i))) \cdot n^2 < n^2 \cdot n^2 = n^4.$$

This implies a bound less than $O(n^4)$. Q.E.D.

The lower bound of Algorithm 1 occurs when the number of states of the MSM is the minimum n. In this case, Step 5 determines the algorithm's bound. Since Step 5 takes n comparisons to determine the number of elements in the next states of the n states, it is bound by n^2. Thus, the lower bound is $O(n^2)$.

2.4 A Necessary and Sufficient Condition for Graph Isomorphism

As previously stated, the graph isomorphism problem is to determine any isomorphism which exists between two graphs G = (V,A) and G' = (V',A'). Using the equations of Section 2.2, a graph G can be represented as the NDSM N, the DSM D, and a MSM M. Analogously, the graph G' can be represented by the NDSM N', the DSM D', and a MSM M' with primed symbols used in the definitions except for (2.2.1) and the outputs in O'.

Since any graph can be alternatively represented by a NDSM which is unique within an isomorphism, it is concluded that an isomorphism exists between two graphs iff an isomorphism exists between their NDSM's. This result is stated as the following theorem.

Theorem 2.4.1. An isomorphism γ between two given graphs G and G' exists iff there is an isomorphism between their corresponding NDSM's N and N'.

Proof. Since the NDSM representing a graph is unique within an isomorphism, it is not necessary to use two different symbols for differentiating an isomorphism between two graphs and that between their corresponding NDSM's.

Let γ: $V \to V'$ be an invertible function. The theorem is proved by showing that γ preserves state transitions in N and N', i.e.,

$$\{\gamma(b) \mid b \in F(a,i)\} = F'(\gamma(a),i), \tag{2.4.1}$$

iff γ preserves graph incidences in G and G', i.e.,

$$(a,b) \in A \text{ iff } (\gamma(a),\gamma(b)) \in A'. \tag{2.4.2}$$

Suppose that γ preserves state transitions as defined by (2.4.1). Note that the left-hand side of (2.4.1) cannot be denoted by $\gamma(F(a,i))$ since $F(a,i)$ is a subset of V rather than an element of V. However, for a later convenience, γ can be extended so that

$$\gamma(F(a,i)) = \{\gamma(b) \mid b \in F(a,i)\} \tag{2.4.3}$$

Then by means of (2.2.2), γ being invertible, (2.4.1), (2.4.3) and (2.2.2) again, for any arc $(a,b) \in A$

$$\begin{aligned}(a,b) \in A &\text{ iff } b \in F(a,i)\\ &\text{iff } \gamma(b) \in \gamma(F(a,i))\\ &\text{iff } \gamma(b) \in F'(\gamma(a),i)\\ &\text{iff } (\gamma(a),\gamma(b)) \in A'\end{aligned}$$

Thus, γ satisfies (2.4.2).

On the other hand suppose that γ satisfies (2.4.2). Then by means of (2.2.2), (2.4.2), and (2.2.2) again,

$$\begin{aligned}\{\gamma(b) \mid b \in F(a,i)\} &= \{\gamma(b) \mid (a,b) \in A\}\\ &= \{\gamma(b) \mid (\gamma(a),\gamma(b)) \in A'\}\\ &= F'(\gamma(a),i)\end{aligned}$$

Thus, γ satisfies (2.4.1). Q.E.D.

Since a NDSM is generally not closed, the corresponding DSM, which is closed and unique within an isomorphism, is constructed. However, as will be stated in the following theorem, it cannot be concluded that an isomorphism between two DSM's determines an isomorphism between the corresponding NDSM's.

Theorem 2.4.2. If there is an isomorphism γ between the NDSM's N and N' corresponding to the given graphs G and G', then there exists an isomorphism β between the DSM's D and D'. The converse may not be true.

Before proving Theorem 2.4.2, it is noted that if X is a finite set, then X_r denotes a subset of X. For proving Theorem 2.4.2., the NDSM $D_r = (S_r, I, H_r)$ is defined from D with

$$S_r = \{\{v\} \mid v \in V\} \tag{2.4.4}$$

and H_r: $S_r \times I \to S$ such that

$$H_r(\{v\},i) = F(v,i) \text{ for all } \{v\} \in S_r. \tag{2.4.5}$$

By (2.2.5), (2.2.3) and 2.4.4), S_r is a subset of S and by (2.2.1), (2.2.2), (2.4.4) and (2.4.5), D_r and N are isomorphic. Similarly, $D'_r = (S'_r, I, H'_r)$, from D', is defined to establish an isomorphism between D'_r and N'. Then, for every function γ: $V \to V'$, the function β_r: $S_r \to S'_r$ is defined such that $\beta_r(\{v\}) = \{\gamma(v)\}$ for all $v \in V$. Thus, γ is an isomorphism from G to G' (or from N to N') iff β_r is an isomorphism from D_r to D'_r. Consequently, Theorem 2.4.2 is easily proved based on β_r rather than γ because it is easier to extend β_r. The proof is now given.

Proof. The function β_r: $S_r \to S'_r$ is extended in such a way that β:$S \to S'$ satisfying

$$\beta(s) = \bigcup_{v \in s} \beta_r(\{v\}) \text{ for all } s \in S \qquad (2.4.6)$$

Suppose that β_r is an isomorphism from D_r to D'_r. Then β_r is invertible and preserves state transitions in D_r and D'_r, i.e.,

$$\bigcup_{u \in H_r(\{v\},i)} \beta_r(\{u\}) = H'_r(\beta_r(\{v\}),i) \text{ for all } \{v\} \in S_r \qquad (2.4.7)$$

where the union on the left-hand side is equal to $\beta(H_r(\{v\},i))$.

To show that β is one-to-one, let s_1 and s_2 be any states in S satisfying $\beta(s_1) = \beta(s_2)$. Then (2.4.6) implies that

$$\bigcup_{v_x \in s_1} \beta_r(\{v_x\}) = \bigcup_{v_y \in s_2} \beta_r(\{v_y\}).$$

Since β_r is one-to-one, it must be true that $s_1 = s_2$.

To show that β is onto, H_r is extended in such a way that $H_r: S_r \times I^* \to S$ such that $H_r(\{v\},\lambda) = \{v\}$, and $H_r(\{v\},wi) = F(v,wi)$. Similarly, H'_r is extended. Then, let $H'_r(\{v'\},w)$ for some $(\{v'\},w) \in S'_r \times (I^* - \{\lambda\})$ be a state in $S' - S'_r$. Since β_r is onto, there exists at least one state $\{u\}$ in S_r such that $\beta_r(\{u\}) = \{v'\}$. Thus, $H'_r(\{v'\},w) = \beta(H_r(\{u\}, w))$.

To show that β preserves state transitions, for any $s \in S$,

$$\begin{aligned}
\beta(H(s,i)) &= \beta\Big(\bigcup_{a \in s} F(a,i)\Big) \\
&= \bigcup_{a \in s} \beta(F(a,i)) \\
&= \bigcup_{a \in s} \beta(H_r(\{a\},i)) \\
&= \bigcup_{a \in s} H'_r(\beta_r(\{a\}),i)
\end{aligned}$$

$$= H'\Big(\bigcup_{a \in s} \beta_r(\{a\}), i\Big)$$

$$= H'(\beta(s), i)$$

The above equalities are established by applying (2.2.6), β being invertible, (2.4.5), (2.4.7), β being invertible again, and (2.4.6). Thus, β defines an isomorphism from D to D'.

On the other hand to show that the converse may not be true, suppose β is an isomorphism from D to D' and β_r is its restriction from D_r onto D'_r. If $\beta(s) \neq \bigcup_{u \in s} \beta(\{u\})$ for some next state

$s = H(\{v\}, i)$, then

$$\bigcup_{u \in H(\{v\},i)} \beta(\{u\}) \neq H'(\beta(\{v\}), i) \tag{2.4.8}$$

although

$$\beta(H(\{v\}, i)) = H'(\beta(\{v\}), i) \tag{2.4.9}$$

Comparing (2.4.8) with (2.4.7), it is clear that the restriction β_r does not preserve all state transitions in D_r and D'_r. Thus, the converse of Theorem 2.4.2 may not be true. Q.E.D.

As a direct consequence of Theorems 2.4.1 and 2.4.2, the following corollary is obtained.

Corollary 2.4.1. If there is an isomorphism γ between two given graphs G and G', then there exists an isomorphism γ between their corresponding DSM's D and D'.

This corollary provides only a necessary condition for two graphs being isomorphic. By adding an additional requirement, a necessary and sufficient condition for the existence of an isomorphism between two graphs can be stated as the following corollary.

Corollary 2.4.2. An isomorphism γ between two given graphs G and G' exists iff there is an isomorphism β between their corresponding DSM's D and D' and

$$H'(\beta(\{v\}),i)) = \bigcup_{u \in H(\{v\},i)} \beta(\{u\}) \tag{2.4.10}$$

for all states $\{v\}$ in S_r.

Proof. For proving Corollary 2.4.2, it needs only to show that if β is an isomorphism between the DSM's D and D', and (2.4.10) is satisfied, then the restriction β_r of β is an isomorphism between NDSM D_r and D'_r. Suppose that β is an isomorphism between the DSM's D and D'. Then the restriction β_r of β is obviously invertible and $\beta(H_r(\{v\},i)) = H'_r(\beta_r(\{v\}),i)$ for all states in S_r as easily seen from (2.4.9). If (2.4.10) is also satisfied for all $\{v\}$ in S_r, then (2.4.7) holds for all $\{v\}$ in S_r. Then β_r preserves all state transitions in NDSM's D_r and D'_r. Q.E.D.

Definition 2.4.1. A function $\alpha: S \to S'$ is called an isomorphism from M to M' if α is invertible and preserves state transitions and outputs, i.e., for all $\{v\}$ in S,

$$\alpha(H(\{v\},i)) = H'(\alpha(\{v\}),i)$$

and

$$J(H(\{v\},i)) = J'(H'(\alpha(\{v\}),i)). \tag{2.4.11}$$

Since every isomorphism between two MSM's must not only preserve state transitions but also outputs as defined in (2.4.11), it is obvious that this additional requirement implies that the set of all isomorphisms between two MSM's M and M' constructed from DSM's D and D' is a subset of that between D and D'. Thus, the following corollary which is a consequence of Corollary 2.4.3 is stated without proof.

Corollary 2.4.3. An isomorphism γ between two given graphs G and G' exists iff there is an isomorphism α between their corresponding MSM's M and M', and

$$H'(\alpha(\{v\}),i) = \bigcup_{u \in H(\{v\},i)} \alpha(\{u\}) \qquad (2.4.12)$$

for all $\{v\}$ in S_r.

Thus, either Corollary 2.4.2 or 2.4.3 can be used as a necessary and sufficient condition for graph isomorphism. The advantage of using Corollary 2.4.3 instead of Corollary 2.4.2 occurs when the number of isomorphisms between M and M' is fewer than those between D and D'. However, the disadvantage of using Corollary 2.4.3 is the additional requirements of defining outputs and of checking condition (2.4.11). These and other considerations are discussed in Sections 2.5 and 2.6.

In order to determine any isomorphism γ between two graphs, all isomorphisms α between the two corresponding MSM's must first be found. Thus, before presenting the graph isomorphism algorithm, a discussion of the partitioning method which is used to determine all isomorphisms between two MSM's is presented.

2.5 Partitioning on the State Set of a Moore Sequential Machine

Definition 2.5.1. A partition P over the state set S of a MSM M is a set of pairwise, disjoint sets called blocks such that the union of all blocks B is S.

Definition 2.5.2. A partition P is called closed and output-consistent if $H(B_j,i) \subseteq B_k$ and $K(J(B_j)) = 1$,

for each $(B_j,i) \in P \times I$ and some $B_k \in P$, where

$$H(B_j,i) = \bigcup_{s \in B_j} \{H(s,i)\} \text{ and } J(B_j) = \bigcup_{s \in B_j} \{J(s)\}.$$

Definition 2.5.3. A partition P is called trivial if K(P) = K(S) or K(P) = 1; otherwise, it is called nontrivial.

Yang (1975) determined all isomorphisms between two MSM's by using a modified version of his earlier method (Yang, 1974) in which all closed partitions over the state set of a sequential machine were generated. The modified method first constructed a MSM $M^* = (S \cup S', I, H^*, O \cup O', J^*)$ where

$$H^*(s^*,i) = \begin{cases} H(s^*,i) \text{ if } s^* \notin S \\ H'(s^*,i) \text{ if } s^* \in S' \end{cases}$$

and

$$J^*(s^*) = \begin{cases} J(s^*) \text{ if } s^* \notin S \\ J'(s^*) \text{ if } s^* \in S'. \end{cases}$$

Next, all nontrivial closed and output-consistent partitions were generated by taking the union of the closure classes of the subsets $\{s,s'\}$ for each $(s,s') \notin S \times S'$.

Definition 2.5.4. The closure class of a subset $\{s,s'\}$ is the set containing 1) $\{s,s'\}$, 2) all sets $\{s_k,s_k'\} = H(\{s,s'\},i)$ such that $\{s_k,s_k'\} \not\subseteq \{s,s'\}$, and 3) all distinct sets $\{s_r,s_r'\} = H(\{s,s'\},w)$ such that $\{s_r,s_r'\} \not\subseteq \{s,s'\}$ and $\{s_r,s_r'\} \not\subseteq \{s_k,s_k'\}$ for all w.

If the union of the closure classes were closed, then the union defined a partition and consequently, defined an isomorphism α between the MSM's M and M'.

Since a modified version of the above partitioning method is used by the graph isomorphism algorithm, an illustration of the method is presented in Section 2.6.

2.6 The Graph Isomorphism Algorithm

The necessary and sufficient condition of Corollary 2.4.3 is implemented by Algorithm 2 which is presented below. Algorithm 2 determines all isomorphisms between two MSM's and then, determines all isomorphisms between the corresponding graphs by checking the state transitions for all states s in S_r using (2.4.12). The algorithm finds all isomorphisms between two MSM's by using a modified version of the partitioning method which was developed by Yang. In Algorithm 2, only those closed and output-consistent partitions which are generated by the union of the closure classes of the subsets $\{s,s'\}$ for each $(s,s') \in S_r \times S_r'$ are considered in determining if an isomorphism exists between two MSM's.

First, Algorithm 2 is presented and illustrated by an example. Then, the computational complexity of Algorithm 2 is discussed

2.6.1 Algorithm 2

Algorithm 2 determines all isomorphisms between two graphs G and G'. The MSM's M and M' are represented by the arrays H,J and H',J', respectively. These arrays are created by two successive calls to Algorithm 1. The two graphs G and G' are respectively represented by an adjacency list GL and an adjacency matrix GA'. By using different data structure for each graph, the algorithm can easily check (2.4.12). Other data structures which are used by Algorithm 2 are: the matrix Class to contain all output-consistent closure classes for all subsets $\{s,s'\}$ for each (s,s') in $S_r \times S_r'$; the matrix Candidate to hold all possible closed and output-consistent partitions which require checking for covering condition; the matrix Partition to define all isomorphisms

between M and M'; the array Gamma to define all isomorphisms between G and G' (all isomorphisms of Partition which satisfy (2.4.12); and the variables si and si' to index the states of S and S' respectively.

Step 1. (If the numbers of states in M and M' represented respectively by K(S) and K(S') are not equal, then no isomorphism exists.)
If $K(S) \neq K(S')$ then no isomorphism exists, stop.

Step 2. If every output in J does not have a corresponding output in J' then no isomorphism exists, stop.

Step 3. (Steps 3-11 generate the output-consistent closure classes for all subsets {si,si'} for each $(s,s') \in S_r \times S'_r$.)
si ← 1, do Steps 4-11 until si > K(V), go to Step 12.

Step 4. si' ← 1, do Steps 5-10 until si' > K(V'), go to Step 11.

Step 5. (In order to know the subset {si,si'} whose closure class is being generated, the indices iptr and jptr must be used.)
iptr ← si, jptr ← si'.

Step 6. (If the outputs of the states represented by iptr and jptr are not equal, the two states are not output-consistent.)
If $J_{iptr} \neq J'_{jptr}$ then go to Step 9.

Step 7. (Otherwise, the states represented by iptr and jptr are output-consistent and become an element of the output-consistent closure class of {si,si'}. If (iptr,jptr) is already an element of {si,si'} closure class, then all elements of the closure class have been generated and the closure class for {si,si'+1} must be generated.)
If (iptr,jptr) is already an element of $Class_{si,si'}$ then go to Step 10.

Step 8. (Otherwise, (iptr,jptr) becomes an element of the closure class of {si,si'}. Since this is not the last element of the closure class, the indices for the next states of iptr and jptr are determined, and the process of checking for output-consistency begins again.)

Place (iptr,jptr) in $Class_{si,si'}$, iptr $\leftarrow H_{iptr}$, jptr $\leftarrow H'_{jptr}$, go to Step 6.

Step 9. (In order to indicate that the state of S_r represented by si and the state of S'_r represented by si' do not have an output-consistent closure class, $Class_{si,si'}$ is set to zeros.)

$Class_{si,si} \leftarrow (0,0)$.

Step 10. si' $\leftarrow$ si' + 1.

Step 11. si $\leftarrow$ si + 1.

Step 12. (Steps 12-13 define all possible closed and output-consistent candidate partitions by taking the union of all output-consistent closure classes and by checking the covering condition.)

t $\leftarrow$ 0, n $\leftarrow$ 0.

Step 13. t $\leftarrow$ t + 1.

If there does not exists a

$$Candidate_t = \cup \{u\text{-}v \mid (u,v) \in Class_{si,si'} \text{ for } Class_{si,si'} \neq (0,0) \wedge u \neq u_1 \wedge v \neq v_1, \text{ for any } u_1\text{-}v_1 \in Candidate_t\}$$

where $Candidate_t \neq Candidate_m$ $(1 \leq m \leq t)$, for $1 \leq si \leq K(V)$ and $1 \leq si' \leq K(V')$

then go to Step 15.

Step 14. (Check each candidate partition for covering condition. If the covering condition is satisfied, then the candidate partition becomes a partition which defines an isomorphism between M and M'.)

If for all $1 \leq si \leq K(S)$, $si\text{-}si' \in Candidate_t$

then $n \leftarrow n + 1$, $Partition_n \leftarrow Candidate_t$.

Go to Step 13.

Step 15. (This step determines if each Partition generated in Step 14 satisfies (2.4.12) of Corollary 2.4.3. Each Partition which satisfies (2.4.12) induces an isomorphism between G and G'.)

For each $1 \leq m \leq n$

if for all $1 \leq si \leq K(V)$

$$GA'(si',sl') = \begin{cases} 1, \text{ for all } sk \in GL_{si} \\ 0, \text{ otherwise} \end{cases}$$

where GL_{si} is the adjacency list for vertex si and $si\text{-}si'$ and $sk\text{-}sl'$ are elements of $Partition_m$

then $Gamma \leftarrow \{si\text{-}si' \mid si\text{-}si' \in Partition_m, 1 \leq si \leq K(V)\}$ defines and isomorphism between G and G'.

Step 16. Stop.

For illustrating Algorithm 2, graph 7 of Figure 2.3.1.1 and graph 8 of Figure 2.6.1.1 are used. The corresponding MSM's M and M' which are generated by Algorithm 1 are shown in Table 2.3.1.1 and Table 2.6.1.1. After determining in Step 1 that the number of states of M is equal to the number of states of M', Algorithm 2 proceeds to check the outputs of the MSM's. Step 2 determines each output in M has a corresponding output in M'. In Steps 3-11, the output-consistent

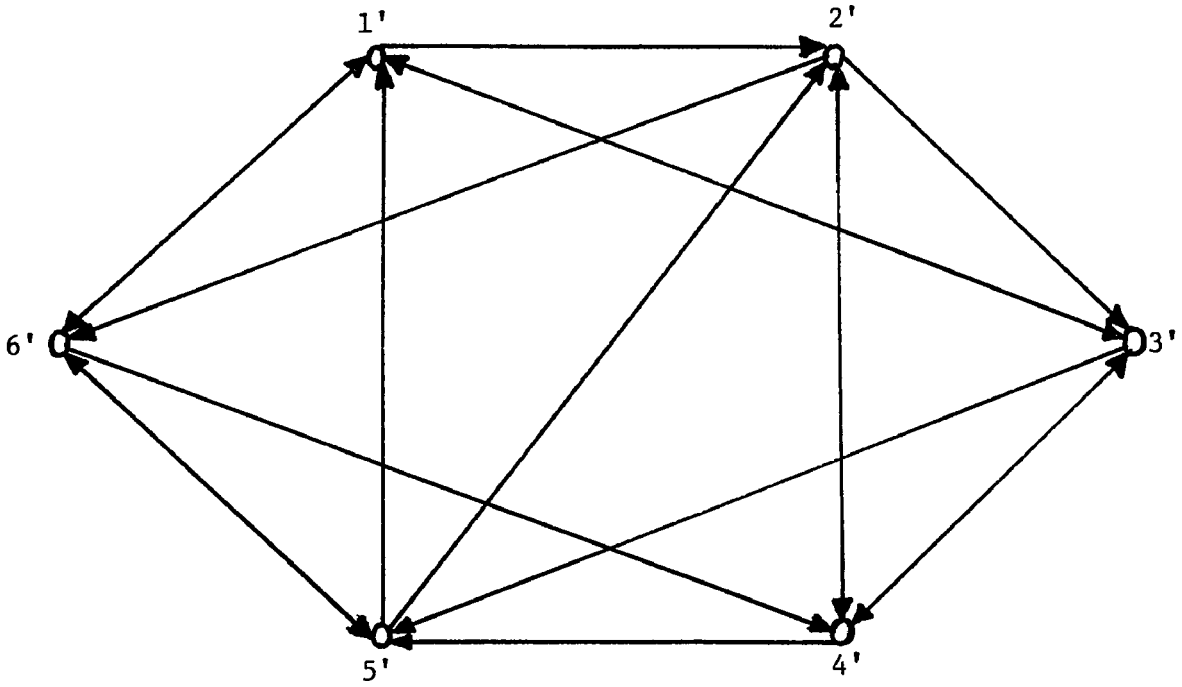


Figure 2.6.1.1 Graph 8

TABLE 2.6.1.1

MSM M' CORRESPONDING TO GRAPH 8

Index si'	State s'	Next State H'(s',i)	Output J'(s')
1'	{1'}	{2',3',6'}	[[6],[2]]
2'	{2'}	{3',4',6'}	[[6],[1]]
3'	{3'}	{1',4',5'}	[[7],[2]]
4'	{4'}	{2',3',5'}	[[6],[2]]
5'	{5'}	{1',2',6'}	[[6],[1]]
6'	{6'}	{1',4',5'}	[[6],[2]]
7'	{2',3',6'}	{1',3',4',5',6'}	[[3],[1,2,2]]
8'	{3',4',6'}	{1',2',3',4',5'}	[[3],[2,2,2]]
9'	{1',4',5'}	{1',2',3',5',6'}	[[4],[1,2,2]]
10'	{2',3',5'}	{1',2',3',4',5',6'}	[[4],[1,1,2]]
11'	{1',2',6'}	{1',2',3',4',5',6'}	[[3],[1,2,2]]
12'	{1',3',4',5',6'}	{1',2',3',4',5',6'}	[[2],[1,2,2,2,2]]
13'	{1',2',3',4',5'}	{1',2',3',4',5',6'}	[[2],[1,1,2,2,2]]
14'	{1',2',3',5',6'}	{1',2',3',4',5',6'}	[[2],[1,1,2,2,2]]
15'	{1',2',3',4',5',6'}	{1',2',3',4',5',6'}	[[1],[1,1,2,2,2,2]]

closure classes for $(s,s') \in S_r \times S'_r$ are generated. Indices si and si' are used to represent each state of S and S'. As an example, the generation of the closure classes for {5,si'} is illustrated in Figure 2.6.1.2. The downward arrows represent the next state function with the outputs of the states written in parentheses. For state 5, the only output-consistent closure class is generated from {5,5'}. Similarly, the closure classes for the other $(s,s') \in S_r \times S'_r$ are generated and are shown in Table 2.6.1.2. The only candidate which is also the only closed and output-consistent partition generated in Steps 12-14 is Partition_1 = {1-6',2-1',3-2',4-3',5-5',6-4',7-9',8-7',
9-8',10-11',11-10',12-14',13-12',14-13',
15-15'}

This partition defines an isomorphism between M and M', and also, by satisfying the condition of Step 15, defines an isomorphism between G and G'. The isomorphism between G and G' is
Gamma = {1-6',2-1',3-2',4-3',5-5',6-4'}.

As seen from this example, when using the MSM representation of graphs 7 and 8, 6 equations of (2.4.12) are checked for the one partition generated in Step 14. If the DSM representation is used, $2 \cdot 4!$ or 48 equations of (2.4.12) require checking. Of course, the time required to check the 48 equations must be weighed against the time required to define and check the outputs for M and M'. However, as the K(V) increases, it becomes impractical to store and check a factorial number of equations. Thus, it is obvious that the MSM, rather than the DSM, representation of a graph leads to a more practical and efficient graph isomorphism algorithm.

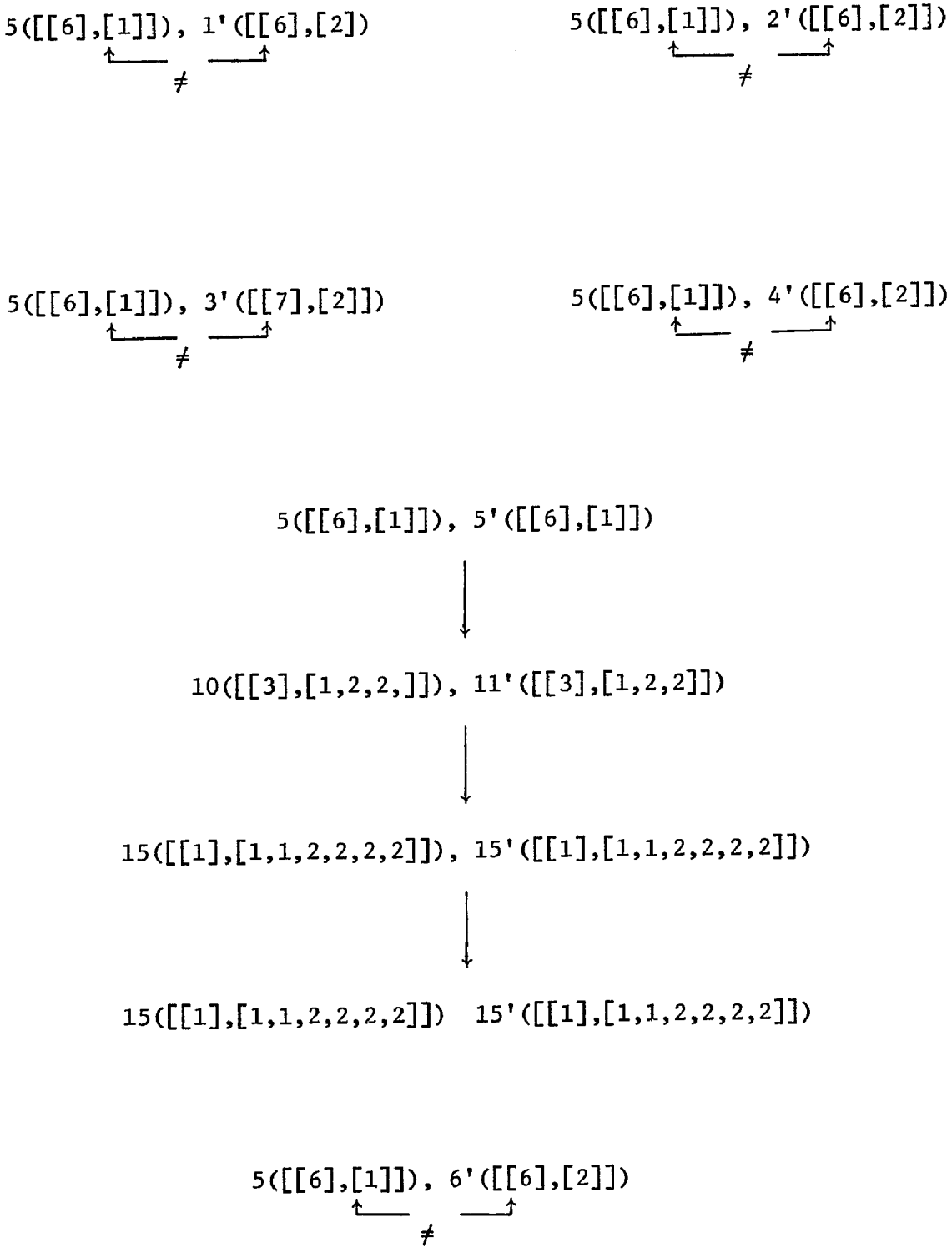


Figure 2.6.1.2 Generation of all Closure Classes for {5,si'}

TABLE 2.6.1.2

OUTPUT-CONSISTENT CLOSURE CLASSES FOR $(s,s') \in S_r \times S'_r$

Index Pair (si,si')	Output-Consistent Closure Classes
(1,6')	{(1,6'),(7,9'),(12,14'),(15,15')}
(2,1')	{(2,1'),(8,7'),(13,12'),(15,15')}
(3,2')	{(3,2'),(9,8'),(14,13'),(15,15')}
(4,3')	{(4,3'),(7,9'),(12,14'),(15,15')}
(5,5')	{(5,5'),(10,11'),(15,15')}
(6,4')	{(6,4'),(11,10'),(15,15')}

2.6.2 The Computational Complexity of Algorithm 2

The worst case for Algorithm 2 exists when the outputs of the states of the MSM's are all the same. If the outputs are the same, the number of output-consistent closure classes for the states of the MSM's are equal to the number of closure classes for the states of the corresponding DSM's.

Theorem 2.6.2.1. Algorithm 2 processes a pair of graphs in at most $O(n^2 \cdot n!)$ time.

Proof. If the outputs of the states of the MSM's are the same, there can be up to n^2 output-consistent closure classes generated in Steps 4-11 of Algorithm 2. In Steps 12-14, these n^2 closure classes can define up to n! isomorphisms between the MSM's. Each of these n! isomorphisms can have at most n equations of (2.4.12) to be checked. Each equation can have up to n terms. Thus the upper bound of Algorithm 2 is $O(n^2 \cdot n!)$. Q.E.D.

The lower bound for Algorithm 2 can be determined from the special case that exists when the graphs are not isomorphic, and the corresponding MSM's reflect this fact by having a differing number of states. The algorithm terminates in Step 1; and the lower bound is 1.

Since the upper bound and lower bound of any algorithm reflect the worst and best cases, they are of theoretical interest only. Because many practical problems contain neither the worst nor the best cases, it is very important to obtain experimental bounds. Experimental bounds for both Algorithms 1 and 2 are discussed in Chapter V.

CHAPTER III

A REVIEW OF THE CURRENT BACKTRACKING GRAPH ISOMORPHISM ALGORITHMS

3.1 Introduction

In recent years, the most popular approach for solving the graph isomorphism problem has been the use of the backtracking technique. Backtracking algorithms used some necessary conditions to partition the sets of vertices of the two graphs and the backtracking technique to select possible vertex assignments to test for isomorphism. The backtracking approach represented an improved method over the heuristic approach, since if the heuristic part of the backtracking algorithm failed to reduce the number of possible vertex assignments, the backtracking technique insured a stepwise elimination of all inconsistent vertex assignments. Since backtracking algorithms could process graphs of large orders, this approach also represented an improvement over the coding algorithms.

In this Chapter, the graph isomorphism backtracking algorithms which were developed by Berztiss (1973), Ullmann (1976), and Schmidt and Druffel (1976) are reviewed. Each such algorithm is described and illustrated by using graph 7 of Figure 2.3.1.1. and graph 8 of Figure 2.6.1.1.

3.2 Berztiss' Backtracking Algorithm

This method specified the graph G by a linear notation called a K-formula and then derived a similar K-formula for the graph G' by

using permissible K-formula transformations, a condition necessary for graph isomorphism, and a backtracking procedure. If the K-formulas which represented G and G' had the same pattern, then the corresponding vertices defined an isomorphism.

The K-formula notation was based on the representation of an arc (a,b) by the K-formula *ab. The representation *ab was derived by an application of K-operator * to the vertices named a and b. For instance, the arc (1,2) of graph 7 can be represented by *12. In general, a K-formula, which represented n arcs originating from a given vertex in a graph, consisted of n K-operators, followed by the name of the given vertex, followed in turn by the names of the n vertices at which the arcs terminated. Using the K-operator * in this way, the adjacency relation of a vertex could be completely specified. For example, vertex 1 of graph 7 can be specified by the K-formula ***1256. Thus, graph 7 can be completely specified by the set of K-formulas ***1256, ***2134, ***3146, ***4256, ***5123, ***6345. By using the switch transformation, the terminal vertices could be written in any order, i.e., ***1256 can be rewritten as ***1562.

The K-formulas could then be combined, using the substitution transformation by which vertex in a K-formula could be substituted by its respective K-formula. Thus, the vertex 2 in ***1256 can be substituted by ***2134 to give ***1***213456. Continuing in this way, the K-formulas of graph 7 are transformed to yield a single K-formula ***1***21***31***42***5123***63456456.

Formally, Berztiss defined a K-formula as (1) a single vertex symbol or (2) $*F_1F_2$ where F_1 and F_2 are K-formulas. The algorithm published by Berztiss (1973) which describes the above process of

generating the set of K-formulas which represent a graph is given in Appendix E.1 as Algorithm 3. From the definition, it is noted that a K-formula of a graph usually consist of several K-formulas which are called K-subformulas. In order to establish whether a given string of a K-formula is a K-subformula, Berztiss used an equivalent iterative definition of a K-formula.

Definition 3.2.1. The string $s_1 s_2 \ldots s_i \ldots s_m$, consisting of K-operators and vertex symbols and containing a substring $s_1 \ldots s_i$ having k_i K-operators and n_i vertex symbols, is a K-formula iff $n_i \leq k_i$ for $i = 1,2,\ldots,m-1$ and $n_m = k_m + 1$.

Processing the K-formula of graph G from left to right, the backtracking algorithm of Berztiss would attempt to construct a partial K-subformula of graph G' that was isomorphic to the K-subformula of G, which was defined by the processed vertices. If on processing the next vertex in the K-formula of G, it was found that the K-subformula of G' could not be extended on the basis of the current vertex correspondence, then the procedure backtracked to the vertex in the K-formula of G that last caused an addition to the tentative vertex correspondences, and a different vertex of G' would be chosen to correspond to the vertex of G. The procedure would continue in this manner, until a complete K-formula of G' was generated, in which case an isomorphism existed; or, until no feasible vertex correspondence existed. Berztiss' backtracking algorithm (1973) is given in Appendix E.1 as Algorithm 5. Algorithm 4 (Berztiss, 1973) of Appendix E.1 is used to construct the data structures used in Algorithm 5.

As an illustration of this backtracking algorithm, the K-formula of graph 8 is constructed to have the same pattern as the K-formula of

graph 7. First, graph 8 can be completely specified by the set of K-formulas {***1'2'3'6', ***2'3'4'6', ***3'1'4'5', ***4'2'3'5', ***5'1'2'6', ***6'1'4'5'}. Since all the K-formulas of graph 8 match the first subformula ***5123 of graph 7, each of the K-formulas of graph 8 are extended, by using the switch and substitute transformations, in an effort to produce a K-formula that has the same pattern as the next subformula ***42***5123***6345. After a number of vertex correspondences and backtracks, in an attempt to extend the first two K-formulas of graph 8, the procedure successfully extends the K-formula ***3'1'4'5' to ***3'1'***5'2'1'6'***4'2'3'5' based on the vertex correspondence 2-1', 4-3', 5-5', and 6-4'. Using these correspondences, the K-formula is further extended to ***2'5'***3'1'***5'6'1'2' ***4'2'3'5'4' in order to match the next subformula ***31***42***5123 ***63456. From these K-formulas the vertex correspondences 3-2' and 1-6' are made. Using these correspondences the K-formula is extended to

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

to match the next subformula

2131***42***5123***634564.

Finally, the K-formula of graph 8 is extended to

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

to match the K-formula of graph 7

121***31***42***5123***63456456.

Since the two patterns are the same, the corresponding isomorphism γ = (1-6',2-1',3-2',4-3',5-5',6-4') is defined.

3.3 Ullmann's Refinement/Backtracking Algorithm

The basic idea of this method was to construct a matrix M which represented possible vertex assignments between the vertices of the two graphs G and G', and then, after making possible vertex assignments, to refine M by using a necessary condition for graph isomorphism. The necessary condition was based on the adjacency relations of the vertices. If the graphs were isomorphic, then M would, after possible backtracking to reassign vertices, be refined to a matrix which specified a one-to-one mapping from V onto V'.

The initial matrix M which represented possible vertex correspondences was constructed according to

$$m_{ij} = \begin{cases} 1, \text{ if the indegree and outdegree of vertex } i \text{ of } G \text{ is the same as the indegree and outdegree of vertex } j \text{ and } G', \\ 0, \text{ otherwise,} \end{cases}$$

where $M = [m_{ij}]$. For example, the matrix M for graphs 7 and 8 is constructed. Since the indegree and outdegree of each vertex of graphs 7 and 8 are both three, the construction of M results in a matrix having 1's for all entries as is shown in Table 3.3.1.

The constructed matrix M would then be used in making the initial vertex assignments. After a vertex of G was assigned to one of G', the remaining entries in the corresponding row of M were set to 0's, and M would be refined as follows: $m_{ij} = 1$ was changed to $m_{ij} = 0$ unless

$$(\forall x)_{1 \le x \le K(V)} \left((g_{ix} = 1) \Rightarrow (\exists y)_{1 \le y \le K(V')} (m_{xy} \cdot g'_{jy} = 1) \right) \qquad (3.3.1a)$$

and

$$(\forall x)_{1 \le x \le K(V)} \left((g_{xi} = 1) \Rightarrow (\exists y)_{1 \le y \le K(V')} (m_{xy} \cdot g'_{yj} = 1) \right). \qquad (3.3.1b)$$

TABLE 3.3.1

INITIAL MATRIX M FOR GRAPHS 7 AND 8

	1'	2'	3'	4'	5'	6'
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	1	1	1	1	1

where $[g_{ij}]$ and $[g'_{ij}]$ represented the adjacency matrices of graphs G and G'. Conditions (3.3.1a) and (3.3.1b) represented the adjacency relations of the vertices of G and G'. From the definition of graph isomorphism, it is necessary that if v_i of V corresponded to v'_j of V' in an isomorphism, then for each vertex v_{ix} adjacent to v_i, there must exist a v'_{jy} that is adjacent to v'_j such that $\gamma(v_{ix}) = v'_{jy}$, and for each v_{xi} to which v_i is adjacent, there must exist a v'_{yj} to which v'_j is adjacent such that $\gamma(v_{xi}) = v'_{yj}$.

The refinement procedure would continue refining M until no more m_{ij}'s were changed or until a row of M became all 0's, in which case the last vertex assignment was inconsistent. Ullmann's algorithm used a depth first tree search method in making vertex assignments. In general, a depth d corresponded to row d of the matrix M. The refinement procedure, by eliminating some of the 1's of M, eliminated successor nodes in the tree search. If at level $d > 1$, all vertex assignments were found inconsistent, i.e., the refined M contained a zero row, the algorithm would then backtrack to depth d-1. Using the matrix M at level d-1, the procedure would try another vertex assignment of vertex d-1 of G. If a level d, matrix M was left unchanged by the refinement procedure, then G would be concluded isomorphic to G'. However, if upon backtracking to depth 1, no more possible vertex assignments could be made, then G would be concluded nonisormophic to G', and the algorithm would terminate. Ullmann's refinement/backtracking algorithm (1976) is given as Algorithm 6 of Appendix F.1.

As an illustration of Ullmann's isomorphism algorithm, the matrix M of Table 3.3.1 is used in making a vertex assignment for vertex 1 of

graph 7. Since $m_{11'} = 1$, the procedure first tries vertex assignment 1-1'. This assignment causes the other entries in row 1 of M to become 0's as is shown in Table 3.3.2. The initial matrix M of Table 3.3.2. is refined until the refined M of Table 3.3.2 is obtained. This refined M indicates that 1-1' is inconsistent, since it implies no vertex correspondence exists for vertex 5 (row 5 being all 0's). The algorithm then attempts based on the last valid M, in this case the matrix of Table 3.3.1, to assign vertex 1 to another vertex of graph 8. The assignments 1-2', 1-3', 1-4', and 1-5' all produce a refined M with a zero row, and thus, these assignments are all inconsistent. These vertex assignments and the corresponding refined M is shown in Table 3.3.3. However, the last possible vertex assignment 1-6' results in the refined M of Table 3.3.4. Based on this matrix, the vertex assignments 2-1', 3-2', 4-3', 5-5', and 6-4' are made, and since at each level M remains unchanged, the matrix M of Table 3.3.4 is the desired matrix which specifies the isomorphism γ = (1-6',2-1',3-2', 4-3',5-5',6-4'). For this example, no backtracking was required.

3.4 Schmidt and Druffel's Backtracking Algorithm

This method used information contained in the distance matrix representation of a graph initially to partition the sets of vertices of graphs G and G' into like classes. Next, a backtracking procedure selected vertices from like classes and then, based on the distance matrix information, checked the vertex assignments for consistency. The algorithm terminated when the two graphs were found either isomorphic or nonisomorphic.

TABLE 3.3.2

MATRIX M AFTER VERTEX ASSIGNMENT 1-1'

Initial M for graphs 7 and 8

	1'	2'	3'	4'	5'	6'
1	1	0	0	0	0	0
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	1	1	1	1	1

Refined M for graphs 7 and 8

	1'	2'	3'	4'	5'	6'
1	1	0	0	0	0	0
2	0	0	1	0	0	1
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	0	0	0	0	0
6	1	1	1	1	1	1

TABLE 3.3.3

INCONSISTENT VERTEX ASSIGNMENTS FOR GRAPHS 7 AND 8

Vertex Assignment	Corresponding Refined M
1-2'	$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
1-3'	$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
1-4'	$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$
1-5'	$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$

TABLE 3.3.4

MATRIX M AFTER REFINEMENT OF VERTEX ASSIGNMENT 1-6'

	1'	2'	3'	4'	5'	6'
1	0	0	0	0	0	1
2	1	0	0	0	0	0
3	0	1	0	0	0	0
4	0	0	1	0	0	0
5	0	0	0	0	1	0
6	0	0	0	1	0	0

M defines the isomorphism

γ = (1-6',2-1',3-2',4-3',5-5',6-4')

For every pair of vertices v_i and v_j of a graph there is a unique minimum distance which can be represented by a distance matrix.

Definition 3.4.1. The distance matrix D of graph G having n vertices is an n x n matrix in which the element d_{ij} represents the shortest path between vertices v_i and v_j. If i = j, then $d_{ij} = 0$. If no path exists between v_i and v_j, then $d_{ij} = \infty$ (infinity).

Since the distance matrices were unique representations of graphs G and G' and since they contained information concerning the relationship between the vertices of the graphs, they were used initially to partition the sets of vertices into like classes. By comparing the row characteristic matrices with the column characteristic matrices, the initial partitioning procedure formed the characteristic matrices of G and G'.

Definition 3.4.2. A row characteristic matrix XR is an n x (n-1) matrix such that the element xr_{im} is the number of vertices which are at a distance m away from v_i.

Definition 3.4.3. A column characteristic matrix XC is an n x (n-1) matrix such that each element xc_{im} is the number of vertices from which v_i is at a distance m.

Definition 3.4.4. A characteristic matrix X is formed by composing the corresponding rows of XR and XC.

The initial partition was then generated from the characteristic matrices by assigning the same class to all vertices having identical rows. Class vectors C and C' were used to contain the classes for each vertex of G and G'. The initial partitioning algorithm is given as Algorithm 8 (Schmidt and Druffel, 1976) of Appendix G.1.

To illustrate the formation of the initial partition, graphs 7

and 8 are used. First, using Definition 3.4.1, the distance matrices D and D' are constructed. These matrices are shown in Table 3.4.1. Using Definitions 3.4.2 and 3.4.3, the matrices XR, XC, XR', and XC' are generated and are shown in Tables 3.4.2 and 3.4.3. Next, the characteristic matrices X and X' are generated by composing the corresponding rows of XR with XC and XR' with XC'. The resulting matrices are shown in Table 3.4.4. Since all the rows of X and X' are generated by composing the corresponding rows of XR with XC and XR' with XC'. The resulting matrices are shown in Table 3.4.4. Since all the rows of X and X' are identical, the initial partition defined by C = (1,1,1,1,1,1) and C' = (1,1,1,1,1,1), consist of just one class.

After the initial partition was generated, the graph isomorphism algorithm selected two vertices belonging to the smallest class and checked to determine if the assignment was consistent. The assignment of vertex i of G to vertex j of G' was consistent if (1) every element $d_{ir} = d'_{js}$ and $d_{ri} = d'_{sj}$, where vertex r had been assigned to vertex s and (2) every element d_{ik} (vertex k not previously assigned) had a corresponding d'_{jp} (vertex p not previously assigned) such that $c_k = c_p$, where c_k and c_p represent the class of vertices k and p. Thus, a consistent vertex assignment implied that row i and column i of D had corresponding elements in row j and column j of D', at least for all previously assigned vertices, and that the remaining elements of those rows and columns did not preclude further consistent vertex assignments, if any remained. This condition was checked by composing the class vector for the appropriate graph with the respective rows and columns of the distance matrix. This composition generated a new class vector which could define a refined partition. If the vertex

TABLE 3.4.1

DISTANCE MATRICES D AND D' FOR GRAPHS 7 AND 8

D

	1	2	3	4	5	6
1	0	1	2	2	1	1
2	1	0	1	1	2	2
3	1	2	0	1	2	1
4	2	1	2	0	1	1
5	1	1	1	2	0	2
6	2	2	1	1	1	0

D'

	1'	2'	3'	4'	5'	6'
1'	0	1	1	2	2	1
2'	2	0	1	1	2	1
3'	1	2	0	1	1	2
4'	2	1	1	0	1	2
5'	1	1	2	2	0	1
6'	1	2	2	1	1	0

TABLE 3.4.2

ROW CHARACTERISTIC AND COLUMN CHARACTERISTIC MATRICES FOR GRAPH 7

XR

	1	2	3	4	5
1	3	2	0	0	0
2	3	2	0	0	0
3	3	2	0	0	0
4	3	2	0	0	0
5	3	2	0	0	0
6	3	2	0	0	0

XC

	1	2	3	4	5
1	3	2	0	0	0
2	3	2	0	0	0
3	3	2	0	0	0
4	3	2	0	0	0
5	3	2	0	0	0
6	3	2	0	0	0

TABLE 3.4.3

ROW CHARACTERISTIC AND COLUMN CHARACTERISTIC MATRICES FOR GRAPH 8

XR'

	1	2	3	4	5
1'	3	2	0	0	0
2'	3	2	0	0	0
3'	3	2	0	0	0
4'	3	2	0	0	0
5'	3	2	0	0	0
6'	3	2	0	0	0

XC'

	1	2	3	4	5
1'	3	2	0	0	0
2'	3	2	0	0	0
3'	3	2	0	0	0
4'	3	2	0	0	0
5'	3	2	0	0	0
6'	3	2	0	0	0

TABLE 3.4.4

CHARACTERISTIC MATRICES FOR GRAPHS 7 AND 8

X						X'					
	1	2	3	4	5		1	2	3	4	5
1	33	22	00	00	00	1'	33	22	00	00	00
2	33	22	00	00	00	2'	33	22	00	00	00
3	33	22	00	00	00	3'	33	22	00	00	00
4	33	22	00	00	00	4'	33	22	00	00	00
5	33	22	00	00	00	5'	33	22	00	00	00
6	33	22	00	00	00	6'	33	22	00	00	00

assignment was inconsistent, another pair of vertices was chosen. If a partition is reached such that there are no consistent vertex assignments, then an assignment between two vertices which did not belong to an isomorphism had been made, and it became necessary to backtrack to try another vertex assignment. This process would be continued either until each vertex of G was assigned to a vertex of G' and all assignments were verified as consistent, in which case G was concluded isomorphic to G', or until nonisomorphism was established. Two graphs were concluded nonisomorphic, if a level 1 of the tree of vertex choices, there were no more untried vertex pairs. The graph isomorphism algorithm is given as Algorithm 9 (Schmidt and Druffel, 1976) of Appendix G.1.

As an illustration of this backtracking method, the class vectors C = (1,1,1,1,1,1) and C' = (1,1,1,1,1,1) are used to define the initial partition. Since there is just one class in this partition, the vertex assignment 1-1' is first selected. The assignment is checked by composing C with row 1 and column 1 of D and C' with row 1' and column 1' of D'. This composition yields the new class vectors C = (1,2,3,4,2,5) and C' = (1,5,2,4,3,2). The process is illustrated in Table 3.4.5. These class vectors indicate 1-1' is consistent, since the number of vertices of each class of C equals the number of vertices for the corresponding class of C'. At this point, a choice of which vertex pair to use next is made based on choosing a vertex pair from the smallest class. Then, the vertex pair 3-5' is selected. The checking for consistency generates the new class vectors C = (1,2,3,4,2,5) and C' = (1,6,2,7,3,8). Since these class vectors indicate the 3-5' is inconsistent, the algorithm backtracks to

TABLE 3.4.5

CLASS VECTORS GENERATED BY VERTEX ASSIGNMENT 1-1'

Old C	Row 1 of D	Column 1 of D	New C	Old C'	Row 1' of D'	Column 1' of D'	New C'
1	0	0	1	1	0	0	1
1	1	1	2	1	1	2	5
1	2	1	3	1	1	1	2
1	2	2	4	1	2	2	4
1	1	1	2	1	2	1	3
1	1	2	5	1	1	1	2

the last consistent vertex assignment, 1-1', and attempts another assignment for vertex 1. Vertex assignments 1-2', 1-3', 1-4', and 1-5' are either inconsistent or lead to other inconsistent vertex assignments. Finally, the vertex assignment 1-6', after being checked for consistency induces the class vectors C = (1,2,3,4,2,5) and C' = (2,3,4,5,2,1). The next consistent assignment 3-2' induces the class vectors C = (1,2,3,4,2,5) and C' = (2,3,4,5,2,1). Then, the consistent vertex assignment 4-3' induces the class vectors C = (1,2,3,4,5,6) and C' = (2,3,4,6,5,1). Since there are no more arbitrary choices of vertex pairs, the algorithm checks the remaining assignments 2-1', 5-5', and 6-4' for consistency. Since these are all consistent, the graphs are concluded to be isomorphic with the isomorphism γ = (1-6',2-1',3-2',4-3',5-5',6-4').

CHAPTER IV

EVALUATION PROCEDURE FOR DETERMINING THE EFFICIENCY OF THE GRAPH ISOMORPHISM ALGORITHMS

4.1 Introduction

All of the previously described graph isomorphism algorithms which were reviewed in Chapters I and III either employed heuristics or used a necessary and sufficient condition. The heuristics were used to reduce the number of possible invertible functions to be checked for isomorphism either by an enumeration method or by a backtracking technique. A necessary and sufficient condition was used efficiently only for graphs with a small number of vertices. For each of these algorithms, and also for the new graph isomorphism algorithm of Chapter II, the upper computational bound was factorial and the lower computational bound was dependent upon the types of graphs. Since a theoretical evaluation of the graph isomorphism algorithms would consider only the worst (upper bound) and the best (lower bound) case analyses, it would give little insight into the actual computing performances. However, since all the previous graph isomorphism algorithms were implemented by different languages, executed on different computers, and tested by using random graphs, experimental evaluation based on the algorithm's reported execution time performance would be inaccurate.

Thus, in order to make a comparison of the graph isomorphism algorithms, the new graph isomorphism algorithm and the graph

isomorphism algorithms of Berztiss (1973), Ullmann (1976), and Schmidt and Druffel (1976) were chosen for evaluation, since these algorithms represented the four most current graph isomorphism algorithms.

In this Chapter, the procedure which was used to evaluate the four algorithms is presented. The procedure involved using PL/I to implement the graph isomorphism algorithms; using several classes of graphs to test the PL/I implementations; and using the results of the tests to analyze the performance of the algorithms.

4.2 PL/I Implementations

Each of the four algorithms was implemented by PL/I and compiled under the IBM PL/I OS Optimizing Compiler Version 1, Release 2.2. Since the implemented algorithms were evaluated and compared based on their execution time performance, efficiency of the implemented steps using PL/I was more important than the structure of the implemented steps. Thus, the programs were written with the concept of efficiency rather than with the concept of structured programming. The programs were executed on an IBM 370/158 with a 192 K partition size. The size of the necessary partition could be reduced by using the features of PL/I to allocate and free storage; however, execution times would be increased.

The implementation of the new graph isomorphism algorithm involved writing PL/I programs for Algorithm 1 of Section 2.3 which transformed each graph into a MSM and for Algorithm 2 of Section 2.6 which determined all isomorphisms between the two graphs. The PL/I source listings are given in Appendix D. For efficiency of implementation of Equation (2.4.12), graph G was represented by an adjacency list, and

both graphs G and G' were represented by a bit adjacency matrix. Using PL/I, the bit adjacency matrix was processed element by element and row by row.

Berztiss' method was implemented by writing PL/I programs for Algorithm 3 of Appendix E.1 which produced a K-formula for graph G, for Algorithm 4 of Appendix E.1 which created the data structures representing the K-formula, and for Algorithm 5 of Appendix E.1 which determined all isomorphisms between graphs G and G'. The PL/I source listings are given in Appendix E.2. Algorithm 5, the backtracking algorithm, was programmed to terminate after one isomorphism was found or nonisomorphism was established. Since the performance of the backtracking algorithm was very dependent on the structure of the K-formula, Algorithm 3 was programmed to produce a K-formula with the adjacency structural information contained in the beginning of the formula. Thus, for graph 7 of Figure 2.3.1.1, the K-formula

121***31464***5123***63***42565

would be produced instead of the K-formula

121***31***42***5123***63456456

which was used for illustration in Section 3.2.

The implementation of Ullmann's algorithm involved writing PL/I programs for the calculation of the initial matrix M, which is defined in Appendix F.1 and for Algorithm 6 of Appendix F.1 which determined if the two graphs were isomorphic. The PL/I source listings are given in Appendix F.2. Ullmann's isomorphism algorithm was modified and programmed to terminate when the matrix M, after refinement, contained exactly one 1 in each row and each column. The refinement conditions (3.3.1a) and (3.3.1b), which were used by Algorithm 6, were programmed

as an internal procedure. In order to efficiently execute the refinement conditions, each row of M, and each row and column of the adjacency matrix of G' were stored in separate computer words. Thus, (3.3.1a) and (3.3.1b) were implemented by "ORing" word by word the appropriate row of M with the appropriate row or column of G'. This proved to be much faster than the corresponding element by element computation.

Schmidt and Druffel's algorithm was implemented by writing PL/I programs for Algorithm 7 (Floyd, 1962) of Appendix G.1, which constructed the distance matrices, for Algorithm 8 of Appendix G.1, which generated the initial partition, and for Algorithm 9 of Appendix G.1, which determined if an isomorphism existed between two graphs. The PL/I source listings are given in Appendix G.2. Step 6 of Algorithm 9 was programmed as an internal procedure which chose from the smallest class a previously unassigned vertex G. This strategy of choosing a vertex from the smallest class could refine the partition by reducing the size of its larger class and hence, could reduce the searching required at the next level of the vertex assignment tree. Thus, the overall effect of using this strategy was that usually the breadth of search was reduced while the depth of search was increased.

4.3 Input Data

The programs were then executed by using several classes of regular graphs. Regular graphs were chosen since all previous graph isomorphism algorithms encountered difficulty in processing regular graphs. The difficulty was due to the fact that all previous graph isomorphism algorithms employed heuristics which could not distinguish

between vertices having the same characteristics. In the case of regular graphs, all vertices had the same degree characteristic, i.e., all vertices of a k-regular graph had the same indegree and outdegree k. Thus, a graph isomorphism algorithm which used a heuristic based on the degrees of the vertices could not effectively process k-regular graphs.

The PL/I program "GRAPHS" of Appendix B was used in randomly generating both nonisomorphic and isomorphic regular graphs. Twenty-five pairs of random n/2 regular nonisomorphic graphs having n vertices, for n = 6, 8, 10, 12, 20, 30 were generated. Each pair of nonisomorphic graphs was produced by first, randomly constructing an n/2 regular graph, and then, by randomly permuting its rows, a second n/2 regular nonisomorphic graph was produced.

Twenty-five pairs of random n/2 simple regular graphs having n vertices for n = 6, 8, 10, 12, 16, 20, 26, 30, 40 were generated. Each pair of isomorphic graphs was constructed by first, randomly producing an n/2 simple regular graph, and then, by randomly permuting its rows and columns, a second n/2 simple regular isomorphic graph was produced.

Several published sets of strongly regular nonisomorphic graphs were also tested. The set of strongly regular graphs produced by Paulus (1973) was used. This set contained fifteen nonisomorphic graphs of order 25 with the indegree and the outdegree of each vertex equal to 12 ($d(v_i) = 12$, for all i). From these fifteen graphs, fifteen random pairs were chosen. A similar collection of strongly regular nonisomorphic graphs of orders 35 and 36 were developed by Bussemaker and Seidel (1970). From this collection, fifteen random

pairs of the first 35 (numbered 1-35) of the 80 Steiner graphs of order 35 with $d(v_i) = 19$, for all i, and five random pairs of the 12 (numbered 81-92) Latin Square graphs of order 36 with $d(v_i) = 16$, for all i, were used.

4.4 Analysis Procedure

Since each of the four methods required some initial preparation of the graph representation used by the isomorphism algorithm, each method was considered to contain two primary algorithms: the graph representation algorithm and the graph isomorphism algorithm. The classification for the algorithms of each method is shown in Table 4.4.1.

Each of the four methods was tested by using the collection of graphs which were described in the previous section. The execution time results of each procedure were obtained by using a Basic Assembly Language routine "ASMTME". ASMTME, given in Appendix C, accumulated in units of 26.04166 microseconds the CPU time which was used by a specified section of a procedure. This unit of time was directly dependent upon the power supply used by the IBM 370/158 and thus, may be different for other computers.

In evaluating any algorithm, the performance of its implementation should be distinguished from the performance of the algorithm itself. The performance of the implementation is both language and computer dependent, whereas the performance of the algorithm is data dependent.

The performance of the implementation of the algorithms of each graph isomorphism method was obtained by measuring the execution time required by the algorithm to process a pair of graphs. As an aid to

TABLE 4.4.1

CLASSIFICATION OF THE ALGORITHMS USED BY THE GRAPH ISOMORPHISM METHODS

Methods	Graph Representation		Graph Isomorphism	
	Algorithm No.	Reference	Algorithm No.	Reference
New	1	Sect. 2.3	2	Sect. 2.6
Berztiss	3	App. E.1.1	5	App. E.1.3.
	4	App. E.1.2		
Ullmann	6 (Step 0)	App. F.1	6 (Steps 1-7)	App. F.1
Schmidt and Druffel	7	App. G.1.1	8	App. G.1.2
			9	App. G.1.3

evaluating this performance, the execution times, for the set of nonisomorphic and isomorphic regular graphs were experimentally fitted by using the Statistical Analysis System (SAS) developed at the North Carolina State University.

The raw data, up to 25 observations for each value of the number n of vertices tested, were input to the linear regression procedure of SAS. Next, linear and quadratic polynomial curve fits of the raw data were tried. If the coefficient of determination R^2 of either of these curves was the calculated maximum, i.e., the maximum percentage of variation of execution time which could be explained by a polynomial curve fit, then no higher degree polynomial was tried. Otherwise, the polynomial fitting would continue by increasing the degree of the polynomial until the maximum R^2 was achieved. If the R^2 of any polynomial curve was above 90%, then the data were assumed to be polynomial in behavior, and the polynomial curve with the highest R^2 was chosen. However, if the R^2's of the polynomial curves were less than 90%, then linear and quadratic exponential fits were tried. As above, the exponential fitting was terminated at the degree which achieved the maximum R^2. The curve, either polynomial or exponential, with the highest R^2 was then chosen to fit the raw data.

For the nonisomorphic regular graphs, the performance of the implementation was measured only for the graph isomorphism algorithms. However, for the isomorphic regular graphs, the performance of the implementation was measured for both the graph representation algorithms and graph isomorphism algorithms.

On the other hand, the performance of the algorithm was measured only for the graph isomorphism algorithm using all classes of graphs

tested. The performance of the new graph isomorphism algorithm was measured by the number of isomorphisms between the two MSM's, each of which was checked by (2.4.12), and by the number of vertex assignments required to establish either isomorphism or nonisomorphism. Both of these measurements reflected the power of the output function J(s) used in defining the MSM's. The performance of the three backtracking algorithms was measured by the number of times the algorithm had to backtrack in processing a pair of graphs and by the number of vertex assignments required to established either isomorphism or nonisomorphism. These measurements reflected the power of both the heuristics and the backtracking technique used by each algorithm.

CHAPTER V

EXPERIMENTAL RESULTS AND CONCLUSIONS

5.1 New Algorithms

The new graph isomorphism algorithm was tested using the set of nonisomorphic regular graphs which were described in Section 4.3. Then, the graph representation algorithm and the graph isomorphism algorithm (see Table 4.4.1) were applied to the set of isomorphic regular graphs (see Section 4.3). The performance of the implementation for each algorithm using these regular graphs is summarized in Table 5.1.1. The performance of the graph isomorphism algorithm is summarized in Table 5.1.2. Next, the sets of strongly regular graphs (see Section 4.3) were used. However, since the isomorphism algorithm failed to complete the determination of nonisomorphism for one pair of each type in 4 minutes, the testing was terminated.

5.1.1 Performance of the Implementation

Using the procedure which was described in Section 4.4, the raw execution times, t, summarized in Table 5.1.1 were fitted by linear and quadratic polynomial functions of the number of vertices, n. It is noted that all execution times are given in milliseconds. Since for the nonisomorphic regular graphs the raw execution times generated by the isomorphism algorithm were centered around a constant line, the fit was the following equation independent of the number of vertices n

$$t = 1.89.$$

TABLE 5.1.1

PERFORMANCE OF IMPLEMENTATION FOR NEW ALGORITHMS USING REGULAR GRAPHS
(Time in milliseconds)

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs			
		Graph Iso. Alg.		Graph Iso. Alg.		Graph Rep. Alg.	
		Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
6	25*	2.07	1.04	18.78	0.47	684.41	55.36
8	25	1.86	0.06	27.46	3.88	1004.32	163.36
10	25	1.84	0.06	40.08	10.83	1292.83	224.27
12	25	1.86	0.07	50.87	12.66	1658.87	105.65
16	25	...**	...	71.21	6.51	2727.94	75.08
20	25	1.86	0.06	99.81	34.16	4070.14	101.04
26	25	...	...	145.52	3.51	6903.64	54.34
30	25	1.89	0.10	175.50	5.70	8781.60	141.82
40	25	...	...	292.00	8.32	15594.50	150.13

* Only 16 observations were used for isomorphic graphs with 6 vertices.

** Nonisomorphic graphs were not generated for missing entries.

TABLE 5.1.2

PERFORMANCE OF NEW GRAPH ISOMORPHISM ALGORITHM USING REGULAR GRAPHS

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs	
		Mean No. MSM's Iso.	Mean No. Vertex Assign.	Mean No. MSM's Iso.	Mean No. Vertex Assign.
6	25*	0	0	1.00	6.00
8	25	0	0	2.92	23.36
10	25	0	0	7.48	70.48
12	25	0	0	7.88	94.56
16	25	...**	...	5.08	81.28
20	25	0	0	6.76	135.20
26	25	...	...	1.52	39.52
30	25	0	0	1.88	56.40
40	25	...	...	1.60	64.00

* Only 16 observations used for isomorphic graphs with 6 vertices.

** Nonisomorphic graphs were not generated for missing entries.

This equation with raw execution times is plotted in Figure 5.1.1.1. The constant 1.89 was calculated by averaging the averages of the execution times for each n. Using the isomorphic regular graphs, the raw execution times generated by the isomorphism algorithm were fitted by the quadratic equation

$$t = 0.123n^2 + 2.216n + 3.891, \quad R^2 = 97.40\%$$

where the calculated maximum R^2 for a polynomial fit was 97.50%. This equation with the raw execution times is plotted in Figure 5.1.1.2. Based on the same isomorphic graphs, the raw execution times generated by the graph representation algorithm were fitted by the quadratic equation

$$t = 9.705n^2 - 8.523n + 403.195, \quad R^2 = 99.90\%$$

where the calculated maximum R^2 for a polynomial fit was 99.92%. This equation with the raw execution times is plotted in Figure 5.1.1.3.

Based on these equations, the performance of the implementation of the new algorithms was of an experimental order $O(n^2)$. Thus, based on the implementation results, the algorithms performed efficiently for the class of regular graphs tested.

5.1.2 Performance of the Graph Isomorphism Algorithm

The isomorphism algorithm determined nonisomorphism for the non-isomorphic regular graphs based on their MSM representations. Either their MSM's had a different number of states or a different set of outputs. Thus, the number of MSM's isomorphism and the number of vertex assignments of Table 5.1.2 are all zero. This indicated that the output function J(s) was very effective in distinguishing non-isomorphic graphs. In order to evaluate the performance of the

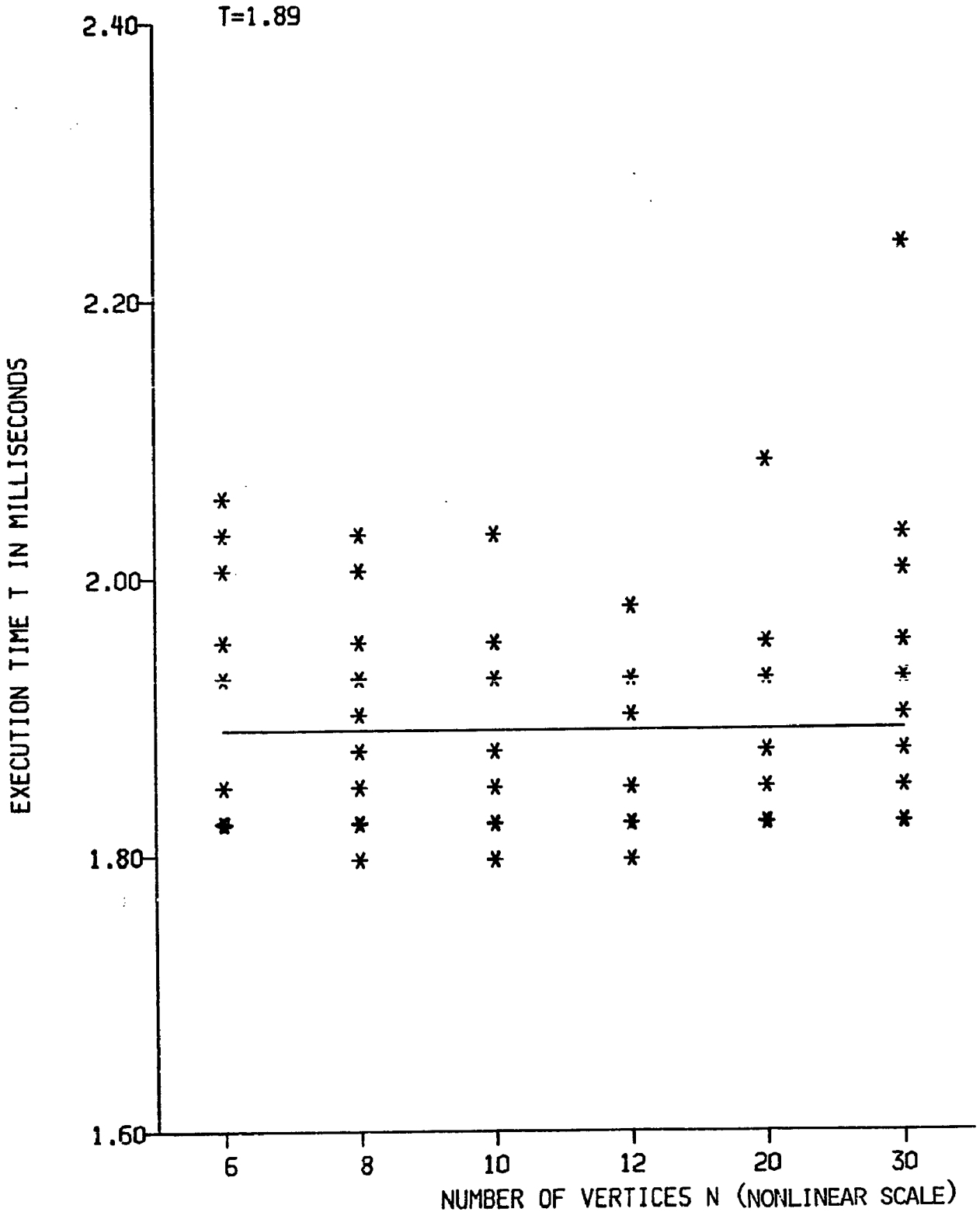


Figure 5.1.1.1 Plot of New Graph Isomorphism Algorithm Using Nonisomorphic Regular Graphs.

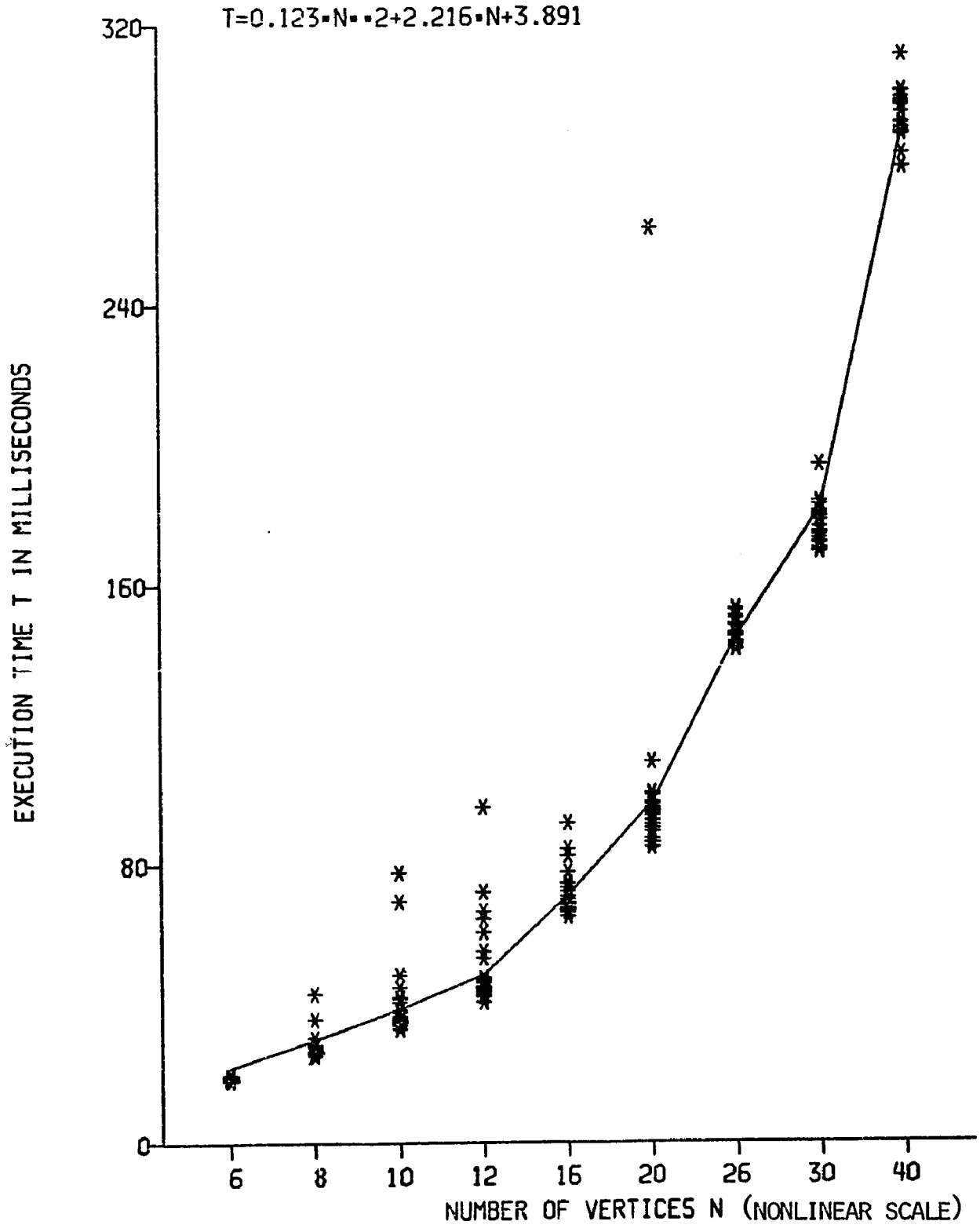


Figure 5.1.1.2 Plot of New Graph Isomorphism Algorithm Using Isomorphic Regular Graphs.

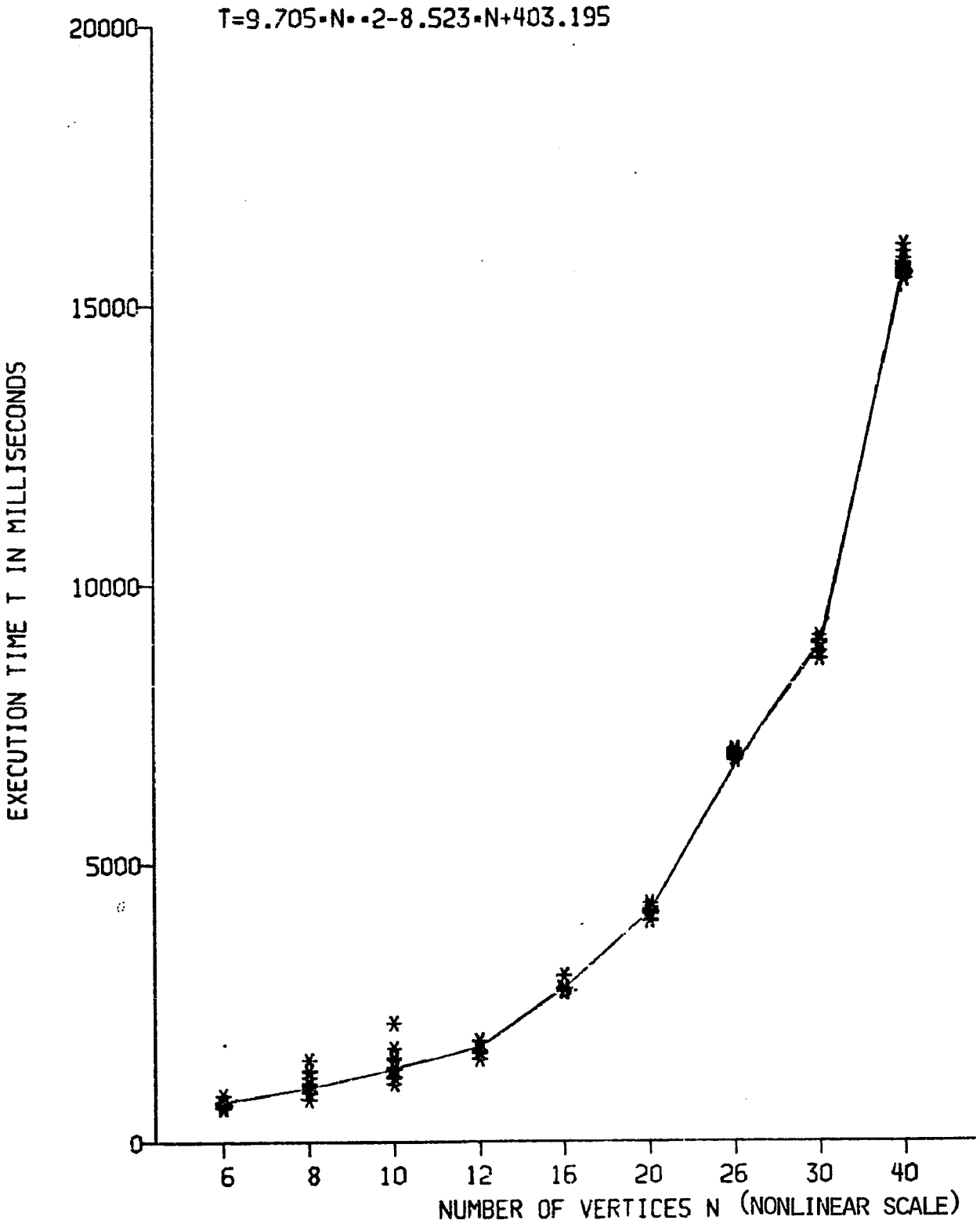


Figure 5.1.1.3 Plot of Graph Representation Algorithm (New) Using Isomorphic Regular Graphs.

isomorphism algorithm for the isomorphic regular graphs, only those isomorphic graphs having one isomorphism were used. Thus, only 16 observations were used for graphs of 6 vertices since the other 9 contained more than one isomorphism. Based on the data of Table 5.1.2, the algorithm was effective in determining isomorphisms for graphs of low and high orders, and reasonably effective for graphs between the two extremes.

5.2 Berztiss' Algorithms

The performance of the implementation for each of Berztiss' algorithms (see Table 4.4.1) using regular graphs is summarized in Table 5.2.1. The performance of the graph isomorphism algorithm is summarized in Table 5.2.2. Results for either nonisomorphic or isomorphic regular graphs having 20 or more vertices were not obtained, since the isomorphism algorithm was unable to process one pair of these graphs in 4 minutes execution time on the IBM 370/158. Also, for the same reason, no results were obtained for any of the nonisomorphic strongly regular graphs.

5.2.1 Performance of the Implementation

For the isomorphism algorithm, the raw execution times, t, summarized in Table 5.2.1 were fitted by linear and quadratic polynomials and by linear and quadratic exponential functions of n. For the nonisomorphic regular graphs, the calculated maximum R^2 for a polynomial fit was 30.22%, and the calculated maximum R^2 for an exponential fit was 83.20%. Thus, the following second order exponential equation was chosen:

$t = \exp(0.025n^2 + 0.236n + 3.946) \times 26.04166 \times 10^{-3}$, $R^2 = 83.19\%$.

TABLE 5.2.1

PERFORMANCE OF IMPLEMENTATION FOR BERZTISS' ALGORITHMS USING REGULAR GRAPHS
(Time in milliseconds)

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs			
		Graph Iso. Alg.		Graph Iso. Alg.		Graph Rep. Alg.	
		Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
6	25	13.95	3.01	16.59	4.65	15.49	0.59
8	25	47.80	24.34	50.88	28.90	20.71	0.86
10	25	265.68	262.29	208.62	196.07	27.33	1.50
12	25	1301.93	1609.10	1284.67	1706.99	34.49	1.50
16	25	...*	...	81629.48	167142.58	54.28	2.34

* Nonisomorphic graphs were not generated for n = 16.

TABLE 5.2.2

PERFORMANCE OF BERZTISS' GRAPH ISOMORPHISM ALGORITHM USING REGULAR GRAPHS

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs	
		Mean No. Backtracks	Mean No. Vertex Assign.	Mean No. Backtracks	Mean No. Vertex Assign.
6	25	54.00	54.00	30.00	36.00
8	25	248.96	248.96	216.04	224.04
10	25	1414.96	1414.96	1013.76	1023.76
12	25	6646.44	6646.44	6305.56	6317.56
16	25	...*	...	340094.80	340110.80

* Nonisomorphic graphs were not generated for n = 16.

This equation with the raw execution times is plotted in Figure 5.2.1.1. For the isomorphic regular graphs, the calculated maximum R^2 for a polynomial fit was 16.45%, and the calculated maximum R^2 for an exponential fit was 85.17%. Thus, the following second order exponential equation was chosen:

$t = \exp(0.0291n^2 + 0.0829n + 4.8736) \times 26.04166 \times 10^{-3}$, $R^2 = 85.13\%$.

This equation with the raw execution times is plotted in Figure 5.2.1.2. Based on the same isomorphic graphs, the raw execution times generated by the graph representation algorithm were fitted by the quadratic equation

$$t = 0.165n^2 + 0.229n + 8.255, \quad R^2 = 98.83\%$$

where the calculated maximum R^2 for a polynomial fit was 98.85%. This equation with the raw execution times is plotted in Figure 5.2.1.3.

Based on these equations, the performance of the implementation of Berztiss' graph isomorphism algorithm was of an experimental order $O(\exp(n^2))$. Similarly, the performance of the implementation of the graph representation algorithm was of the experimental order $O(n^2)$. Thus, although the graph representation algorithm performed efficiently for the regular graphs tested, the isomorphism algorithm displayed exponential growth in processing time as the number of vertices increased and thus, could not be considered efficient.

5.2.2 Performance of the Graph Isomorphism Algorithm

In evaluating the performance of the isomorphism algorithm, the number of backtracks and the number of vertex assignments of Table 5.2.2 were considered. Based on these data, the isomorphism algorithm was ineffective in establishing either nonisomorphism or isomorphism.

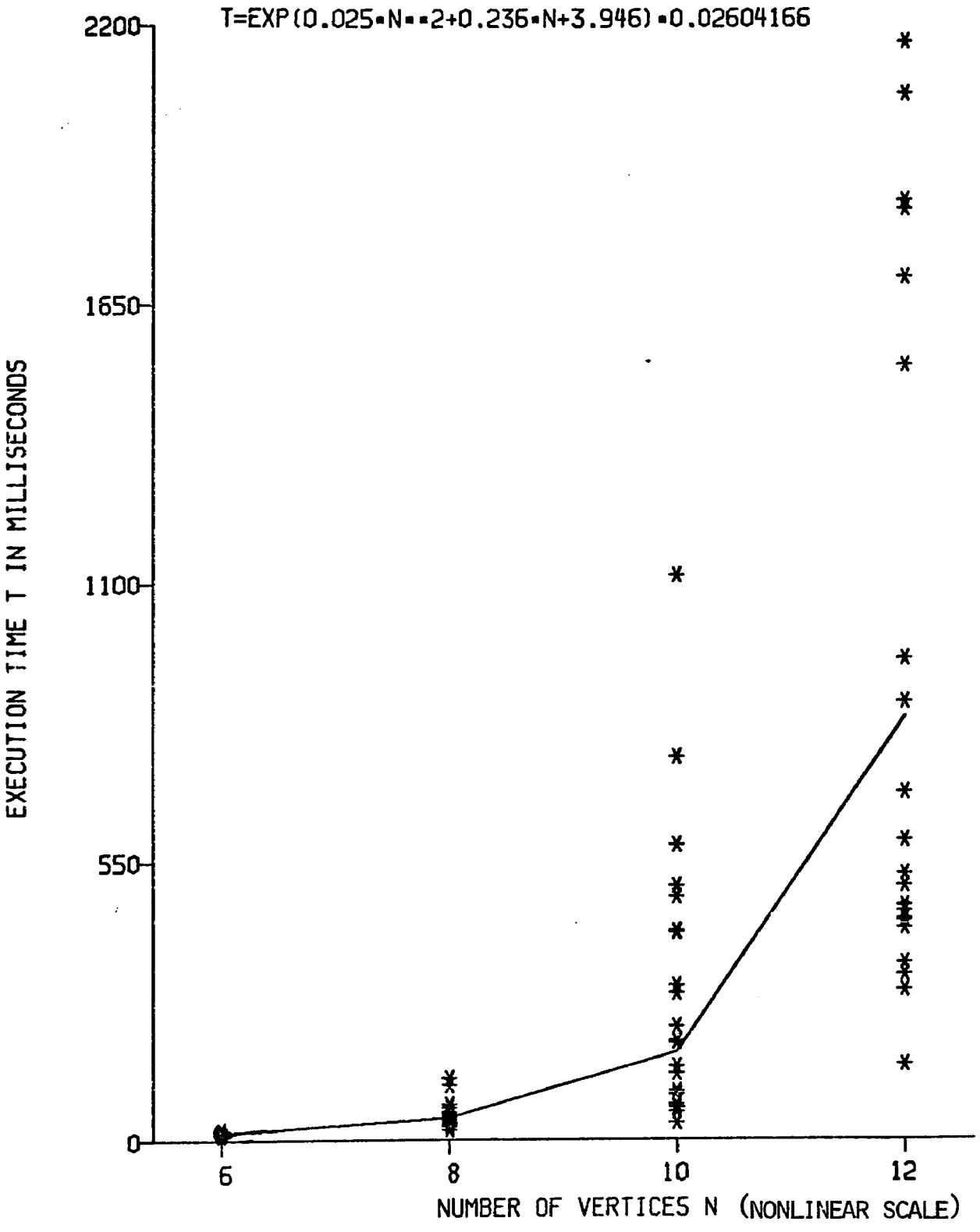


Figure 5.2.1.1 Plot of Berztiss' Isomorphism Algorithm Using Nonisomorphic Regular Graphs.

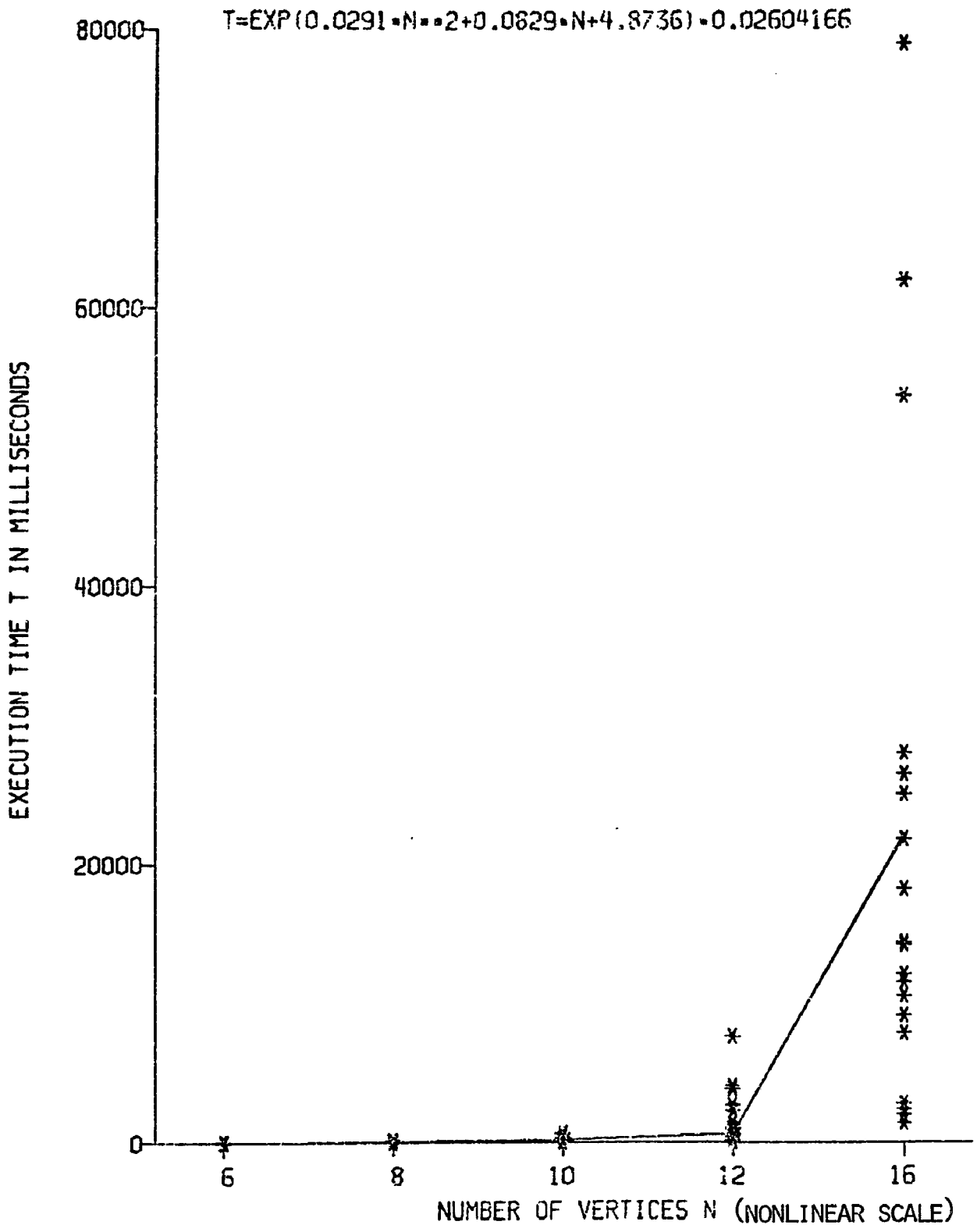


Figure 5.2.1.2 Plot of Berztiss' Isomorphism Algorithm Using Isomorphic Regular Graphs.

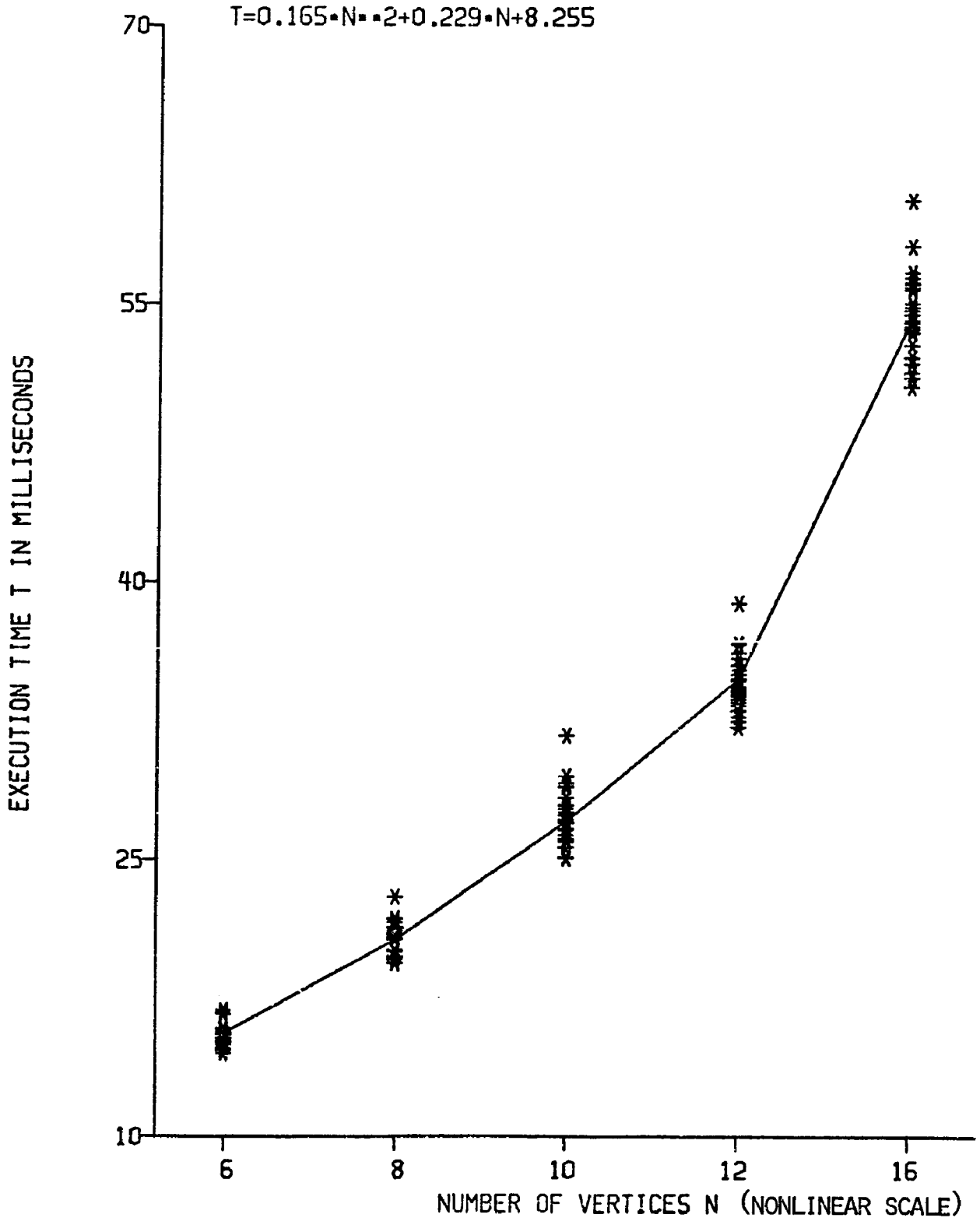


Figure 5.2.1.3 Plot of Graph Representation Algorithm (Berztiss) Using Isomorphic Regular Graphs.

The number of backtracks and vertex assignments grew exponentially as the number of vertices increased. Thus, it could be concluded that the K-formula representation of a graph was not a good heuristic for the graph isomorphism problem, and that the backtracking technique of Berztiss was ineffective.

5.3 Ullmann's Algorithms

The performance of the implementation for each of Ullmann's algorithms (see Table 4.4.1) using regular graphs is summarized in Table 5.3.1. The performance of the graph isomorphism algorithm is summarized in Table 5.3.2. Only 10 observations of isomorphic regular graphs were made, due to the amount of execution time required by each pair of graphs. Results for nonisomorphic graphs having 20 or more vertices and for isomorphic graphs having 26 or more vertices were not obtained, since the isomorphism algorithm was unable to process one pair of these graphs in 4 minutes. Also, for the same reason, no results were obtained for any of the nonisomorphic regular graphs.

5.3.1 Performance of the Implementation

For the isomorphism algorithm, the raw execution times, t, summarized in Table 5.3.1 were fitted by linear and quadratic polynomials and by linear and quadratic exponential functions of n. For the non-isomorphic regular graphs, the calculated maximum R^2 for a polynomial fit was 69.94%, and the calculated maximum R^2 for an exponential fit was 81.37%. Thus, the following second order exponential equation was chosen:

$t = \exp(0.0096n^2 + 0.241n + 10.880) \times 26.04166 \times 10^{-3}$, $R^2 = 81.36\%$.

This equation with the raw execution times is plotted in Figure 5.3.1.1.

TABLE 5.3.1

PERFORMANCE OF IMPLEMENTATION FOR ULLMANN'S ALGORITHMS USING REGULAR GRAPHS
(Time in milliseconds)

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs			
		Graph Iso. Alg.		Graph Iso. Alg.		Graph Rep. Alg.	
		Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
6	25	9034.26	3932.18	1558.80	902.67	426.55	3.38
8	25	19056.53	7329.52	4214.03	2815.34	453.75	10.23
10	25	45682.46	23330.16	8653.00	6646.77	455.78	10.97
12	25	108315.46	45408.05	26003.25	22338.78	458.28	9.16
16	25	...*	...	61541.97	45641.89	484.76	8.76
20	10	...**	...	192062.50	124310.96	515.99	8.34

* Nonisomorphic graphs were not generated for n = 16.

** Missing entries for n = 20 due to excessive time required.

TABLE 5.3.2

PERFORMANCE OF ULLMANN'S GRAPH ISOMORPHISM ALGORITHM USING REGULAR GRAPHS

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs	
		Mean No. Backtracks	Mean No. Vertex Assign.	Mean No. Backtracks	Mean No. Vertex Assign.
6	25	10.12	12.88	0.08	2.96
8	25	18.32	26.44	0.52	6.84
10	25	40.68	60.12	1.64	12.76
12	25	89.08	128.44	4.84	37.96
16	25	...*	...	8.60	77.40
20	10	...**	...	20.00	226.60

* Nonisomorphic graphs were not generated for n = 16.

** Missing entries for n = 20 due to excessive time required.

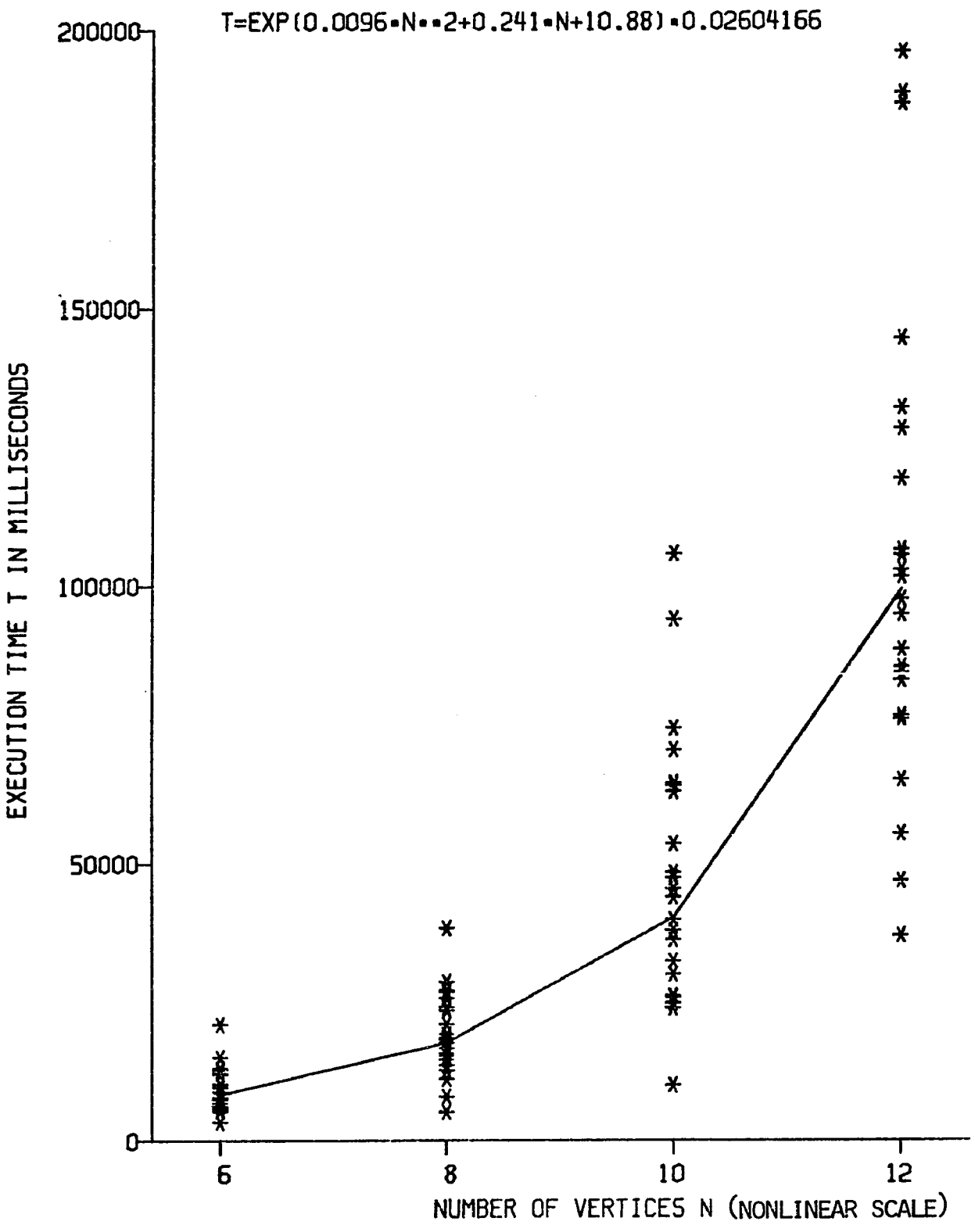


Figure 5.3.1.1 Plot of Ullmann's Isomorphism Algorithm Using Nonisomorphic Regular Graphs.

For the isomorphic regular graphs, the calculated R^2 for a polynomial fit was 62.33%, and the calculated R^2 for an exponential fit was 73.59%. Thus, the following second order exponential equation was chosen:
$t = \exp(-0.0084n^2 + 0.5470n + 7.8639) \times 26.04166 \times 10^{-3}$, $R^2 = 72.91\%$. This equation with the raw execution times is plotted in Figure 5.3.1.2. Based on the same isomorphic graphs, the raw execution times generated by the graph representation algorithm were fitted by the quadratic equation

$$t = 0.066n^2 + 3.868n + 407.988, \quad R^2 = 82.13\%$$

where the calculated maximum R^2 for a polynomial fit was 88.23%. This equation with the raw execution times is plotted in Figure 5.3.1.3.

Based on these equations, the performance of the implementation of Ullmann's graph isomorphism algorithm was of experimental order $O(\exp(n^2))$. Similarly, the performance of the graph representation algorithm was of the experimental order $O(n^2)$. Thus, although the graph representation algorithm performed efficiently for the regular graphs tested, the isomorphism algorithm displayed exponential growth in processing as the number of vertices increased and thus, could not be considered efficient.

5.3.2 Performance of the Graph Isomorphism Algorithm

Based on the number of backtracks and on the number of vertex assignments of Table 5.3.2, the isomorphism algorithm was more effective in establishing isomorphism than it was in establishing nonisomorphism. Because Ullmann's isomorphism algorithm was modified to terminate when the matrix M, after refinement, contained exactly one 1 in each row and each column, the number of mean vertex assignments was sometimes less

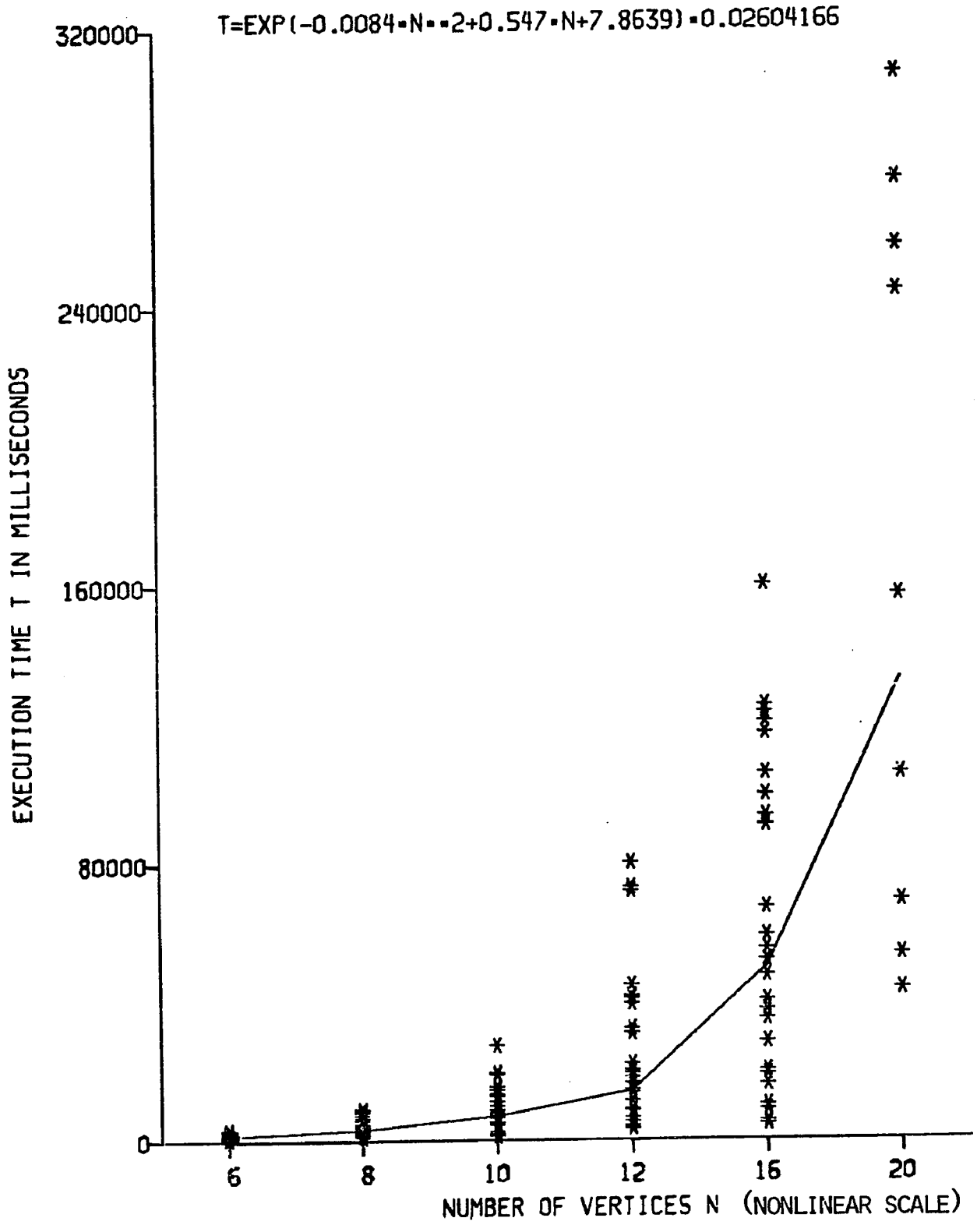


Figure 5.3.1.2 Plot of Ullmann's Isomorphism Algorithm Using Isomorphic Regular Graphs.

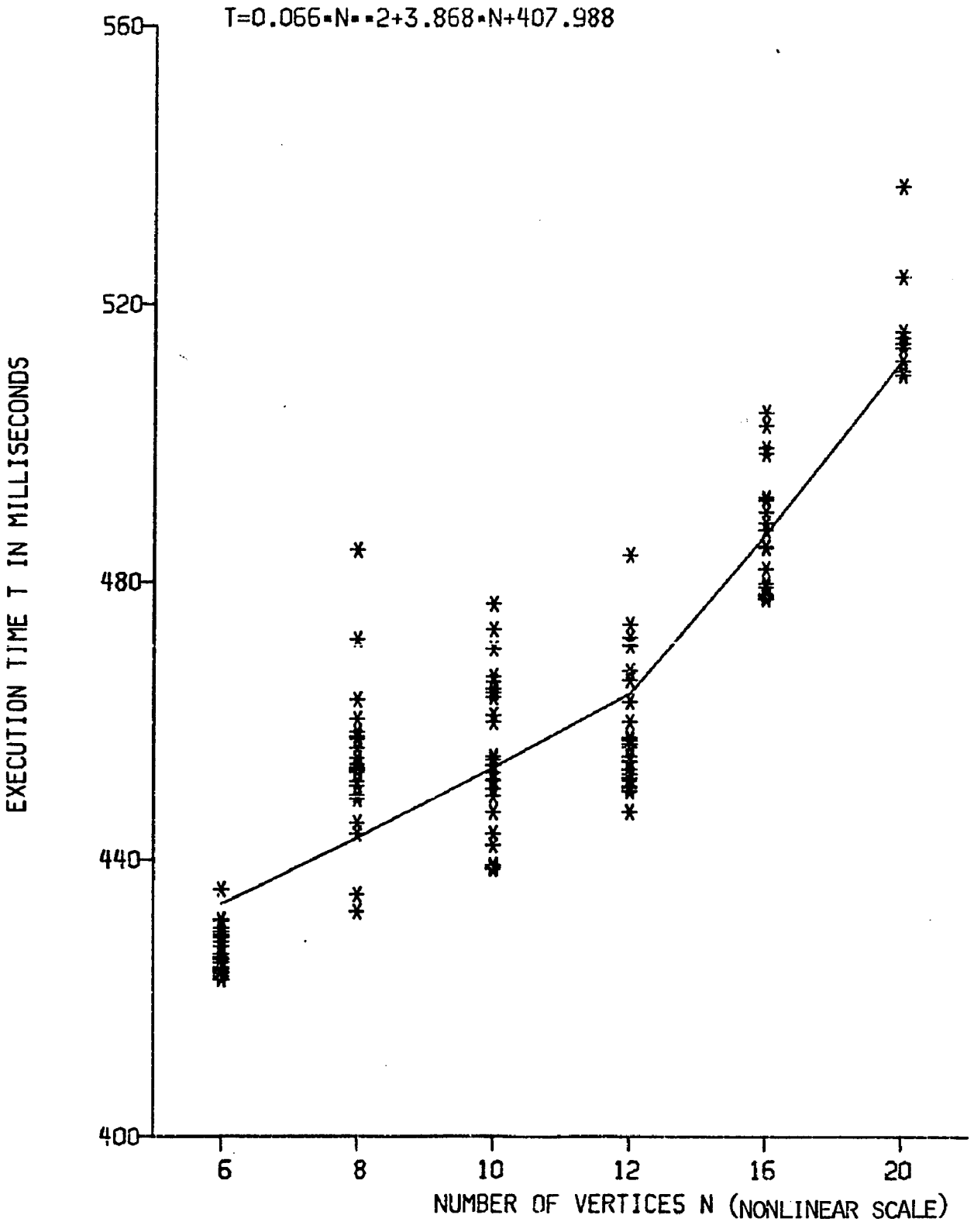


Figure 5.3.1.3 Plot of Graph Representation Algorithm (Ullmann) Using Isomorphic Regular Graphs.

than the order of the graphs being tested. However, as the number of vertices increased, both the number of backtracks and the number of vertex assignments increased exponentially. Thus, it could be concluded that the refinement conditions of (3.3.1a) and (3.3.1b) performed as an effective heuristic for graphs of 16 or less vertices but was not effective for the general graph isomorphism problem, and that the backtracking technique of Ullmann also became ineffective as the number of vertices increased.

5.4 Schmidt and Druffel's Algorithms

The performance of the implementation for each of Schmidt and Druffel's algorithms (see Table 4.4.1) using regular graphs is summarized in Table 5.4.1. The performance of the graph isomorphism algorithm is summarized in Table 5.4.2. For the three sets of strongly regular graphs, the overall performance for the algorithm is given in Tables 5.4.3, 5.4.4, and 5.4.5. Because of the large amount of time required to process each pair of Latin Square graphs, only five pairs of these graphs were tested.

5.4.1 Performance of the Implementation

For the isomorphism algorithm, the raw execution times, t, summarized in Table 5.4.1 were fitted by a linear and quadratic polynomial. For the nonisomorphic regular graphs, the raw execution times were fitted by the quadratic equation

$$t = 0.056n^2 + 0.117n + 5.593, \quad R^2 = 99.56\%$$

where the calculated maximum R^2 for a polynomial fit was 99.57%. This equation with the raw execution times is plotted in Figure 5.4.1.1.

TABLE 5.4.1

PERFORMANCE OF IMPLEMENTATION FOR SCHMIDT AND DRUFFEL'S ALGORITHMS USING REGULAR GRAPHS
(Time in milliseconds)

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs			
		Graph Iso. Alg.		Graph Iso. Alg.		Graph Rep. Alg.	
		Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
6	25	8.17	0.15	42.90	3.67	44.59	1.12
8	25	10.15	0.35	61.60	8.21	64.04	2.22
10	25	12.54	0.45	97.80	12.46	99.78	2.43
12	25	15.29	0.47	138.60	15.38	143.80	4.60
16	25	...*	...	250.90	28.52	262.23	5.57
20	25	30.32	0.85	417.00	38.82	440.74	8.72
26	25	...	...	816.82	82.14	832.63	16.20
30	25	60.00	2.76	1151.42	89.98	1169.93	33.65
40	25	...	...	2468.92	314.22	2432.21	50.63

* Nonisomorphic graphs were not generated for n = 16, 26, 40.

TABLE 5.4.2

PERFORMANCE OF SCHMIDT AND DRUFFEL'S GRAPH ISOMORPHISM ALGORITHM USING REGULAR GRAPHS

No. of Vertices	No. of Tests	Nonisomorphic Graphs		Isomorphic Graphs	
		Mean No. Backtracks	Mean No. Vertex Assign.	Mean No. Backtracks	Mean No. Vertex Assign.
6	25	0	0	0.88	8.64
8	25	0	0	1.80	13.68
10	25	0	0	2.08	18.88
12	25	0	0	2.28	23.60
16	25	...*	...	2.32	31.48
20	25	0	0	2.00	38.80
26	25	...	...	2.92	56.84
30	25	0	0	2.68	65.68
40	25	...	...	3.96	94.36

* Nonisomorphic graphs were not generated for n = 16, 26, 40.

TABLE 5.4.3

PERFORMANCE RESULTS FOR SCHMIDT AND DRUFFEL'S ALGORITHMS USING STRONGLY REGULAR NONISOMORPHIC GRAPHS OF ORDER 25
(Time in milliseconds)

Graph Pairs G-G'	Backtracks	Vertex Assign.	Total Run Time
4-12	1880	4788	29989.26
13-9	1541	4767	27589.75
1-2	894	3161	16219.87
13-15	1587	4837	28078.14
7-10	1421	4291	23858.61
15-11	1635	5473	31549.71
13-9	1541	4767	27436.63
3-7	423	2113	8519.66
10-1	327	1825	7013.18
7-3	1446	4360	24197.42
5-4	2325	6259	44012.20
10-4	1155	3877	21029.08
10-7	423	2113	8501.87
11-9	1541	4768	28214.91
14-11	1713	4976	30602.02

TABLE 5.4.4

PERFORMANCE RESULTS FOR SCHMIDT AND DRUFFEL'S ALGORITHMS USING NONISOMORPHIC STEINER GRAPHS OR ORDER 35
(Time in milliseconds)

Graph Pairs G-G'	Backtracks	Vertex Assign.	Total Run Time
9-26	2009	12243	64483.03
30-22	1737	10795	61987.48
1-5	5829	13651	149035.24
30-14	2289	14107	85157.19
19-24	1909	7751	50776.68
33-25	2441	5814	49426.45
26-22	2733	12811	83181.97
9-15	2237	13343	70138.55
24-1	597	3955	19358.38
19-5	2997	10579	78309.25
14-9	2747	9153	77575.92
24-9	2749	12819	85840.68
24-15	2581	12171	82169.28
25-22	3129	12379	91342.29
33-25	2441	5814	50127.23

TABLE 5.4.5

PERFORMANCE RESULTS FOR SCHMIDT AND DRUFFEL'S ALGORITHMS USING NONISOMORPHIC LATIN SQUARE GRAPHS OF ORDER 36
(Time in milliseconds)

Graph Pairs G-G'	Backtracks	Vertex Assign.	Total Run Time
83-90	3997	20196	115413.43
91-87	3493	21636	122977.57
81-82	10573	24252	362458.44
91-84	3989	22132	142182.88
86-88	8677	26388	261802.92

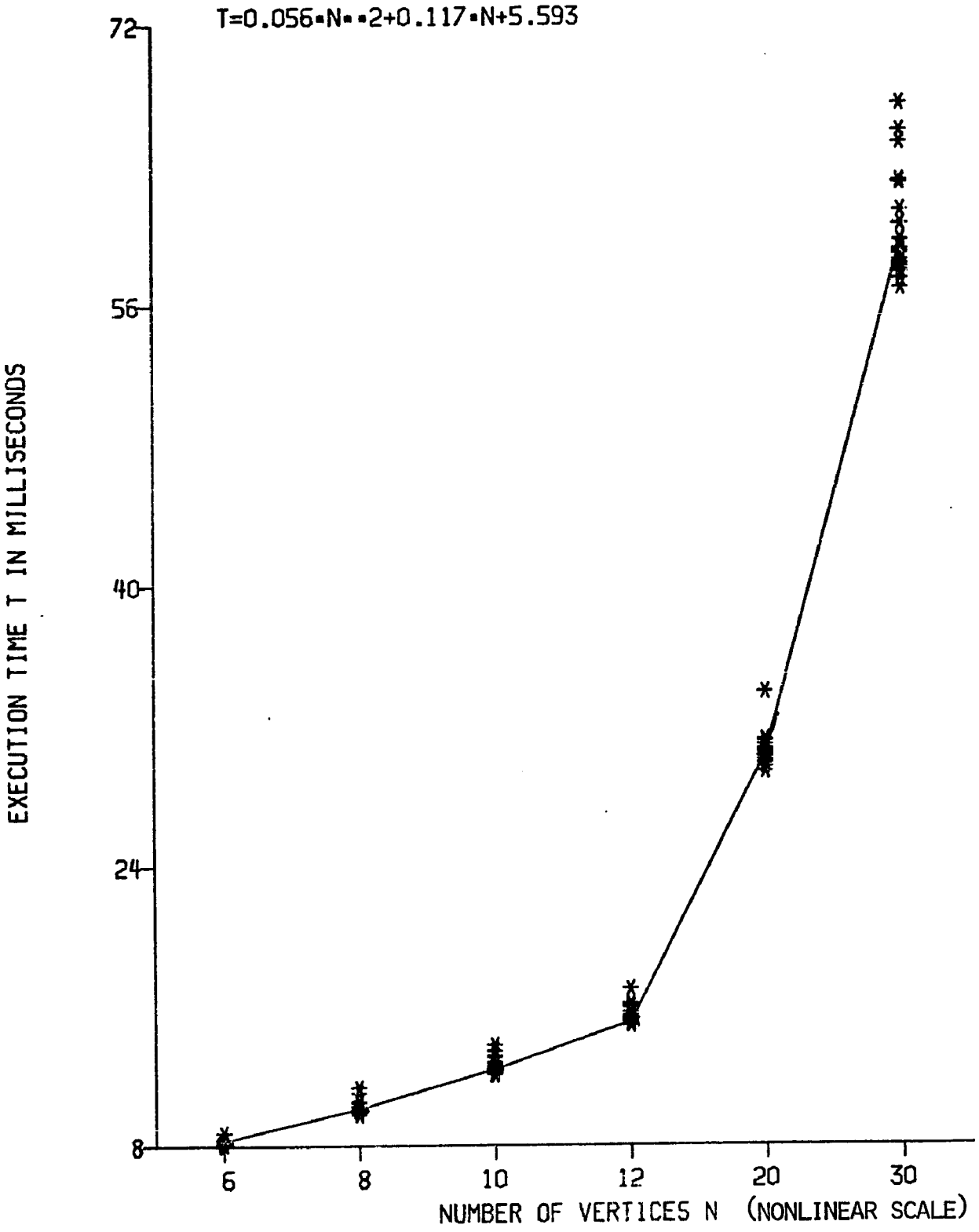


Figure 5.4.1.1 Plot of Schmidt and Druffel's Isomorphism Algorithm Using Nonisomorphic Regular Graphs.

For the isomorphic regular graphs, the raw execution times were fitted by the quadratic equation

$$t = 2.294n^2 - 35.856n + 210.124, \quad R^2 = 97.72\%$$

where the calculated maximum R^2 for a polynomial fit was 97.84%. This equation with the raw execution times is plotted in Figure 5.4.1.2. The execution times used by the graph representation algorithm in processing isomorphic regular graphs were fitted by the quadratic equation

$$t = 2.159n^2 - 30.402 + 175.786, \quad R^2 = 99.85\%$$

where the calculated maximum R^2 for a polynomial fit was 99.85%. This equation with the raw execution times is plotted in Figure 5.4.1.3.

Based on these equations, the performance of the implementation of each of Schmidt and Druffel's algorithms is of experimental order $O(n^2)$. Thus, for these classes of regular graphs, the algorithms of Schmidt and Druffel performed efficiently.

For the strongly regular graphs, the total average processing times, summarized in Tables 5.4.3, 5.4.4, and 5.4.5 were: 23.79 seconds for the graphs of order 25; 73.26 seconds for the Steiner graphs; and 200.97 seconds for the Latin Square graphs. Total processing time was calculated by taking the sum of the execution times for the graph isomorphism algorithm and the graph representation algorithm. While it could not be concluded that the algorithms processed these graphs efficiently, it was concluded that since the graphs were processed, the graph isomorphism algorithm was effective.

5.4.2 Performance of the Graph Isomorphism Algorithm

Based on the number of backtracks and the number of vertex

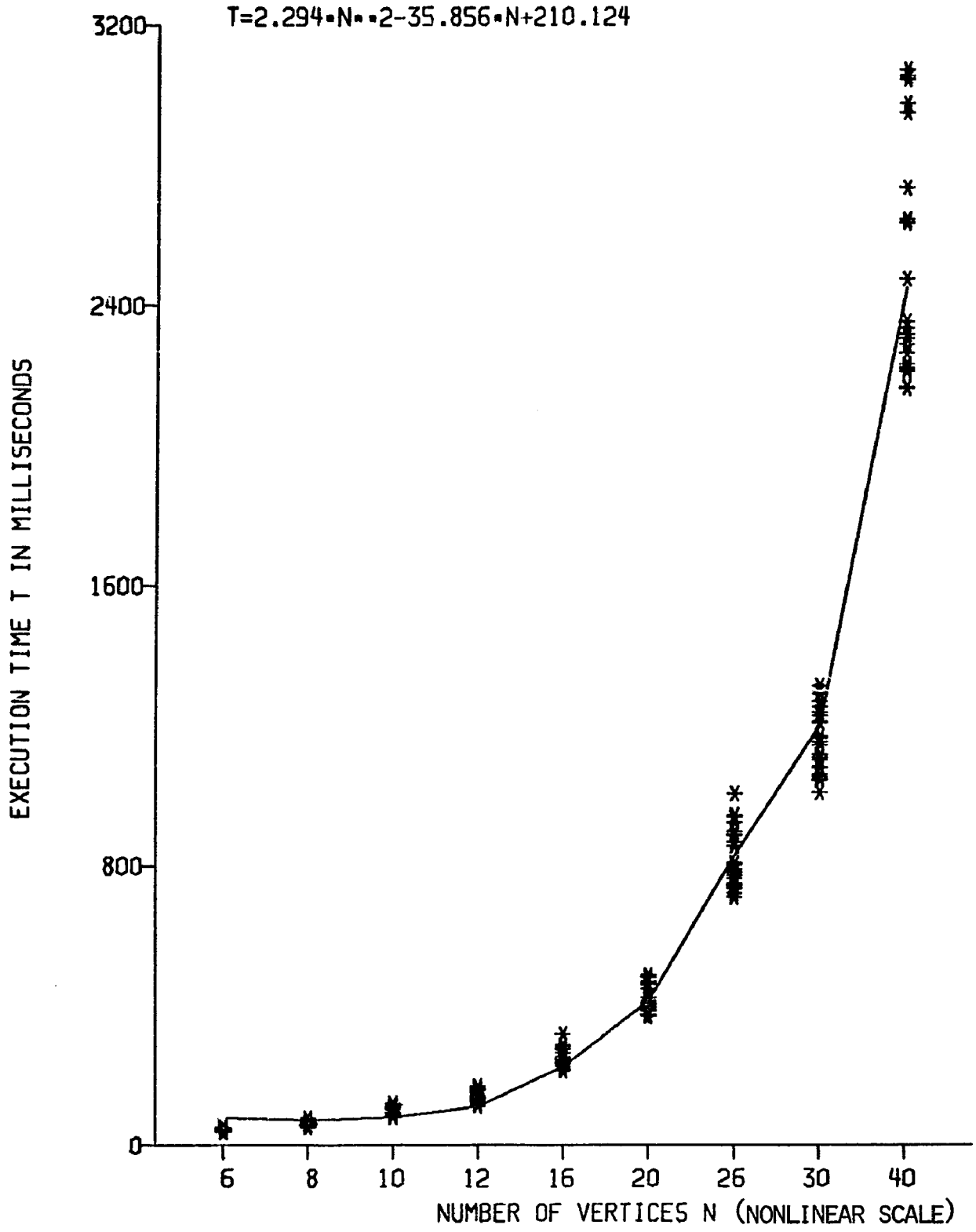


Figure 5.4.1.2 Plot of Schmidt and Druffel's Isomorphism Algorithm Using Isomorphic Regular Graphs.

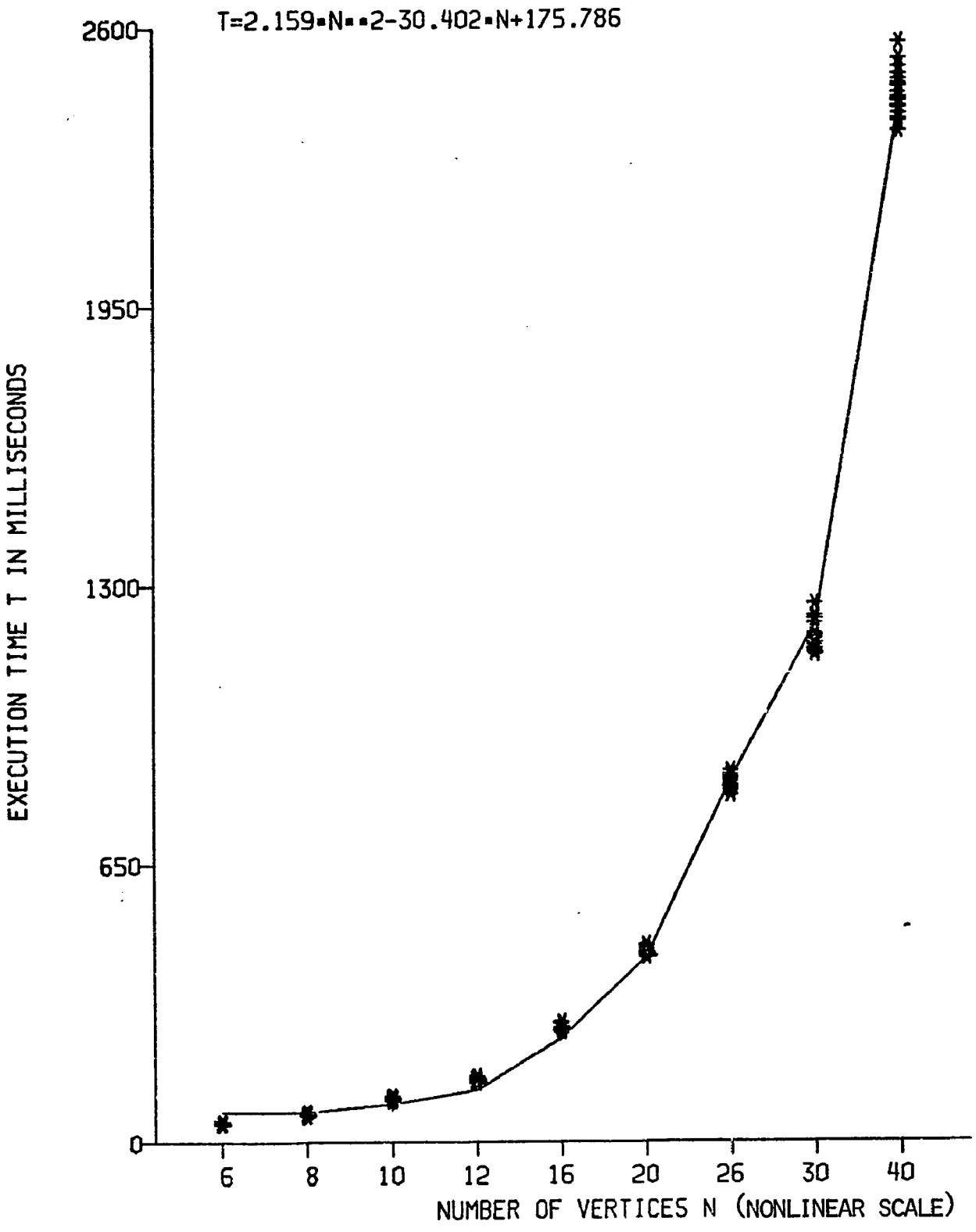


Figure 5.4.1.3 Plot of Graph Representation Algorithm (Schmidt and Druffel) Using Isomorphic Regular Graphs.

assignments summarized in Table 5.4.2, the isomorphism algorithm was effective in establishing for the class of regular graphs either nonisomorphism or isomorphism. Since the number of backtracks and vertex assignments of Table 5.4.2 for the class of nonisomorphic graphs were all zero, the isomorphism algorithm determined the nonisomorphism of the regular graphs from the initial partition.

For the sets of strongly regular graphs, the number of vertex assignments of Tables 5.4.3, 5.4.4, and 5.4.5 always fell between n^2 and n^3. This performance measurement suggested that the isomorphism algorithm was probably of experimental order $O(n^3)$ for the strongly regular sets of graphs.

Hence, for all the graphs tested, the distance matrix proved to be a very effective heuristic for solving the graph isomorphism problem. Also, since the number of backtracks for any set of observations or any pairs of graphs never exceeded n^3, it was concluded that the backtracking technique was quite stable and effective.

5.5 Comparison of the Algorithms

The four methods were compared based on the performance of their implementations and the performance of their algorithms.

5.5.1 Comparison of Performance of the Implementations

For the set of nonisomorphic regular graphs, the performance of the implementation of the new isomorphism algorithm was superior to that of the other three isomorphism algorithm, since the algorithm required the least amount of processing time, t = 1.89 milliseconds. Schmidt and Druffel's isomorphism algorithm, while slower than the new isomorphism algorithm, processed on the average all nonisomorphic

regular graphs in less than a minute and thus executed much faster than either Berztiss' or Ullmann's. These last two algorithms were very slow, Ullmann's being slower than Berztiss'.

For the isomorphic regular graphs, the new isomorphism algorithm processed the graphs faster than Schmidt and Druffel's, however, the execution time of the MSM representation algorithm was much slower than the distance matrix representation algorithm. Thus, in order to compare these methods based on execution time, the total execution times for all methods were calculated and are summarized in Table 5.5.1.1. The total raw execution times, t, for all methods were fitted by using polynomial and exponential functions of n. The equations which best fit the raw execution times are given with the corresponding R^2's in Table 5.5.1.2. These equations with the total raw execution times are plotted in Figures 5.5.1.1-5.5.1.4. Thus, based on total execution time, Schmidt and Druffel's algorithms, of order $O(n^2)$, were faster than the new algorithms, also of order $O(n^2)$. However, the new algorithms were much faster than those of Berztiss or Ullmann, both of which were very slow.

For the set of strongly regular graphs, it is obvious that Schmidt and Druffel's method was superior to the other three methods, since it was the only method of the four tested which was able to process any of the nonisomorphic strongly regular graphs.

5.5.2 Comparison of the Performance of the Graph Isomorphism Algorithms

For the nonisomorphic regular graphs, the new isomorphism algorithm and Schmidt and Druffel's isomorphism algorithm performed equally well. For the new isomorphism algorithm, nonisomorphism was established based

TABLE 5.5.1.1

PERFORMANCE OF IMPLEMENTATION OF FOUR METHODS BASED ON TOTAL EXECUTION TIMES
USING ISOMORPHIC REGULAR GRAPHS
(Time in milliseconds)

No. of Vertices	No. of Tests	New Algorithms		Berztiss		Ullmann		Schmidt and Druffel	
		Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.	Mean	St. Dev.
6	25*	703.19	55.73	32.08	4.72	1985.35	903.03	87.48	3.86
8	25	1031.78	162.50	71.60	28.97	4667.77	2816.30	125.64	8.80
10	25	1332.91	222.02	235.96	196.87	9108.78	6650.37	197.57	13.09
12	25	1709.74	101.68	1319.16	1707.21	26461.52	22335.22	282.39	16.12
16	25	2799.15	73.70	81683.76	167142.72	62026.73	45642.80	513.09	29.53
20	25**	4169.96	109.95	...	...	192578.49	124307.49	857.74	44.29
26	25	7049.16	55.49	...	...	...	...	1649.45	86.35
30	25	8957.09	144.74	...	...	...	...	2321.34	108.70
40	25	15886.94	151.22	...	...	...	...	4901.12	331.24

* Only 16 tests were used for New Algorithms.

** Only 10 tests were used for Ullmann's Algorithm.

TABLE 5.5.1.2

EQUATIONS OF FIT FOR TOTAL RAW EXECUTION TIMES
SUMMARIZED IN TABLE 5.5.1.1

Method	Equation	R^2(%)
New	$t = 9.828n^2 - 6.307n + 407.086$	99.90
Berztiss	$t = \exp(0.0362n^2 - 0.1420n + 6.6480) \times 26.04166 \times 10^{-3}$	84.93
Ullmann	$t = \exp(-0.0055n^2 + 0.4517n + 8.6299) \times 26.04166 \times 10^{-3}$	73.39
Schmidt and Druffel	$t = 4.453n - 66.258n + 385.910$	99.27

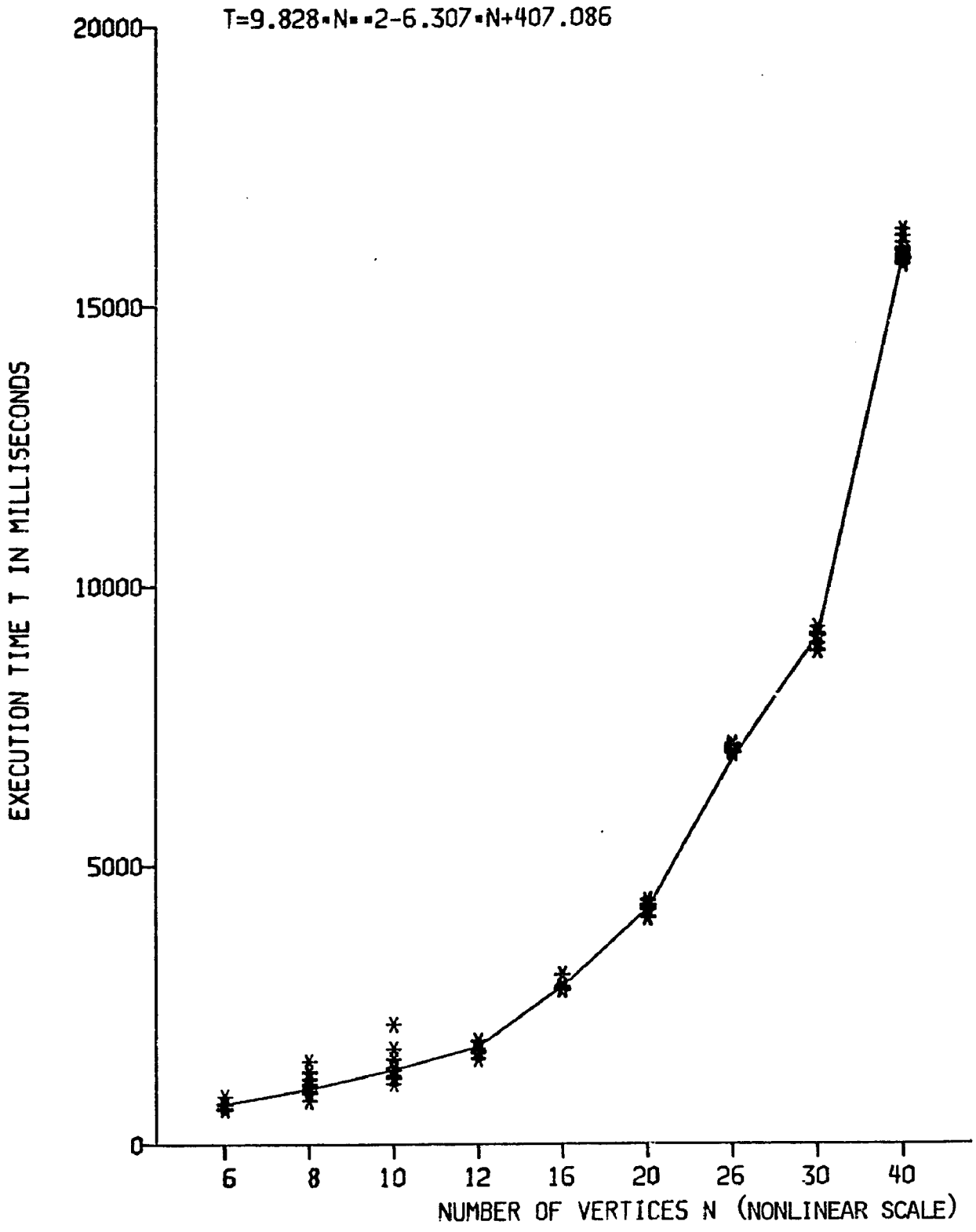


Figure 5.5.1.1 Plot of Total Time for New Algorithms Using Isomorphic Regular Graphs.

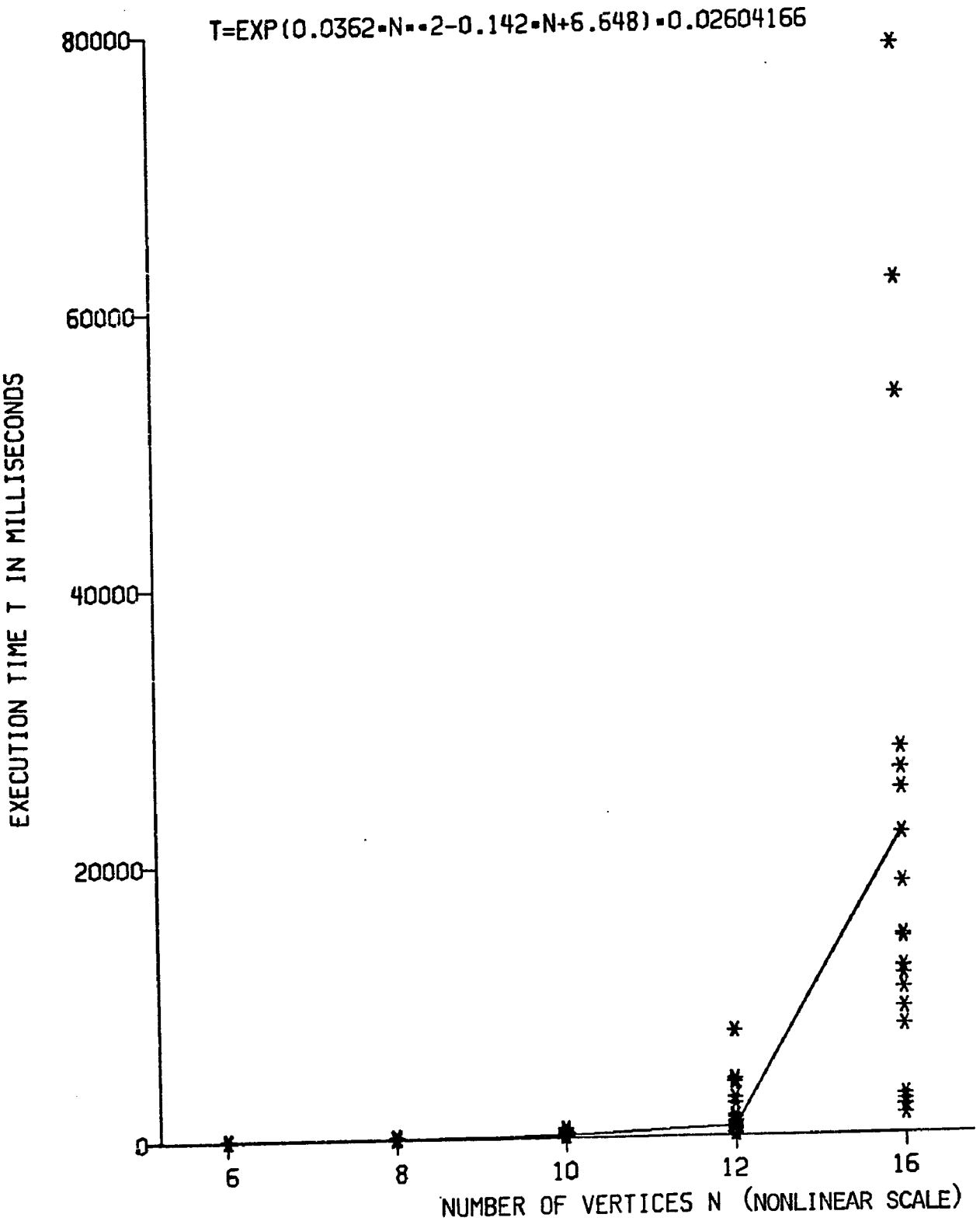


Figure 5.5.1.2 Plot of Total Time for Berztiss' Algorithms Using Isomorphic Regular Graphs.

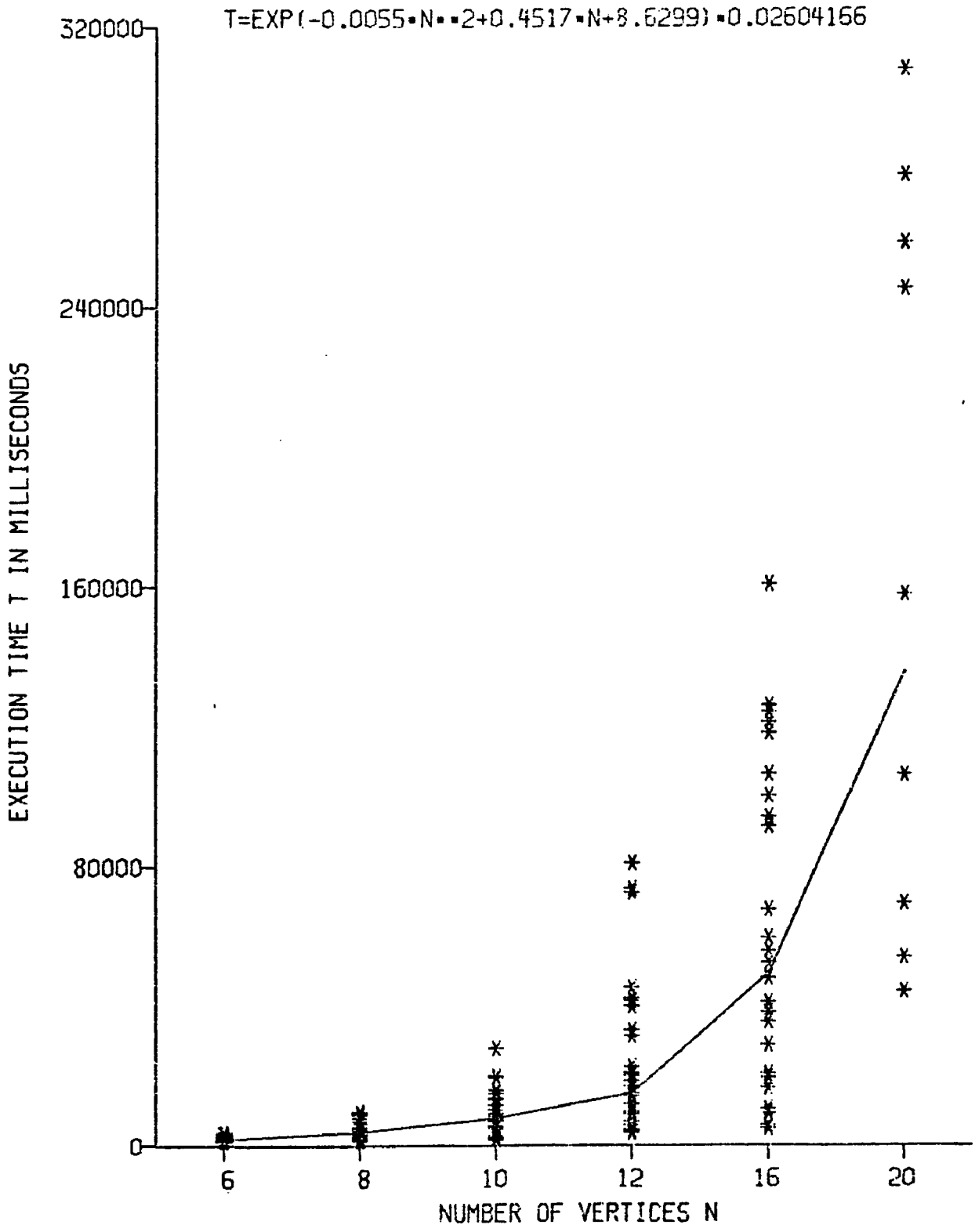


Figure 5.5.1.3 Plot of Total Time for Ullmann's Algorithms Using Isomorphic Regular Graphs.

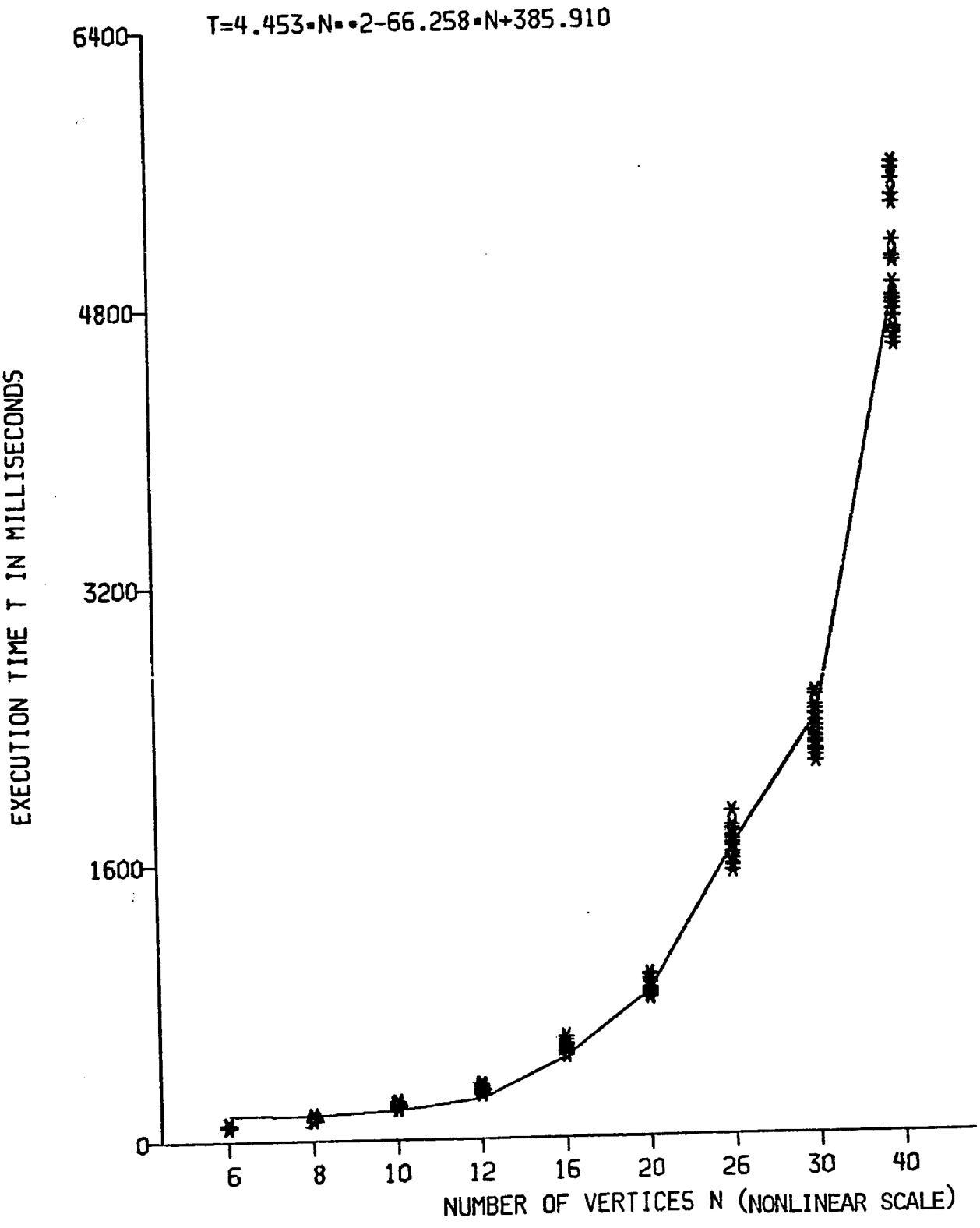


Figure 5.5.1.4 Plot of Total Time for Schmidt and Druffel's Algorithms Using Isomorphic Regular Graphs.

on the MSM's representation of the graphs, and for Schmidt and Druffel's isomorphism algorithm, nonisomorphism was determined from the initial partition. Of the two remaining isomorphism algorithms, Ullmann's was more effective than Berztiss', since the number of backtracks and vertex assignments were considerably less than those of Berztiss.

For the isomorphic regular graphs, based on the number of vertex assignments, the new isomorphism algorithm performed better than Schmidt and Druffel's for isomorphic regular graphs having 26 or more vertices and compared favorably with theirs for the other graphs. Again based on the number of backtracks and vertex assignments, Ullmann's algorithm performed better than Berztiss'.

For the strongly regular graphs, no comparisons could be made, since Schmidt and Druffel's algorithm was the only one of the four to process these graphs in less than 4 minutes execution time of the IBM 370/158.

5.6 Conclusions

The experimental data,contained in the Tables presented in the previous sections, provided a basis for choosing a graph isomorphism algorithm for particular types of graphs. The general case would probably not include graphs any more difficult than the set of nonisomorphic strongly regular graphs tested. Thus, Schmidt and Druffel's isomorphism algorithm was superior to the other three isomorphism algorithms, since it was usually able to process all graphs.

However, for the regular graphs tested, the new isomorphism algorithm, based on overall performance measurements, was superior to the other three algorithms. Thus, if the graphs being considered were

regular, the new algorithm should be used.

Both Berztiss' and Ullmann's algorithms required large amounts of execution time. Neither algorithm appeared to be practical for graph isomorphism problems.

LIST OF REFERENCES

Barrow, H. G. and J. R. Popplestone. 1971. "Relational Descriptions in Picture Processing". In Machine Intelligence. ed. by B. Meltzer and D. Michie. Edinburgh University Press, 377-396.

Barrow, H. G., A. P. Ambler, and R. M. Burstall. 1972. "Some Techniques for Recognizing Structures in Pictures". In Frontiers Of Pattern Recognition. ed. by S. Watanabe. Academic Press, New York, 1-29.

Berztiss, A. T. 1973. A Backtrack Procedure for Isomorphism of Directed Graphs. J. ACM, 20, 3, 365-377.

Brown, W. 1963. Enumeration of Non-separable Planar Maps. Canad. J. Math., 15, 526-545.

Brown, W. 1966. On the Enumeration of Non-planar Maps. Mem. Amer. Math. Soc., No. 65.

Bussemaker, F. C. and J. J. Seidel. 1970. Symmetric Hadamard Matrices of Order 36. T.H.-Report 70-WSK-02, Dept. of Mathematics, Technological University Eindhoven, Netherlands.

Collatz, L. and U. Sinogowitz. 1957. Spektrem endlicher Graphs. Abh. Math. Sem. Univ. Hamburg., 21, 63-77.

Corneil, D. G. 1968. Graph Isomorphism. Ph.D. Diss., Dept. of Comp. Sc., Univ. of Toronto, Toronto, Ontario, Canada.

Corneil, D. G. 1974. The Analysis of Graph Theoretical Algorithms. Tech Rep. No. 65, Dept. of Comp. Sc., Univ. of Toronto, Ontario, Canada.

Cornog, J. R., and H. L. Bryan. 1966. Search Methods Used with Transistor Patent Applications. IEEE-Spectrum, 3, 2, 116-121.

de Bruijn, N. G. 1964. "Polya's Theory of Counting". In Applied Combinatorial Mathematics. ed. by E. F. Beckenbach. Wiley, New York.

Druffel, L. E. 1975. Graph Related Algorithms Isomorphism, Automorphism, and Containment. Ph.D. Diss., Dept. of Comp. Sc., Vanderbilt Univ.

Fisher, M. E. 1966. On Hearing the Shape of a Drum. J. Comb. Th., 1, 105-125.

Floyd, R. W. 1962. Algorithm 97, Shortest Path, Comm. ACM, 5, 6, 345.

Grimsdale, R. L., F. H. Sumner, C. J. Tunis and T. Kilburn. 1959. A System for Automatic Recognition of Patterns. Proc. IEEE, 106, 211-221.

Harary, F. 1960. Unsolved Problems in the Enumeration of Graphs. Magyar Tud. Akad. Mat. Kutato Int. Kozl., 5, 63-95.

Harary, F. 1962. The Determinant of the Adjacency Matrix of a Graph. SIAM Rev., 4, 3, 202-210.

Harary, F. 1964. "Combinatorial Problems in Graphic Enumeration". In Applied Combinatorial Mathematics. ed. E. F. Beckenbach. Wiley, New York.

Hoffman, A. J. 1963. On the Polynomial of a Graph. Amer. Math. Monthly, 70, 30-36.

Hopcroft, J. E. and R. E. Tarjan. 1972. "Isomorphism of Planar Graphs". In Complexity of Computer Computations. ed. by R. E. Miller and J. W. Thatcher. Plenum Press, New York, 143-150.

Hopcroft, J. E. and J. K. Wong. 1974. Linear Time Algorithm for Isomorphism of Planar Graphs. Proc. 6th Annual ACM Symp. on Theory of Computing, Seattle, Wash., 172-184.

Karp, R. M. 1972. "Reducibility Among Combinatorial Problems". In Complexity of Computer Computations. ed. by R. E. Miller and J. W. Thatcher. Plenum Press, New York, 85-103.

Knodel, W. 1971. Ein Verfahren zur Feststellung der Isomorphic von endlichen, zusammenhangenden Graphen. Computing, 8, 329-334 (Abstract).

Levi, G. 1974. Graph Isomorphism: A Heuristic Edge-partioning-oriented Algorithm. Computing, 12, 291-313.

Lynch, M. F., J. M. Harrison, W. G. Town, and J. E. Ash. 1971. Computer Handling of Chemical Structure Information. Macdonald, London.

Morpurgo, R. 1971. Un Metodo Eurisicu per la Verifica dell' Isomorphismo di due Grafi Semplici non Orientati. Calcolo, 8, 1-31 (Abstract).

Paulus, A. J. L. 1973. Conference Matrices and Graphs of Order 26. T.H.-Report 73-WSK-06, Dept. of Mathematics, Technological University Eindhoven, Netherlands.

Pólya, G. 1937. Kombinatorische Anzahtlbestimmungen für Gruppen, Graphen und chemische Verbindungen. Acta. Math., 68, 145-254 (Abstract).

Proskurowski, A. 1974. Search for a Unique Incidence Matrix of a Graph. BIT. 14, 209-226.

Sakai, T., M. Nagao, and H. Matsushima. 1972. Extraction of Invariant Picture Substructures by Computer. Comput. Graphics and Image Process., 1, 1, 81-96.

Salton, G. 1968. Automatic Information Organization and Retrieval. McGraw-Hill, New York.

Salton, G. and E. H. Sussenguth, Jr. 1964. Some Flexible Information Retrieval Systems Using Structure Matching Procedures. Proc. AFIPS, 25, 587-597.

Saucier, G. 1971. Un Algorithme Efficace Recherchant 1'Isomorphisme de 2 Graphes. Rev. Française d'Informat. Recherche Operationelle, 5, R-3, 39-51. (Abstract).

Schmidt, D. C. and L. E. Druffel. 1976. A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices. J. ACM, 23, 3, 433-445.

Seshu, S. and M. Reed. 1961. Linear Graphs and Electrical Networks. Addison Wesley, Reading, Mass.

Shah, Y. J., G. I. Davida and M. K. McCarthy. 1974. Optimum Features and Graph Isomorphism. IEEE-TSMC, 4, 313-319.

Sherman, H. 1960. A Quasi-topological Method for the Recognition of Line Patterns. Information Processing-59. North Holland, Amsterdam, 232-238.

Sirovich, F. 1971. Isomofisco fra Grafi: un Algoritma Efficiente per Trovare tutti gli Isomorfismi. Calcolo, 8, 301-337. (Abstract).

Steen, J. P. 1969. Principe d'un Algorithme de Recherche d'un Isomorphisme entre deux Graphes. Rev. Française d'Informat. Recherche Operationelle, 3, R-3, 51-69. (Abstract).

Sussenguth, E. H., Jr. 1964. Structure matching in Information Processing. Ph.D. Diss., Harvard Univ.

Sussenguth, E. H., Jr. 1965. A Graph-theoretic Algorithm for Matching Chemical Structures. J. Chem. Doc., 5, 1, 36-43.

Tate, F. A. 1967. Handling Chemical Compounds in Information Systems. Ann. Rev. Inf. Sci. Tech., 2, 285-309.

Turner, J. 1967. Point Symmetric Graphs with a Prime Number of Points. J. Comb. Th., 3, 136-145.

Turner, J. 1968. Generalized Matrix Functions and the Graph Isomorphism Problem. SIAM J. Appl. Math., 16, 3, 520-526.

Tutte, W. T. 1962. A Cenus of Planar Triangulations, Cand. J. Math., 14, 21-38.

Tutte, W. T. 1963. A Cenus of Planar Maps. Canad. J. Math., 15, 249-271.

Tutte, W. T. 1964. A Cenus of Hamiltonian Polygons. Canad. J. Math., 14, 402-417.

Ullmann, J. R. 1973. Pattern Recognition Techniques. Crane, Russak & Company, Inc., New York.

Ullmann, J. R. 1976. An Algorithm For Subgraph Isomorphism. J. ACM, 23, 31-42.

Unger, S. H. 1964. GIT-A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism. C. ACM, 7, 1, 26-34.

Weinberg, L. 1966. A Simple and Efficient Algorithm for Determining Isomorphism of Planar Triply Connected Graphs. IEEE-TCT, 13, 142-148.

Yang, C. C. 1974. Generation of all Closed Partitions on a State Set of a Sequential Machine. IEEE Trans. Comput. C-23, 530-533.

Yang, C. C. 1975. Structure Preserving Morphisms of Finite Automata and an Application to Graph Isomorphism. IEEE Trans. Comput. C-24, 1133-1139.

APPENDICES

APPENDIX A

DEFINITIONS AND NOTATIONS

Terms and notations which are used in this dissertation are defined in this Appendix. The terms and notations are classified into three areas. First, the definitions and corresponding notations of graph properties and graph representations are presented. Special types of graphs are then defined. Next, since this dissertation is concerned with the analyses of graph isomorphism algorithms, terms used in the analyses of algorithms are defined and explained.

A.1 Graph Related Terms

Definition A.1.1. A directed graph G is a pair (V,A) where V is a finite set called the set of vertices, and $A \subseteq V \times V$ is a binary relation on V called the incidence relation or the set of arcs.

Definition A.1.2. The number of vertices of a graph is called the order of the graph.

Definition A.1.3. Two vertices v_i and v_j of a graph are said to be adjacent if $(v_i,v_j) \in A$ or $(v_j,v_i) \in A$.

Definition A.1.4. The indegree of a vertex v_i ($id(v_i)$) is the number of arcs that terminate on it, and the outdegree of a vertex v_i ($od(v_i)$) is the number of arcs that originate from it.

Definition A.1.5. Given an arc (v_i,v_j) in A, the vertices v_i and v_j in V are called the origin and terminus of the arc, respectively. A path P of G is a sequence of arcs such that for each pair of consecutive arcs in P, the terminus of the first arc, and the origin of the second arc coincide. The number of arcs along a path is called the length of the path.

Definition A.1.6. A circuit is a path such that the origin of the first arc coincides with the terminus of the last.

Definition A.1.7. A circuit of length 1 is called a loop.

Definition A.1.8. A vertex v_i is said to be reachable from a vertex v_j, if $v_i = v_j$ or there is a path from v_j to v_i.

Definition A.1.9. A graph G is said to be strongly connected if for every pair of vertices v_i and v_j in G, there is a path from v_i to v_j.

Definition A.1.10. A graph $G_s = (V_s,A_s)$ is called a subgraph of a graph $G = (V,A)$ if V_s V and if $A_s \subseteq A$ and $A_s \subseteq V_s \times V_s$.

Definition A.1.11. A component $G_c = (V_c,A_c)$ of a graph $G = (V,A)$ is a strongly connected subgraph of G.

Definition A.1.12. A graph G is said to be complete if every pair of vertices in G is joined by one arc.

Definition A.1.13. An adjacency matrix $[g_{ij}]$ of a graph G having n vertices in a n x n array in which

$$g_{ij} = \begin{cases} 1, \text{if } (v_i,v_j) \in A \\ 0, \text{ otherwise} \end{cases}$$

Definition A.1.14. An incidence matrix $[b_{ij}]$ of a graph G having n vertices v_1 through v_n and m arcs a_1 through a_m is an m x n array in which

$$b_{ij} = \begin{cases} 1, \text{ if } a_i \text{ is incident on } v_j \\ 0, \text{ otherwise.} \end{cases}$$

Definition A.1.15. An invertible (one-to-one and onto) mapping $\gamma: V \to V'$ is an isomorphism from $G = (V,A)$ to $G' = (V',A')$ iff it preserves graph incidences, i.e., for every arc $(v_i,v_j) \in A$ there is a corresponding arc $(\gamma(v_i), \gamma(v_j)) \in A'$ and vice versa.

Definition A.1.16. An automorphism is an isomorphism of G onto itself.

2.2 Special Types of Graphs

Definition A.2.1. If the incidence relation of a directed graph is symmetric, i.e., for distinct v_i and v_j in V, every arc (v_i,v_j) in A implies the arc (v_j,v_i) in A, then the graph is called undirected. An undirected graph is denoted by a pair (V,E) where E A is called the set of edges. In an undirected graph, the number of edges incident on a vertex is called the degree of the vertex, and a path is called a chain.

Definition A.2.2. A simple graph is a directed graph which has no loops.

Definition A.2.3. A finite undirected graph is planar if it can be drawn in a plane in such a way that no two of its edges intersect except, possibly at vertices.

Definition A.2.4. A directed (undirected) graph is called k-regular if for all vertices v_i, the indegree and outdegree (the degree) are each equal to k.

Definition A.2.5. An undirected graph G is connected if for each pair of vertices v_1 and v_2 in G, there is a chain between v_1 and v_2.

Definition A.2.6. An undirected graph is a polygon if it is connected and 2-regular.

Definition A.2.7. Two vertices of a graph are similar if there is an automorphism which maps one into the other. A graph is point symmetric if all vertices are pairwise similar.

Definition A.2.8. An undirected graph which is not complete and whose set of edges is not empty is strongly regular if constants $P^1_{11}, P^1_{12}, P^1_{22}, P^2_{11}, P^2_{12}, P^2_{22}$ exist such that

(1) for every two adjacent vertices v_i and v_j, i.e., $(v_i,v_j) \in E$, there are

(i) P^1_{11} vertices adjacent to both v_i and v_j

(ii) P^1_{12} vertices adjacent to v_i but not adjacent to v_j

(iii) P^1_{22} vertices adjacent to neither v_i nor v_j,

and

(2) for every two nonadjacent vertices v_i and v_j, i.e., $(v_i,v_j) \notin E$, there are

(i) P^2_{11} vertices adjacent to v_i and v_j.

(ii) P^2_{12} vertices adjacent to v_i but not adjacent to v_j.

(iii) P^2_{22} vertices adjacent to neither v_i nor v_j.

Definition A.2.9. A Latin square of order 6 consists of 36 triples selected from 6 symbols such that for each pair of coordinates, every pair of symbols occurs exactly once. The Latin Square graph (Bussemaker and Seidel, 1970) is a strongly regular graph which has as its vertices the 36 triples of a Latin square, and any two vertices are adjacent iff the corresponding triples have one symbol in common.

For example, the two vertices v_i and v_j which are represented respectively by the two triples (a,b,c) and (c,d,e), from the six

symbols a,b,c,d,e,f, are adjacent since they have the symbol c in common.

Definition A.2.10. A Steiner triple system of order 15 consists of 35 unordered triples selected from 15 symbols such that every unordered pair of symbols occurs in exactly one triple. A Steiner graph (Bussemaker and Seidel, 1970) is a strongly regular graph which has as its vertices the 35 triples of a Steiner triple system, and any two vertices are adjacent iff the corresponding triples have one symbol in common.

A.3 Algorithms

Definition A.3.1. An algorithm is a finite set of rules which gives a sequence of operations for transforming some input set into an output set. In terms of a graph isomorphism algorithm, the input set is the set of vertices and arcs defining the two graphs and, the output set is the function which defines an isormorphism or a message which indicates no isomorphism between these two graphs.

Definition A.3.2. An algorithm is said to be effective if its operations are sufficiently basic, i.e., the operations can be performed manually in a finite length of time.

Definition A.3.3. The order of an algorithm is a measure of the efficiency of the algorithm and is determined by considering the number of steps required for the algorithm to terminate as a function of input set size. The order of an algorithm is given by the 0 notation. In this dissertation the following conventions have been used

$$f(n) = O(f(n))$$

$$c \cdot O(f(n)) = O(f(n))$$

where c is a constant.

Definition A.3.4. A polynomial algorithm terminates in a number of steps bounded by some polynomial function of input set size. Algorithms which belong to the polynomial class of algorithms are said to be efficient.

Definition A.3.5. An exponential algorithm terminates in a number of steps bounded by some exponential function of input set size. Algorithms which belong to the exponential class of algorithms are impractical for very large problems but sometimes can be applied to smaller real problems.

Definition A.3.6. A factorial algorithm terminates in a number of steps bounded by some factorial function of input set size. The time required to execute algorithms of the factorial class even for a small problem is prohibitive.

Definition A.3.7. A heuristic algorithm is based on a strategy, sometimes called a "rule of thumb", which may or may not improve the efficiency of the algorithm in discovering the solution of a particular problem. For the graph isomorphism problem, the strategy is the application of some necessary conditions for isomorphism which may or may not immediately show that no isomorphism exist or may greatly reduce the number of functions to be checked for isomorphism.

Definition A.3.8. A backtracking algorithm is based on a depth-first tree search through the space of all possible solutions. A backtracking technique systematically attempts all possible solutions, eliminates potential solutions as quickly as possible, and never retries a potential

solution that has already been tried.

Definition A.3.9. All NP-complete problems can be solved in polynomial time on a nondeterministic Turing machine. If any one NP-complete problem can be solved in polynomial time on a one-tape deterministic Turing machine, then all NP-complete problems can be solved in polynomial time on such a machine. The graph isomorphism problem is not known to be NP-complete.

Definition A.3.10. The coefficient of determination R^2 is defined by

$$R^2 = 1 - SSE/SST$$

where SSE is the sum of squares of residuals $\Sigma(t_i - \hat{t}_i)^2$ for $\hat{t}_i$ the observed value and t_i the predicted value, and SST is the corrected total sum of squares, $\Sigma(t_i - \bar{t}_i)^2$ for $\bar{t}_i$ the mean value. In this dissertation, the coefficient of determination is given as $100 \cdot R^2$ which is the percent variation of t explained by the experimental equation.

APPENDIX B

A PL/I SOURCE LISTING OF THE RANDOM GRAPH GENERATING PROCEDURE GRAPHS

```
GRAPHS: PROCEDURE OPTIONS(MAIN);
/*                                                                    */
/*    THE MAIN PROCEDURE GRAPHS GENERATES TWO ISOMORPHIC N/2 RANDOM */
/* SIMPLE REGULAR GRAPHS OR TWO NONISOMORPHIC N/2 REGULAR GRAPHS.   */
/* FIRST, THE PROCEDURE MATRIX IS CALLED TO GENERATE THE GRAPH GK.  */
/* NEXT, THE RANDOM NUMBER GENERATOR PROCEDURE PRAND IS CALLED TO   */
/* SPECIFY A VERTEX ASSIGNMENT TO BE USED TO GENERATE THE SECOND    */
/* GRAPH GK.  IF GK IS TO BE ISOMORPHIC TO GJ, ISO='1'B, THEN THE   */
/* ROWS AND COLUMNS OF GJ ARE PERMUTED ACCORDING TO THE VERTEX      */
/* ASSIGNMENTS SPECIFIED BY THE ARRAY MAP AND GRAPH GK IS PRODUCED. */
/* IF GK IS TO BE NONISOMORPHIC TO GJ, ISO='0'B, THEN THE ROWS OF   */
/* GJ ARE PERMUTED ACCORDING TO THE VERTEX ASSIGNMENTS OF ARRAY MAP.*/
/* THE SET OF ARCS OF BOTH GRAPHS ARE WRITTEN TO THE FILE GRAPH.    */
/*                                                                    */
  DECLARE (GJ(64,64),GK(64,64),ISO) BIT(1),(NGJ,NGK,IOUT)
         FIXED BIN(15);
  DECLARE YFL FLOAT;
  DECLARE (NPGJ,NPGK) FIXED BIN(15);
         GET LIST(NUM_GR,NGJ,NGK,IOUT,ISO);
         NPGJ=NGJ*IOUT; NPGK=NGK*IOUT;
         PUT FILE(GRAPH) EDIT(NUM_GR,NGJ,NGK,NPGJ,NPGK) (F(4));
         DO II =1 TO NUM_GR;
          GJ='0'B; GK='0'B;
/*                                                                    */
/* CALL MATRIX TO GENERATE GRAPH GJ                                   */
/*                                                                    */
          CALL MATRIX(GJ,NGJ,IOUT);
/* BY CALLING PRAND, DETERMINE VERTEX ASSIGNMENTS TO BE USED IN     */
/* GENERATING GK.  VERTEX ASSIGNMENTS ARE STORED IN ARRAY MAP.      */
          BEGIN;
          DECLARE MAP(NGK) FIXED BIN(15), VERTEX(NGK) BIT(1);
          VERTEX='0'B; MAP=0;
          DO I=1 TO NGK;
          CALL PRAND(YFL); IYFL=YFL*NGK + 1;
           IF ¬VERTEX(IYFL) THEN DO; MAP(I)=IYFL; VERTEX(IYFL)='1'B;
                                 END;
           END;
          K=1;
          DO I=1 TO NGK;
           IF MAP(I)=0
            THEN DO;
    NXT_VERTEX: IF VERTEX(K) THEN DO; K=K + 1; GO TO NXT_VERTEX; END;
```

```
                    ELSE DO; VERTEX(K)='1'B; MAP(I)=K; K=K + 1; END;
                END;
          END;
/*                                                                    */
/* GENERATE GRAPH GK BY USING VERTEX ASSIGNMENT IN MAP                */
/*                                                                    */
        IF ISO
           THEN DO;
                DO I=1 TO NGJ; DO J=1 TO NGK;
                 IF GJ(I,J) THEN GK(MAP(I),MAP(J))='1'B;
                 END; END;
                PUT SKIP(2) EDIT((MAP(I) DO I=1 TO NGK)) ((64)F(3));
                END;
           ELSE DO I=1 TO NGJ;
                GK(MAP(I),*)=GJ(I,*);
                END;
/*                                                                    */
/* WRITE OUT ADJACENCY LIST FOR EACH GRAPH TO FILE GRAPH              */
/*                                                                    */
        DO I=1 TO NGJ; DO J=1 TO NGJ;
         IF GJ(I,J) THEN PUT FILE(GRAPH) EDIT(I,J) (F(3)); END; END;
        DO I=1 TO NGK; DO J=1 TO NGK;
         IF GK(I,J) THEN PUT FILE(GRAPH) EDIT(I,J) (F(3)); END; END;
         END;
        END;
END GRAPHS;


MATRIX: PROCEDURE(G,NG,IOUT);
/*                                                                    */
/*    THE PROCEDURE MATRIX GENERATES A SIMPLE RANDOM GRAPH HAVING     */
/* ADJACENCY MATRIX G, NG VERTICES, AND THE INDEGREE AND OUTDEGREE    */
/* OF EACH VERTEX EQUAL TO IOUT.                                      */
/*    THE ARRAY IN CONTAINS THE INDEGREE FOR EACH VERTEX.             */
/* THE ARRAY INDEX HOLDS THE LIST OF POSSIBLE VERTICES, I.E., ALL     */
/* VERTICES WHOSE INDEGREE < IOUT.                                    */
/* CON_TO ARRAY INSURES THAT A GIVEN VERTEX IS CONNECTED TO IOUT      */
/* DIFFERENT VERTICES.                                                */
/*                                                                    */
  DECLARE G(*,*) BIT(1),(NG,IOUT) FIXED BIN(15),
        (IN(64),INDEX(64)) FIXED BIN(15), CON_TO(64) BIT(1),YFL FLOAT;
/*                                                                    */
  MATRIX_BEGIN:
        G='0'B; IN=0;
        N=NG - 1;
        DO I=1 TO N; INDEX(I)=I + 1; END;
/*                                                                    */
/* RANDOMLY GENERATE THE FIRST NG-1 ROWS OF ADJACENCY MATRIX G SUCH   */
/* THAT THERE ARE NO SELF LOOPS,I.E., G(I,I)=0.                       */
/*                                                                    */
        DO I=1 TO NG - 1;
        CON_TO='0'B; CON_TO(I)='1'B;
         DO J=1 TO IOUT;
```

```
 USED:    CALL PRAND(YFL);
          IYFL=YFL*N + 1;
          IF CON_TO(INDEX(IYFL)) THEN GO TO USED;
          IN(INDEX(IYFL))=IN(INDEX(IYFL)) + 1;
          CON_TO(INDEX(IYFL))='1'B; G(I,INDEX(IYFL))='1'B;
          IF IN(INDEX(IYFL))=IOUT
           THEN DO; N=N - 1;
                DO K=IYFL TO N; INDEX(K)=INDEX(K + 1); END;
                END;
/*                                                                  */
/* CHECK NUMBER OF POSSIBLE VERTEX CHOICES REMAINING.               */
/*                                                                  */
          NPOSS=N;
/*                                                                  */
/* FOR ANY REMAINING VERTEX CHOICES WHICH HAVE BEEN USED IN ROW I   */
/* SUBSTRACT 1 FROM NPOSS.                                          */
/*                                                                  */
          IF N < IOUT
           THEN DO K=1 TO N; IF CON_TO(INDEX(K)) THEN NPOSS=NPOSS - 1;
                END;
/*                                                                  */
/* IF THE NUMBER OF CHOICES IS GREATER THAN THE REMAINING POSSIBLE  */
/* VERTEX CHOICES, THEN FROM THE ROWS LESS THAN I, MAKE MORE POSSI- */
/* BLE VERTEX CHOICES BY INCREASING THE INDEGREE OF THE VERTICES    */
/* WHICH WERE LEFT BY FREEING A VERTEX FROM THE GIVEN ROW.          */
/*                                                                  */
          IF IOUT - J > NPOSS
           THEN DO;
                NEND=N;
                DO KK=1 TO NEND;
                IF CON_TO(INDEX(KK)) THEN
                  DO;
                  K=INDEX(KK);
                  DO L=1 TO NG;
                  IF IN(L) = IOUT & L ¬= I
                   THEN DO II=1 TO I - 1;
                        IF G(II,L) & ¬G(II,K) & K ¬= II
                            THEN DO;
                       EXCHANGE: G(II,L)='0'B; G(II,K)='1'B;
                                 IN(L)=IN(L) - 1; IN(K)=IN(K) + 1;
                                 IF IN(K)=IOUT
                                 THEN DO; INDEX(KK)=L; NPOSS=NPOSS+1;
                                 GO TO KK_END; END;
                                 ELSE DO; N=N+1; INDEX(N)=L;
                                 NPOSS=NPOSS+1; GO TO L_END; END;
                                 END;
                            END;
         L_END:    END;
                   END;
  KK_END:     END;
           END;
          IF IOUT - J > NPOSS THEN GO TO MATRIX_BEGIN;
          END;
```

```
/*                                                                    */
/* PLACE VERTEX I ON AVAILABLE VERTEX ASSIGNMENT LIST.                */
/*                                                                    */
          IF IN(I + 1)=IOUT
           THEN IF IN(I) ¬= IOUT
                     THEN DO; N= N + 1; INDEX(N)=I; GO TO FIXED; END;
                     ELSE GO TO FIXED;
          DO K=1 TO N;
           IF INDEX(K)=I + 1
           THEN IF IN(I) ¬= IOUT
                     THEN INDEX(K)=I;
                     ELSE DO; N=N - 1;
                           DO KK=K TO N; INDEX(KK)=INDEX(KK + 1); END;
                           GO TO FIXED; END;
          END;
  FIXED:
         END;
/*                                                                    */
/* MAKE VERTEX ASSIGNMENTS FOR VERTEX NG.                             */
/*                                                                    */
         DO J=1 TO NG - 1;
         IF IN(J) ¬= IOUT
          THEN DO; IN(J)=  IN(J) + 1; G(NG,J)='1'B;
               END;
         END;
        IN_DEG=0;
         DO WHILE(IN_DEG ¬= IOUT);
        IN_DEG=IOUT;
         DO J=1 TO NG;
         IF IN(J) ¬= IOUT
          THEN DO K=1 TO NG - 1;
               IF ¬G(K,J) & K ¬= J THEN DO L=1 TO NG - 1;
                                  IF G(K,L) & ¬G(NG,L)
                                    THEN DO; G(NG,L)='1'B;
                                    G(K,L)='0'B; G(K,J)='1'B;
                                    IN(J)=IN(J) + 1; IF IN(J) ¬= IOUT
                                    THEN IN_DEG=IN(J);
                                    GO TO NXT_IN;
                                    END;
                                  END;
               END;
  NXT_IN: END;
         END;
END MATRIX;


PRAND: PROCEDURE(R);
/*                                                                    */
PRAND: PROCEDURE(R);
/*    THE PROCEDURE PRAND GENERATES RANDOM NUMBERS BETWEEN 0 AND 1    */
/* FROM A UNIFORM DISTRIBUTION.  THE RANDOM NUMBER IS RETURNED IN     */
/* PARAMETER R.  TO CALL THE PROCEDURE, DECLARE THE ENTRY POINT FOR   */
/* IX AS FIXED BIN(31).                                               */
```

```
/*                                                                    */
     DCL IX FIXED BIN (31) STATIC INITIAL (54321),R FLOAT;
     IX=IX*3125;
     IF IX > 65536 THEN IX=IX-(IX/65536)*65536;
     R=IX;
     R=R/65536.;
     RETURN;
END PRAND;
```

APPENDIX C

AN ASSEMBLY LANGUAGE LISTING OF THE TIMING PROCEDURE ASMTME

```
ASMTME   STARTASM
*
*      THE ROUTINE ASMTME ACCUMULATES EXECUTION TIME USED BY A TASK.
*  WHEN CALLING, IT IS NECESSARY TO PASS ONE PARAMETER TO ASMTME.  IF
*  THIS PARAMETER CONTAINS A NEGATIVE NUMBER, THEN ASMTME SETS THE
*  SYSTEM TIMER TO TWO BILLION TIME UNITS.  IF THE PARAMETER CONTAINS
*  A POSITIVE NUMBER, THEN THE REMAINING TIME IS SUBSTRACTED FROM THE
*  TWO BILLION TIME UNITS IN ORDER TO GET THE AMOUNT OF TIME USED BY
*  TASK.  THIS TIME IS RETURNED TO THE CALLING PROCEDURE IN UNITS OF
*  26.04166 MICROSECONDS THROUGH THE PARAMETER.
*
*  ESTABLISH OS LINKAGE
*
         L        R2,0(,R1)          GET PARM ADDRESS
         L        R3,0(,R2)          GET NUMBER
*
         LTR      R3,R3              DIFFERENCE OR ACTUAL TIME?
         BNP      SETTIME            NOT POS- GO SET TIME
*
         TTIMER   CANCEL,TU          GET THE REMAINDER FOR TASK
*
         L        R1,TIMEINTL        GET THE TIME INTERVAL
         SR       R1,R0              GET AMOUNT TIME USED
         ST       R1,0(,R2)          SAVE RESULT IN PARM
         B        RETURN             GO TO RETURN
*
SETTIME  DS       0H
         MVC   TIMEINTL(4),WKTME
         STIMER   TASK,TUINTVL=TIMEINTL   SET TIMER FOR LONG TIME
*
RETURN   DS       0H
         L        R13,4(,R13)        GET OS SAVE AREA ADDRESS BACK
*
         RETURN   (14,12),RC=0       RETURN CLEAN
*
*
SAVE     DC       18A(0)             OS SAVE AREA
WKTME    DC    F'2000000000'
TIMEINTL DC    F'2000000000'      TWO BILLION TIME UNITS
         CNOP  0,4
```

```
SAV       DS     18F                REG SAVE AREA
TEMP1     DS     D
TEMP2     DS     D
            END
```

APPENDIX D

THE PL/I IMPLEMENTATION OF THE NEW GRAPH ISOMORPHISM ALGORITHM

```
MAY: PROCEDURE OPTIONS(MAIN);
/*                                                                    */
/*    THE MAIN PROCEDURE MAY IMPLEMENTS ALGORITHM 2.  THE STATEMENT */
/* NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF ALGORITHM 2.  IN */
/* STEP 0, THE PROCEDURE CALLS GENGR TO CREATE THE ADJACENCY LIST   */
/* FOR GRAPH GJ (G) AND THE ADJACENCY MATRIX FOR GRAPH GK (G').     */
/* MOOREM IS THEN CALLED TO CREATE THE CORRESPONDING MOORE SEQUENT- */
/* IAL MACHINES (MSM'S) AJ (H,J) AND AK (H',J').  IN STEPS 1-15,
/* ALGORITHM 2 IS IMPLEMENTED.  THE REMAINDER OF THE PROCEDURE      */
/* GATHERS PERFORMANCE INFORMATION.  THE ASSEMBLY LANGUAGE ROUTINE  */
/* ASMTME IS USED TO OBTAIN THE EXECUTION TIME FOR SECTIONS OF A    */
/* PROCEDURE.                                                       */
/*    THE PROCEDURE CAN HANDLE TWO GRAPHS OF UP TO 64 VERTICES AND  */
/* 800 ARCS EACH, AND TWO MSM'S OF UP TO 100 STATES.  IF MORE       */
/* VERTICES, ARCS OR STATES ARE NEEDED, THEN THE DIMENSIONS OF THE  */
/* APPROPRIATE ARRAY NAMES MUST BE INCREASED.                       */
/*                                                                  */
  DECLARE (GJ(64,2),ADJYJ(800)) FIXED BIN(15) EXT,
         GK(64,64) BIT(1) EXT, ADJCNT(64) FIXED BIN(15) EXT,
         (NGJ,NGK,NPGJ,NPGK,NOPRAJ,NOPRAK) FIXED BIN(15) EXT;
  DECLARE (AJ(100,2),AK(100,2)) FIXED BIN(15) EXT;
  DECLARE (JS_AJ(200),JS_AK(200)) FIXED BIN(15) EXT;
  DECLARE (SVMAX,SVCLASS,MAXOUT,CLASSCNT) FIXED BIN(15) EXT;
  DECLARE (TMGR(25),TMTR(25),TMMAY(25)) FIXED BIN(31) EXT,
         (MNGR,MNTR,MNMAY,NO_GRAPH_PRS) FIXED BIN(31) EXT,
         (NOGR,TOTAL) FIXED BIN(31), (MEGR,METR,MEMAY,METOT) FLOAT
         DEC(16);
  DECLARE ITME FIXED BIN(31), ASMTME ENTRY(FIXED BIN(31));
/*                                                                  */
/* READ IN THE TWO GRAPHS AND CREATE CORRESPONDING MSM'S            */
/*                                                                  */
         GET FILE(GRAPH) LIST (NGPRS,NGJ,NGK,NPGJ,NPGK);
         GET LIST(NO_GRAPH_PRS);
         NOGR=NO_GRAPH_PRS; MNGR=0; MNTR=0; MNMAY=0; TMTR=0;
         TMGR=0; TMMAY=0;
  STEP0: CALL GENGR;
         IF(NGJ ^= NGK | NPGJ ^= NPGK)
           THEN GO TO NO_ISOMORPHISM;
           ELSE CALL MOOREM;
/*                                                                  */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                         */
```

```
          ITME= -1; CALL ASMTME(ITME);
/*            THE FOLLOWING CODE REPRESENTS ALGORITHM 2                  */
/*                                                                       */
  STEP1: IF NOPRAJ ^= NOPRAK
           THEN GO TO NO_ISOMORPHISM;
         IF SVCLASS ^= CLASSCNT | SVMAX ^= MAXOUT
            THEN GO TO NO_ISOMORPHISM;
         IF MAXOUT < NOPRAJ THEN MAXOUT=NOPRAJ;
  STEP2: BEGIN;
         DECLARE HASH(0:MAXOUT,0:MAXOUT,2) FIXED BIN(15),
          (FIRST,LAST) FIXED BIN(15);
         HASH=0; JS_NO=0; FIRST=1; NOUT=2; LAST=NOUT;
         DO I=1 TO NOPRAJ;
          JJ=JS_AJ(LAST);
  STEP2_1: J=JS_AJ(FIRST);
          IF HASH(J,JJ,1)=0
           THEN DO; JS_NO=JS_NO + 1; HASH(J,JJ,1)=JS_NO; END;
          IF FIRST=LAST | FIRST= LAST - 1
           THEN DO; HASH(J,JJ,2)=HASH(J,JJ,2) + 1;
                AJ(I,2)=HASH(J,JJ,1); FIRST=LAST + 1; LAST=LAST + NOUT;
                END;
           ELSE DO; JJ=HASH(J,JJ,1); FIRST=FIRST + 1;
                GO TO STEP2_1;
                END;
         END;
/*                                                                       */
         FIRST=1; LAST=NOUT;
         DO I=1 TO NOPRAK;
          JJ=JS_AK(LAST);
  STEP2_2: J=JS_AK(FIRST);
          IF HASH(J,JJ,1)=0 THEN GO TO NO_ISOMORPHISM;
          IF FIRST=LAST | FIRST=LAST - 1
           THEN DO;
                IF HASH(J,JJ,2)=0 THEN GO TO NO_ISOMORPHISM;
                HASH(J,JJ,2)= HASH(J,JJ,2) - 1; AK(I,2)=HASH(J,JJ,1);
                FIRST=LAST + 1; LAST=LAST + NOUT;
                END;
           ELSE DO; JJ=HASH(J,JJ,1); FIRST=FIRST + 1; GO TO STEP2_2;
                END;
         END;
         END;
/*                                                                       */
         BEGIN;
         DECLARE PART(NOPRAJ,NOPRAK,2) FIXED BIN(15);
         DECLARE (ICOMB,JCOMB) FIXED BIN(15);
         DECLARE (COLCNT(NGK),NOSAME) FIXED BIN(15);
         ON SIZE BEGIN; PUT SKIP(4) EDIT('DUE TO AN EXCESSIVE NUMBER',
                 ' OF POSSIBLE COMBINATIONS TO BE CHECKED, THIS METHOD'
                 ,' IS NOT APPLICABLE FOR THE ABOVE TWO GRAPHS.')
                 (A,A,A); GO TO CHECK_NO_GRAPHS;
                 END;
         PART=0; ICOMB=1; COLCNT=0;
```

```
  STEP3:
         DO I=1 TO NGJ;
          JCOMB=0; NOSAME=0;
  STEP4:
          DO J=1 TO NGK;
  STEP5:
          JPTR=I; KPTR=J;
  STEP6: IF AJ(JPTR,2)=AK(KPTR,2)
            THEN DO;
  STEP7_8:
                  PART(JPTR,KPTR,1)=AJ(JPTR,1);
                  PART(JPTR,KPTR,2)=AK(KPTR,1);
                  JPTR=AJ(JPTR,1); KPTR=AK(KPTR,1);
                  IF JPTR= -1 & KPTR= -1
                      THEN GO TO STEP7_8_END;
                  IF JPTR= -1 | KPTR= -1 THEN GO TO STEP9_1;
                  IF PART(JPTR,KPTR,1)=0 THEN GO TO STEP6;
                  IF PART(JPTR,KPTR,1)=-1 & PART(JPTR,KPTR,2) ^= -1
                   THEN GO TO STEP9_1;
  STEP7_8_END:
          COLCNT(J)=COLCNT(J) + 1;
          IF COLCNT(J) > 1 THEN NOSAME=NOSAME + 1;
          JCOMB=JCOMB + 1;
                END;
            ELSE DO;
  STEP9:
/*                                                                    */
/* PART (CLASS) IS SET EQUAL TO (-1,0) (0,0) IF STATES ARE NOT        */
/*  COMPATIBLE.                                                       */
/*                                                                    */
                  PART(JPTR,KPTR,1)=-1;
  STEP9_1:        JPTR=I; KPTR=J;
                  DO WHILE(KPTR > 0 & PART(JPTR,KPTR,1) ^= -1);
                   JSAV=JPTR; JPTR=PART(JPTR,KPTR,1);
                   PART(JSAV,KPTR,1)= -1; KPTR=PART(JSAV,KPTR,2);
                   END;
                  END;
  STEP10:
          END;
          IF JCOMB=0 THEN GO TO NO_ISOMORPHISM;
          IF JCOMB > 1 & JCOMB=NOSAME THEN JCOMB=JCOMB - 1;
   (SIZE): ICOMB=ICOMB*JCOMB;
  STEP11:
         END;
  STEP12: BEGIN;
/*                                                                    */
/* ISOAJ_AK AND ISOAK_AJ REPRESENT CANDIDATE, PARTITION , AND GAMMA   */
/*  OF ALGORITHM 2;                                                   */
/*                                                                    */
         DECLARE (ISOAJ_AK(ICOMB,NGK),ISOAK_AJ(ICOMB,NGJ))
          FIXED BIN(15);
         ISOAJ_AK=0; ISOAK_AJ=0; IALLOC=1;
```

```
  STEP13_14:
         DO I=1 TO NGJ;
          IASAV=IALLOC;
          DO J=1 TO NGK;
          IF PART(I,J,1) ^= -1 | PART(I,J,2)= -1
           THEN DO;
                DO K=1 TO IASAV;
                 IF ISOAK_AJ(K,J)=0
                  THEN IF ISOAJ_AK(K,I)=0
                   THEN DO; ISOAJ_AK(K,I)=J; ISOAK_AJ(K,J)=I;
                            GO TO STEP13_14_END; END;
                   ELSE DO; IALLOC=IALLOC + 1;
                        ISOAJ_AK(IALLOC,*)=ISOAJ_AK(K,*);
                        ISOAK_AJ(IALLOC,*)=ISOAK_AJ(K,*);
                        ISOAK_AJ(IALLOC,ISOAJ_AK(IALLOC,I))=0;
                        ISOAJ_AK(IALLOC,I)=J; ISOAK_AJ(IALLOC,J)=I;
                        END;
        STEP13_14_END:
                END;
                END;
           END;
          END;
          TMGR(NO_GRAPH_PRS)=IALLOC;
  STEP15: DO K=1 TO IALLOC;
          DO I=1 TO NGJ;
           JEND=GJ(I,2) + GJ(I,1) - 1;
           DO J=GJ(I,2) TO JEND;
           IF GK(ISOAJ_AK(K,I),ISOAJ_AK(K,ADJYJ(J)))
            THEN;
            ELSE GO TO STEP15_END;
           END;
          END;
          PUT SKIP(4) EDIT
          ('THE FOLLOWING DEFINES AN ISOMORPHISM FROM GJ TO GK') (A);
          PUT SKIP EDIT((I DO I=1 TO NGJ)) (F(3));
          PUT SKIP EDIT((ISOAJ_AK(K,I) DO I=1 TO NGJ)) (F(3));
  STEP15_END:
         END;
        END;
        END;
         GO TO CHECK_NO_GRAPHS;
  NO_ISOMORPHISM: PUT SKIP(2) EDIT
         ('NO ISOMORPHISM EXISTS FOR GIVEN GRAPHS') (A);
         IF ITME= -1 THEN; ELSE GO TO TOTALING;
/*                                                                  */
/*           END OF ALGORITHM 2                                     */
/*                                                                  */
/* THE REMAINDER OF THE PROCEDURE GATHERS PERFORMANCE INFORMATION.  */
/*                                                                  */
  CHECK_NO_GRAPHS:
/*                                                                  */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE       */
/*                                                                  */
```

```
        ITME= 1;  CALL ASMTME(ITME);
        TMMAY(NO_GRAPH_PRS)=ITME;
  TOTALING:
        MNGR=MNGR + TMGR(NO_GRAPH_PRS);
        MNTR=MNTR + TMTR(NO_GRAPH_PRS);
        MNMAY=MNMAY + TMMAY(NO_GRAPH_PRS);
        NO_GRAPH_PRS=NO_GRAPH_PRS - 1;
        IF NO_GRAPH_PRS ^= 0
         THEN GO TO STEP0;
         ELSE DO; PUT PAGE EDIT('ANALYSIS OF RUN USING ',NOGR,
              ' GRAPH PAIRS OF ',NGJ,' VERTICES EACH') (A,F(2),A,F(2),
              A); PUT SKIP(2) EDIT('NUMBER','MSM ISO','TRANSFO','MAY',
              'TOTAL') (COL(1),A,COL(15),A,COL(34),A,COL(60),A,COL(75)
              ,A); J=1;
              DO I=NOGR TO 1 BY -1;
              TOTAL=TMTR(I) + TMMAY(I);
              PUT SKIP EDIT(J,TMGR(I),TMTR(I),TMMAY(I),TOTAL) (F(2),
              COL(10),F(11),COL(30),F(11),COL(50),F(11),
              COL(70),F(11));
              J=J + 1;
              END;
              TOTAL=MNGR+MNTR+MNMAY;
              PUT SKIP(2) EDIT('TOTAL',MNGR,MNTR,MNMAY,TOTAL)
              (A,COL(10),F(11),COL(30),F(11),COL(50),F(11),
              COL(70),F(11));
              MEGR=FLOAT(MNGR)/NOGR; METR=FLOAT(MNTR)/NOGR;
              MEMAY=FLOAT(MNMAY)/NOGR; METOT=FLOAT(TOTAL)/NOGR;
              PUT SKIP(2) EDIT('MEAN',MEGR,METR,MEMAY,METOT)
              (A,COL(10),F(15,3),COL(30),F(15,3),COL(50),F(15,3),
              COL(70),F(15,3));
              PUT SKIP(2) EDIT('***EACH ABOVE UNIT=26.04166 X 10 -6',
              ' SEC.') (A,A);
              END;
END MAY;


GENGR: PROCEDURE;
/*                                                                  */
/*    THE PROCEDURE GENGR BUILDS AN ADJACENCY LIST FOR GRAPH GJ (G) */
/* AND AN ADJACENCY MATRIX FOR GRAPH GK (G').  ADJACENCY LISTS FOR  */
/* BOTHS GRAPHS ARE PRINTED.                                        */
/*                                                                  */
  DECLARE (GJ(64,2),ADJYJ(800)) FIXED BIN(15) EXT,
         GK(64,64) BIT(1) EXT, (ADJCNT(64),NGJ,NGK,NPGJ,NPGK) FIXED
         BIN(15) EXT;
/*                                                                  */
/* BUILD ADJACENCY LISTS FOR GRAPH GJ                               */
        GJ(*,1)=0; LAST=1; IPTR=1;
        DO I=1 TO NPGJ;
         GET FILE(GRAPH) LIST(II,J);
         IF II ^= LAST
          THEN DO;
               IF GJ(LAST,1) ^= 0 THEN GJ(LAST,2)=IPTR;
```

```
                ELSE GJ(LAST,2)=0;
               IPTR=I; LAST=II;
               END;
          GJ(II,1)=GJ(II,1) + 1; ADJYJ(I)=J;
         END;
         DO I=II TO NGJ;
          IF GJ(I,1)=0 THEN GJ(I,2)=0; ELSE GJ(I,2)=IPTR;
         END;
/*                                                                    */
/* BUILD ADJACENCY MATRIX AND ADJACENCY COUNT FOR GRAPH GK            */
/*                                                                    */
         GK='0'B; ADJCNT=0; LAST=1;
         DO I=1 TO NPGK;
          GET FILE(GRAPH) LIST(II,J);
          GK(II,J)='1'B; ADJCNT(II)=ADJCNT(II) + 1;
         END;
/*                                                                    */
/* PRINT ADJACENCY LIST FOR EACH VERTEX OF GRAPHS GJ AND GK           */
/*                                                                    */
         PUT PAGE EDIT('ADJACENCY LIST FOR GRAPH GJ') (A);
         DO I=1 TO NGJ;
          PUT SKIP EDIT(I) (F(2));
          IF GJ(I,2) ^= 0
           THEN DO; IEND=GJ(I,2) + GJ(I,1) - 1;
                PUT EDIT((ADJYJ(J) DO J=GJ(I,2) TO IEND)) (COL(6),
                 (50)F(3));
                END;
         END;
         PUT SKIP(4) EDIT('ADJACENCY LIST FOR GRAPH GK') (A);
         DO I=1 TO NGK;
          PUT SKIP EDIT(I,' ') (F(2),COL(5),A);
          DO J=1 TO NGK;
          IF GK(I,J) THEN PUT EDIT(J) (F(3));
          END;
         END;
END GENGR;


MOOREM: PROCEDURE;
/*                                                                    */
/*    THE PROCEDURE MOOREM CALLS TRANSFO (ALGORITHM 1) TO TRANSFORM   */
/* GRAPHS GJ (G) AND GK (G') INTO THE CORRESPONDING MSM'S AJ (H,J)    */
/* AND AK (H',J').
/*                                                                    */
  DECLARE (GJ(64,2),ADJYJ(800)) FIXED BIN(15) EXT,
         GK(64,64) BIT(1) EXT,(ADJCNT(64),NGJ,NGK,NOPRAJ,NOPRAK)
         FIXED BIN(15) EXT, (AJ(100,2),AK(100,2)) FIXED BIN(15) EXT;
  DECLARE A(100,64) BIT(1),(NO_WORDS,CNT(100)) FIXED BIN(15);
  DECLARE (JS_AJ(200),JS_AK(200)) FIXED BIN(15) EXT;
  DECLARE (SVMAX,SVCLASS,MAXOUT,CLASSCNT) FIXED BIN(15) EXT;
/*                                                                    */
/* BUILD MOORE MACHINE AJ FOR GRAPH GJ                                */
/*                                                                    */
```

```
        NO_WORDS=CEIL(NGJ/32);
        A='0'B;
        DO I=1 TO NGJ;
         IF GJ(I,1)=0 THEN GO TO INIT_CNT;
         J=GJ(I,2); DO K=1 TO GJ(I,1); A(I,ADJYJ(J))='1'B; J=J + 1;
                    END;
  INIT_CNT: CNT(I)=GJ(I,1);
        END;
        MAXOUT=0; CLASSCNT=0;
        CALL TRANSFO(A,CNT,NGJ,NO_WORDS,AJ,NOPRAJ,JS_AJ);
        SVCLASS=CLASSCNT; SVMAX=MAXOUT;
/*                                                                    */
/* PRINT MOORE MACHINE AJ                                             */
/*                                                                    */
        PUT PAGE EDIT('MOORE SEQUENTIAL MACHINE CORRESPONDING TO GJ')
        (A); PUT SKIP EDIT('STATE S   ','NEXT STATE H(S,I)  ',
        'OUTPUT J(S)') (A,A,COL(32),A);
        II=0;
        DO I=1 TO NOPRAJ;
        PUT SKIP EDIT(I,'--->  ',AJ(I,1),' ') (F(3),A,COL(15),F(3)
         ,COL(34),A);
        DO K=1 TO 2; II=II + 1; PUT EDIT(JS_AJ(II)) (F(3)); END;
        END;
/*                                                                    */
/* BUILD MOORE MACHINE AK FOR GRAPH GK                                */
/*                                                                    */
        A='0'B; DO I=1 TO NGK; A(I,*)=GK(I,*); CNT(I)=ADJCNT(I); END;
        CALL TRANSFO(A,CNT,NGK,NO_WORDS,AK,NOPRAK,JS_AK);
/*                                                                    */
/* PRINT MOORE MACHINE AK                                             */
/*                                                                    */
        PUT SKIP(4) EDIT
        ('MOORE SEQUENTIAL MACHINE CORRESPONDING TO GK') (A);
        PUT SKIP EDIT('STATE S   ','NEXT STATE H(S,I)  ',
        'OUTPUT J(S)') (A,A,COL(32),A);
        II=0;
        DO I=1 TO NOPRAK;
        PUT SKIP EDIT(I,'--->  ',AK(I,1),' ') (F(3),A,COL(15),F(3)
         ,COL(34),A);
        DO K=1 TO 2; II=II + 1; PUT EDIT(JS_AJ(II)) (F(3)); END;
        END;
END MOOREM;


TRANSFO: PROCEDURE(A,CNT,NG,NO_WDS,ANR,NO_PR_ST,JS_ANR);
/*                                                                    */
/*    THE PROCEDURE TRANSFO IMPLEMENTS ALGORITHM 1.  THE STATEMENT    */
/* NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF ALGORITHM 1.       */
/* GIVEN THE ADJACENCY MATRIX A (G) OF A GRAPH WITH NG (K(V))         */
/* VERTICES, THE PROCEDURE CREATES A MSM ANR (H,J) WITH NO_PR_ST (S)  */
/* STATES.  NXT_AVAL_ST REPRESENTS H OF ALGORITHM 1.                  */
/*                                                                    */
  DECLARE A(100,64) BIT(1) CONN,WORDA(100,2) BIT(32) DEF A,
```

```
        (NG,NO_WDS,NO_PR_ST) FIXED BIN(15),
        (CNT(100),ANR(100,2)) FIXED BIN(15) CONN;
  DECLARE JS_ANR(200) FIXED BIN(15) CONN;
  DECLARE (MAXOUT,CLASSCNT) FIXED BIN(15) EXT, CLASS(100,36)
        FIXED BIN(15);
  DECLARE ITME FIXED BIN(31), ASMTME ENTRY(FIXED BIN(31));
  DECLARE (TMGR(25),TMTR(25),TMMAY(25)) FIXED BIN(31) EXT,
        (MNGR,MNTR,MNMAY,NO_GRAPH_PRS) FIXED BIN(31) EXT;
/*                                                                  */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                         */
/*                                                                  */
        ITME= -1; CALL ASMTME(ITME);
/*                                                                  */
/*           THE FOLLOWING CODE REPRESENTS ALGORITHM 1              */
/*                                                                  */
  STEP1:
        NO_PR_ST=0; NXT_AVAL_ST=NG;
  STEP2:
        DO WHILE(NXT_AVAL_ST ¬= NO_PR_ST);
  STEP3:
         IBEGIN=NO_PR_ST + 1; NO_PR_ST=NXT_AVAL_ST;
  STEP4:
         DO I=IBEGIN TO NO_PR_ST;
  STEP5:
          IF CNT(I) > 1
          THEN DO;
  STEP6:
               DO J=1 TO I - 1;
               IF CNT(I)=CNT(J) THEN DO; DO K=1 TO NO_WDS;
                                         IF WORDA(I,K) ¬= WORDA(J,K)
                                         THEN GO TO STEP6_END;
                                         END;
                                         ANR(I,1)=ANR(J,1);
                                         GO TO STEP11;
                                         END;
  STEP6_END:   END;
  STEP7:
               NXT_AVAL_ST=NXT_AVAL_ST + 1;
  STEP8:
               DO K=1 TO NG;
               IF A(I,K) THEN
               A(NXT_AVAL_ST,*)=BOOL(A(NXT_AVAL_ST,*),A(K,*),'0111'B);
               END;
  STEP9:
               ANR(I,1)=NXT_AVAL_ST;
               CNT(NXT_AVAL_ST)=0;
               DO K=1 TO NG;
                IF A(NXT_AVAL_ST,K)
                 THEN CNT(NXT_AVAL_ST)=CNT(NXT_AVAL_ST) + 1;
               END;
             END;
          ELSE
  STEP10:
```

```
                IF CNT(I)=0
                  THEN ANR(I,1)= -1;
                  ELSE DO J=1 TO NG;
                       IF A(I,J) THEN DO; ANR(I,1)=J; GO TO STEP11;
                                      END;
                       END;
  STEP11:
          END;
         END;
         JS_INDEX=1; NO=2;
/*                                                                    */
  STEP12AND14: IBEGIN=JS_INDEX;
         DO JJ=1 TO NG;
          NEXT=0; II=0; IDIFF=0;
          DO WHILE(NEXT ^= NO_PR_ST);
           II=II + 1;
           IF ANR(II,1) > NEXT | ANR(II,1)=JJ
            THEN DO; IF A(II,JJ) THEN IDIFF=IDIFF + 1;
                 IF ANR(II,1) > NEXT THEN NEXT=ANR(II,1);
                 END;
          END;
          JS_ANR(IBEGIN)=IDIFF;
          IF MAXOUT < JS_ANR(IBEGIN) THEN MAXOUT=JS_ANR(IBEGIN);
          IBEGIN=IBEGIN + NO;
         END;
         LAST=NG; JJ=0;
         DO WHILE(LAST < NO_PR_ST);
          JJ=JJ + 1;
          IF ANR(JJ,1) > LAST
           THEN DO; LAST=ANR(JJ,1); NEXT=0; II=0; IDIFF=0;
                DO WHILE(NEXT ^= NO_PR_ST);
                 II=II + 1;
                 IF ANR(II,1) > NEXT
                  THEN DO; NEXT=ANR(II,1);
                       DO J=1 TO NO_WDS;
                       IF WORDA(JJ,J) = (WORDA(JJ,J) & WORDA(II,J))
                        THEN; ELSE GO TO ANOTH_ST;
                       END;
                       IDIFF=IDIFF + 1;
             ANOTH_ST: END;
                END;
                JS_ANR(IBEGIN)=IDIFF;
                IF MAXOUT < JS_ANR(IBEGIN)
                 THEN MAXOUT=JS_ANR(IBEGIN);
                IBEGIN=IBEGIN + NO;
                END;
         END;
         JS_INDEX=JS_INDEX + 1;
/*                                                                    */
  STEP13AND14: IBEGIN=JS_INDEX; MAX=0;
         DO JJ=1 TO NG;
         NSNS=0;
         DO J=1 TO NG;
```

```
          IF A(JJ,J) THEN IF A(J,JJ) THEN NSNS=NSNS + 1;
          END;
        JS_ANR(IBEGIN)=NSNS;
        IF MAX < JS_ANR(IBEGIN) THEN MAX=JS_ANR(IBEGIN);
        IBEGIN=IBEGIN + NO;
        END;
        LAST=NG; JJ=0;
        DO WHILE(LAST < NO_PR_ST);
         JJ=JJ+1;
         IF ANR(JJ,1) ¬< LAST
          THEN DO; LAST=ANR(JJ,1); CLASS(CLASSCNT + 1,*)=0;
               DO J=1 TO NG;
               IF A(JJ,J)
                THEN DO; IFROM=JS_INDEX + (J-1)*NO;
                CLASS(CLASSCNT+1,JS_ANR(IFROM))=
                CLASS(CLASSCNT+1,JS_ANR(IFROM)) + 1;
                     END;
               END;
               DO J=1 TO CLASSCNT;
               DO K=1 TO MAX;
               IF CLASS(CLASSCNT+1,K) ¬= CLASS(J,K)
                THEN GO TO NXT_CLASS;
                END;
               NSNS=MAX + J; GO TO INIT_JS;
NXT_CLASS:     END;
               CLASSCNT=CLASSCNT + 1; NSNS=MAX + CLASSCNT;
INIT_JS:       IFROM=JS_INDEX + (LAST - 1)*NO;
               JS_ANR(IFROM)=NSNS;
               END;
        END;
        IF MAXOUT < MAX + CLASSCNT THEN MAXOUT=MAX + CLASSCNT;
/*                                                                    */
/*          END OF ALGORITHM 1                                        */
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME= 1;  CALL ASMTME(ITME);
        TMTR(NO_GRAPH_PRS)=TMTR(NO_GRAPH_PRS) + ITME;
END TRANSFO;
```

APPENDIX E

BERZTISS' BACKTRACKING ALGORITHM AND PL/I IMPLEMENTATION

E.1 Algorithms

E.1.1 Algorithm 3

Algorithm 3 generates a set of K-formulas that represent a given graph D = (A,P). A K-formula is a K-formula of the vertex whose name is the left most vertex name in the K-formula, and this vertex is the leading vertex of the K-formula.

Step 1. For every isolated vertex a $\in$ A write the K-formula.

Step 2. For every vertex b from which originate arcs (b,t_1), (b,t_2), $\ldots,(b,t_k)$ write the K-formula $**\ldots*bt_1t_2\ldots t_k$, where k K-operators precede the b.

Step 3. Combine the K-formulas according to the following substitution rule: If there exists a K-formula of a vertex and there exists another K-formula in which the name of the vertex appears, substitute the K-formula of the vertex for this name. Apply the substitution rule until it can no longer be applied.

Step 4. (Check step.) Denote the K-formulas produced in Step 3 by $f_1,f_2,\ldots,f_n$, and the leading vertex of an f_i by a_i. If some f_i contains as a subformula a K-formula of vertex b in which a_i occurs, and the b occurs in one of $f_1,\ldots,f_{i-1},\ldots,f_{i+1}$, $\ldots,f_n$, extract the K-formula of b from f_i, inserting b in its place, substitute what now remains of f_i into this formula, and return to Step 3. Otherwise stop.

E.1.2 Algorithm 4

Algorithm 4, given a K-formula, creates arrays N,S, and T. Array N contains the vertex symbols in the order they have in the K-formula. Array S contains the structural information which is represented by the K-formula. Array T is used in backtracking. Array P, which must be initialized to zeros, is used for temporary storage. The algorithm makes use of a stack (last-in-first-out push-down store). The K-formula which is created by Algorithm 3 is given by $s_1 s_2 \ldots s_m$.

Step 1. Set J = 0, K = 0, LL = 1, S(1) = 0, i = 1.

Step 2. If s_i is a vertex symbol, then go to Step 5.

Step 3. Set K = K + 1, Switch = 0.

Step 4. Set i = i + 1. If i > m, then stop; else go to Step 2.

Step 5. Set J = J + 1, N(J) = s_i.

Step 6. If $P(s_i) = 0$, then set T(LL) = J, $P(s_i)$ = LL, LL = LL + 1.

Step 7. If J ≠ 1, then pop up the number LP from the stack and go to Step 9.

Step 8. Push $P(s_i)$ down K times; set K = 0, Switch = 1; go to Step 4.

Step 9. If Switch = 0, then set S(J) = LP and go to Step 8; else set S(J) = -LP and go to Step 4.

E.1.3 Algorithm 5

Algorithm 5 tests pairs of graphs for isomorphism. Arrays N, S and T which describe the first graph are created by Algorithm 4. Arrays L and B describe the set of arcs of the second graph. The vertex correspondences defining an isomorphism are generated in Array R. Initially the elements of R are assumed to be all zero. Arrays P, Q,

and the sign bits of T are used for temporary storage. Parameters n and k define respectively, the number of vertices in the reference graph and the number of entries in Array N. If the procedure stops without having produced an output, then the graphs are not isomorphic.

Step 1. Set V = 1, X = 0, M = 1.

Step 2. If L(M) $\neq$ 0, then set I = M, J = N(1), and go to Step 5.

Step 3. Set M = M + 1.

Step 4. If M $\leq$ n, then go to Step 2; else stop.

Step 5. Set X = X + 1, P(X) = L(I), Q(X) = M, R(J) = I, T(I) = -T(I).

Step 6. Advance in reference K-formula: Set V = V + 1; if V $\leq$ k, then set K = | S(V) |, M = P(K), J = N(V), and to to Step 8.

Step 7. An isomorphism exists: Output J, R(J) for J = 1,...,n. If all isomorphisms are to be found, then to the Step 13; else stop.

Step 8. If R(J) $\neq$ 0, then go to Step 11.

Step 9. Set I = B(M). If T(I) > 0 and, moreover, either or both L(I) $\neq$ 0 and S(V) < 0 hold, then go to Step 5.

Step 10. Set M = M + 1. If B(M) = 0, then go to Step 13, else go to Step 9.

Step 11. If B(M) = R(J), then go to Step 6.

Step 12. Set M = M + 1. If B(M) $\neq$ 0, then go to Step 11.

Step 13. Backtrack step: Set V = |T(X)|, J = N(V), K = R(J), R(J) = 0, T(K) = -T(K), M = Q(X) + 1, X = X - 1. If X = 0, then go to Step 4.

Step 14. If B(M) = 0, then go to Step 13; else go to Step 9.

E.2 PL/I Source Listing

```
BERZTIS: PROCEDURE OPTIONS(MAIN);
/*                                                                    */
/*    THE MAIN PROCEDURE BERZTIS IMPLEMENTS ALGORITHM 5.  THE         */
/* STATEMENT NAMES ROUGHLY CORESPOND TO THE STEP NUMBERS OF           */
/* ALGORITHM 5.  IN STEP 0, THE PROCEDURE CALLS BGENGR TO CREATE      */
/* ADJACENCY LISTS FOR REFERENCE GRAPH DR AND TEST GRAPH DT.  BALG2   */
/* (ALGORITHM 4) IS THEN CALLED TO CREATE THE K_FORMULA OF DR AND     */
/* THE DATA STRUCTURES WHICH REPRESENT THE K_FORMULA.  THESE DATA     */
/* STRUCTURES ARE USED BY THE PROCEDURE BERZTIS.  IN STEPS 1-14,      */
/* ALGORITHM 5 IS IMPLEMENTED.  THE REMAINDER OF THE PROCEDURE        */
/* GATHERS PERFORMANCE INFORMATION.  THE ASSEMBLY LANGUAGE ROUTINE    */
/* ASMTME IS USED TO OBTAIN THE EXECUTION TIME FOR SECTIONS OF A      */
/* PROCEDURE.                                                         */
/*    THE PROCEDURE CAN HANDLE TWO GRAPHS OF UP TO 64 VERTICES AND    */
/* 800 ARCS EACH.  IF MORE VERTICES OR ARCS ARE NEEDED, THEN THE      */
/* DIMENSIONS OF THE APPROPRIATE ARRAY NAMES MUST BE INCREASED.       */
/* THE VARIABLE AND ARRAY NAMES USED IN THIS PROCEDURE CORRESPOND     */
/* TO THOSE USED IN ALGORITHM 5.                                      */
/*                                                                    */
  DECLARE (DR(64),DRADJ(840),L(64),B(840)) FIXED BIN(15) EXT,
          (KN,N(801),S(801),T(64)) FIXED BIN(15) EXT,
          (V,X,M,I,J,K) FIXED BIN(15),
          (NR,NT,NPR,NPT) FIXED BIN(15) EXT;
  DECLARE (TMBALG(25),NO_GRAPH_PRS) FIXED BIN(31) EXT,
          (TMBER(25),NORJ(25),NOBACK(25),MNBER,MNBALG,MNRJ,MNBACK,
          NOGR,TOTAL) FIXED BIN(31),(MEBER,MEBALG,MERJ,MEBACK,METOT)
          FLOAT DEC(16);
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
/*                                                                    */
/* READ IN GRAPHS AND CREATE THE K_FORMULA FOR GRAPH DR               */
/*                                                                    */
          GETJ FILE(GRAPH) LIST(NGPRS,NR,NT,NPR,NPT);
          GET LIST(NO_GRAPH_PRS);
          MNBALG=0; MNBER=0; MNRJ=0; MNBACK=0;
          TMBALG=0; TMBER=0; NORJ=0; NOBACK=0; NOGR=NO_GRAPH_PRS;
  STEP0:  CALL BGENGR;
          IF NR ^= NT | NPR ^= NPT THEN GO TO NO_ISOMORPHISM;
          CALL BALG2;
/*                                                                    */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                           */
/*                                                                    */
          ITME= -1; CALL ASMTME(ITME);
/*                                                                    */
/*          THE FOLLOWING CODE REPRESENTS ALGORITHM 5                 */
/*                                                                    */
  STEP1:  BEGIN;
          DECLARE (P(NR),R(NR),Q(NR)) FIXED BIN(15);
          V=1; X=0; M=1; R=0;
  STEP2:  IF L(M) ^= 0
            THEN DO;
```

```
                I=M; J=N(1);
                GO TO STEP5;
                END;
  STEP3: M=M + 1;
  STEP4: IF M <= NR
           THEN GO TO STEP2;
           ELSE GO TO NO_ISOMORPHISM;
  STEP5: X=X + 1; P(X)=L(I);
         Q(X)=M; R(J)=I; T(I)= -T(I);
         NORJ(NO_GRAPH_PRS)=NORJ(NO_GRAPH_PRS) + 1;
  STEP6: V=V + 1;
         IF V <= KN
           THEN DO;
                K=ABS(S(V)); M=P(K);
                J=N(V); GO TO STEP8;
                END;
  STEP7: PUT SKIP(4) EDIT
          ('THE FOLLOWING DEFINES AN ISOMORPHISM FROM DR TO DT') (A);
          PUT SKIP EDIT((JJ DO JJ=1 TO NR)) (F(3));
          PUT SKIP EDIT((R(JJ) DO JJ=1 TO NR)) (F(3));
          GO TO CHECK_NO_GRAPHS;
  STEP8: IF R(J) ^= 0 THEN GO TO STEP11;
  STEP9: I=B(M);
         IF(T(I) > 0 & (L(I) ^= 0 | S(V) < 0))
           THEN GO TO STEP5;
  STEP10: M=M + 1;
         IF B(M) = 0 THEN GO TO STEP13; ELSE GO TO STEP9;
  STEP11: IF B(M) = R(J) THEN GO TO STEP6;
  STEP12: M=M + 1;
         IF B(M) ^= 0 THEN GO TO STEP11;
  STEP13:
         NOBACK(NO_GRAPH_PRS)=NOBACK(NO_GRAPH_PRS) + 1;
         V=ABS(T(X)); J=N(V); K=R(J); R(J)=0; T(K)= -T(K);
         M=Q(X) + 1; X=X - 1;
         IF X = 0 THEN GO TO STEP4;
  STEP14: IF B(M) = 0 THEN GO TO STEP13; ELSE GO TO STEP9;
         END;
  NO_ISOMORPHISM:
         PUT SKIP(2) EDIT
          ('NO ISOMORPHISM EXISTS FOR GIVEN GRAPHS') (A);
         IF ITME= -1 THEN; ELSE GO TO TOTALING;
/*                                                                  */
/*           END OF ALGORITHM 5                                     */
/*                                                                  */
/* THE REMAINDER OF THE PROCEDURE GATHERS PERFORMANCE INFORMATION . */
/*                                                                  */
  CHECK_NO_GRAPHS:
/*                                                                  */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE       */
/*                                                                  */
         ITME=1; CALL ASMTME(ITME);
         TIMER(NO_GRAPH_PRS)=ITME;
```

```
 TOTALING:
        MNBALG=MNBALG + TMBALG(NO_GRAPH_PRS);
        MNBER=MNBER + TMBER(NO_GRAPH_PRS);
        MNBACK=MNBACK + NOBACK(NO_GRAPH_PRS);
        MNRJ=MNRJ + NORJ(NO_GRAPH_PRS);
        NO_GRAPH_PRS=NO_GRAPH_PRS - 1;
        IF NO_GRAPH_PRS ^= 0
         THEN GO TO STEP0;
         ELSE DO; PUT PAGE EDIT('ANALYSIS OF RUN USING ',NOGR,
              ' GRAPH PAIRS OF ',NR,' VERTICES EACH') (A,F(2),A,F(2),
              A); PUT SKIP(2) EDIT('NUMBER','BACKTRACK',
              'NODE CORRESPONDENCES','BALG1-2','BERZTISS','TOTAL')
              (COL(1),A,COL(15),A,COL(34),A,COL(56),A,COL(75),A
              ,COL(95),A); J=1;
              DO I=NOGR TO 1 BY -1;
              TOTAL=TMBER(I) + TMBALG(I);
              PUT SKIP EDIT(J,NOBACK(I),NORJ(I),TMBALG(I),TMBER(I),
              TOTAL) (F(2),COL(10),F(11),COL(30),F(11),COL(50),F(11),
              COL(70),F(11),COL(90),F(11));
              J=J + 1;
              END;
              TOTAL=MNBALG + MNBER;
              PUT SKIP(2) EDIT
               ('TOTAL',MNBACK,MNRJ,MNBALG,MNBER,TOTAL)
               (A,COL(10),F(11),COL(30),F(11),COL(50),F(11),
               COL(70),F(11),COL(90),F(11));
              MEBACK=FLOAT(MNBACK)/NOGR; MERJ=FLOAT(MNRJ)/NOGR;
              MEBALG=FLOAT(MNBALG)/NOGR; MEBER=FLOAT(MNBER)/NOGR;
              METOT=FLOAT(TOTAL)/NOGR;
              PUT SKIP(2) EDIT
               ('MEAN',MEBACK,MERJ,MEBALG,MEBER,METOT)
               (A,COL(10),F(15,3),COL(30),F(15,3),COL(50),
               F(15,3),COL(70),F(15,3),COL(90),F(15,3));
              PUT SKIP EDIT('***EACH ABOVE UNIT=26.04166 X 10 -6',
               ' SEC.') (A,A);
              END;
END BERZTIS;


BGENGR: PROCEDURE;
/*                                                                  */
/*    THE PROCEDURE BGENGR BUILDS THE ADJACENCY LISTS FOR GRAPH DR  */
/* (DR, DRADJ) AND GRAPH DT (L,B).  ADJACENCY LISTS FOR BOTH GRAPHS */
/* ARE PRINTED.                                                     */
/*                                                                  */
  DECLARE (DR(64),DRADJ(840),L(64),B(840)) FIXED BIN(15) EXT,
         (NR,NT,NPR,NPT) FIXED BIN(15) EXT;
/*                                                                  */
/* BUILD AND PRINT ADJACENCY LIST FOR REFERENCE GRAPH DR            */
/*                                                                  */
         PUT PAGE EDIT('ADJACENCY LIST FOR REFERENCE GRAPH DR') (A);
         LAST=0; JPTR=0;
```

```
        DO I=1 TO NPR;
         GET FILE(GRAPH) LIST(II,J);
         IF II ^= LAST
          THEN DO;
               IF JPTR ^= 0 THEN DO; JPTR=JPTR + 1; DRADJ(JPTR)=0;
                                 END;
               PUT SKIP EDIT(II) (F(2)); LAST=II;
               DR(II)= JPTR + 1;
               END;
         JPTR=JPTR + 1; DRADJ(JPTR)=J; PUT EDIT(J) (F(3));
        END;
        DRADJ(JPTR + 1)=0;
        IF II ^= NR THEN DO; DR(NR)=0; PUT SKIP EDIT(NR) (F(2)); END;
/*                                                                    */
/* BUILD AND PRINT ADJACENCY LIST FOR TEST GRAPH DT                   */
/*                                                                    */
        PUT PAGE EDIT('ADJACENCY LIST FOR TEST GRAPH DT') (A);
        LAST=0; JPTR=0;
        DO I=1 TO NPT;
         GET FILE(GRAPH) LIST(II,J);
         IF II ^= LAST
          THEN DO;
               IF JPTR ^= 0 THEN DO; JPTR=JPTR + 1; B(JPTR)=0; END;
               L(II)=JPTR + 1;
               PUT SKIP EDIT(II) (F(2)); LAST=II;
               END;
         JPTR=JPTR + 1; B(JPTR)=J; PUT EDIT(J) (F(3));
        END;
        B(JPTR + 1)=0;
        IF II ^= NT THEN DO; L(NT)=0; PUT SKIP EDIT(NT) (F(2)); END;
END BGENGR;


BALG2: PROCEDURE;
/*                                                                    */
/*    THE PROCEDURE BALG2 IMPLEMENTS ALGORITHM 4.  THE STATEMENT      */
/* NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF ALGORITHM 4.  IN   */
/* STEP 0, THE PROCEDURE CALLS BALG1 (ALGORITHM 3) TO CREATE THE      */
/* K_FORMULA (K_FOR) FOR GRAPH DR.  BALG2 USES THIS K_FORMULA TO      */
/* CREATE THE DATA STRUCTURES USED BY BERZTIS.  THESE DATA            */
/* STRUCTURES REPRESENT THE STRUCTURAL INFORMATION CONTAINED IN THE   */
/* K_FORMULA REPRESENTATION OF DR.  THE VARIABLE AND ARRAY NAMES      */
/* CORRESPOND TO THOSE USED IN ALGORITHM 4.  BALG2 ASSUMES THAT THE   */
/* GRAPH DR CAN BE DESCRIBED BY A SINGLE K_FORMULA.                   */
/*                                                                    */
  DECLARE (NR,NT,KN,N(801),S(801),T(64)) FIXED BIN(15) EXT,
         (SWITCH,ST_PTR,SI,LP,J,K,LL,I) FIXED BIN(15), CHARNODE CHAR(2)
         ,NUMNODE PIC'99' DEF CHARNODE,K_FOR CHAR(2402) VAR EXT;
  DECLARE (TMBALG(25),NO_GRAPH_PRS) FIXED BIN(31) EXT;
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
/*                                                                    */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                           */
```

```
        ITME= -1; CALL ASMTME(ITME);
/*                                                                    */
/* CALL BALG1 (ALGORITHM 3) TO GENERATE THE K_FORMULA OF GRAPH DR     */
/*                                                                    */
  STEP0: CALL BALG1;
        M=LENGTH(K_FOR);
/*                                                                    */
/*          THE FOLLOWING CODE REPRESENTS ALGORITHM 4                 */
/*                                                                    */
  STEP1: BEGIN;
        DECLARE (STACK(NR,2),P(NR)) FIXED BIN(15);
        STACK=0; P=0; ST_PTR=0;
        J=0; K=0; LL=1; S(1)=0; I=1;
  STEP2_4: DO WHILE(I < = M);
        IF SUBSTR(K_FOR,I,1) ¬= '*'
          THEN DO;
  STEP5:        J=J + 1; CHARNODE=SUBSTR(K_FOR,I,2);
                I=I + 1; SI=NUMNODE;
                N(J)=SI;
  STEP6:        IF P(SI) = 0
                  THEN DO;
                       T(LL)=J; P(SI)=LL; LL=LL + 1;
                       END;
  STEP7:        IF J  = 1
                  THEN DO;
                       LP=STACK(ST_PTR,1);
                       STACK(ST_PTR,2)=STACK(ST_PTR,2) - 1;
                       IF STACK(ST_PTR,2) = 0 THEN ST_PTR=ST_PTR - 1;
  STEP8_9:             IF SWITCH = 0
                       THEN DO;
                       S(J)=LP;
                       ST_PTR=ST_PTR + 1;
                       STACK(ST_PTR,1)=P(SI); STACK(ST_PTR,2)=K;
                       K=0; SWITCH=1;
                            END;
                       ELSE S(J)= -LP;
                       END;
                  ELSE DO;
                       ST_PTR=ST_PTR + 1;
                       STACK(ST_PTR,1)=P(SI); STACK(ST_PTR,2)=K;
                       K=0; SWITCH=1;
                       END;
                END;
            ELSE DO;
                 K=K + 1; SWITCH=0;
                 END;
        I=I + 1;
        END;
        KN=J;
        END;
/*                                                                    */
/*          END OF ALGORITHM 4                                        */
```

```
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME=1; CALL ASMTME(ITME);
        TMBALG(NO_GRAPH_PRS)=ITME;
END BALG2;


BALG1: PROCEDURE;
/*                                                                    */
/*    THE PROCEDURE BALG1 IMPLEMENTS ALGORITHM 3.  THE STATEMENT      */
/* NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF ALGORITHM 3.       */
/* SINCE ALL THE GRAPHS TESTED COULD BE SPECIFIED BY A SINGLE         */
/* K_FORMULA, IT IS NOT NECESSARY TO IMPLEMENT STEP 4 OF ALGORITHM    */
/* 3.  THE K_FORMULA IS STORED IN THE CHARACTER STRING K_FOR.  ALL    */
/* VERTICES ARE REPRESENTED BY A TWO CHARACTER NUMBER, I.E., VERTEX   */
/* 1 IS REPRESENTED BY THE CHARACTERS 01.  THE PROCEDURE PROCESSES    */
/* VERTEX BY VERTEX THE ADJACENCY LIST DRADJ, UNTIL ALL VERTICES      */
/* AND THEIR CORRESPONDING ARCS HAVE BEEN REPRESENTED. THE BEGIN-     */
/* NING OF THE K_FORMULA CONTAINS AS MANY ADJACENT VERTICES WITH      */
/* THEIR CORRESPONDING ARCS AS IS POSSIBLE.  THIS TYPE OF K_FORMULA   */
/* LEADS TO A MORE EFFICIENT EXECUTION OF BERZTIS (ALGORITHM 5).      */
/*                                                                    */
  DECLARE (DR(64),DRADJ(840),NR) FIXED BIN(15) EXT, K_FOR CHAR(2402)
         VAR EXT, FOUND BIT(1),CHARNODE CHAR(2),NUMNODE PIC '99' DEF
         CHARNODE,K_NODE CHAR(102) VAR,(JSAV,ASTER,LNKSTR,KIN) FIXED
         BIN(15),FIRST BIT(1);
/*                                                                    */
/*           THE FOLLOWING CODE REPRESENTS ALGORITHM 3                */
/*                                                                    */
  STEP0: BEGIN;
         DECLARE ROW(NR) BIT(1),ROWSTR BIT(NR) DEF ROW,BINONE BIT(NR);
         ROW='0'B; K_FOR=''; BINONE=¬ ROWSTR;
         DO WHILE(ROWSTR ¬= BINONE);
          LNKSTR=LENGTH(K_FOR);
  STEP1_2: IF LNKSTR=0
            THEN DO; NUMNODE=1; K_FOR=CHARNODE; ROW(1)='1'B; J=DR(1);
                 DO WHILE (DRADJ(J) ¬= 0);
                  NUMNODE=DRADJ(J); JSAV=DRADJ(J);
                  K_FOR='*' || K_FOR || CHARNODE;
                  J=J + 1;
                  END;
                 KIN=J-DR(1)+3;
  NEW_K:         LNKSTR=LENGTH(K_FOR);
  NXT_K:         IF KIN < LNKSTR & ROWSTR ¬= BINONE
                  THEN DO;
                  CHARNODE=SUBSTR(K_FOR,KIN,2); JSAV=NUMNODE;
                  IF  ROW(JSAV) & (DRADJ(DR(JSAV))=1)
                   THEN DO; CALL KDERIVE;
                        IF KIN=LNKSTR - 1
                         THEN K_FOR=SUBSTR(K_FOR,1,KIN-1) || K_NODE;
                         ELSE K_FOR=SUBSTR(K_FOR,1,KIN-1) || K_NODE ||
                             SUBSTR(K_FOR,KIN+2,LNKSTR-(KIN+1));
```

```
                        DO WHILE(SUBSTR(K_FOR,KIN,1) ='*');
                        KIN=KIN+1; END; KIN=KIN+2;
                        GO TO NEW_K;
                        END;
                     ELSE KIN=KIN+2;
                    GO TO NXT_K;
                    END;
                    ELSE GO TO NEXT_ROW;
                   END;
  STEP3: KIN=LNKSTR - 1; FOUND='0'B;
          DO WHILE(¬FOUND);
          CHARNODE=SUBSTR(K_FOR,KIN,2); JSAV=NUMNODE;
          IF  ROW(JSAV)
            THEN DO; CALL KDERIVE;
                 IF KIN = LNKSTR - 1
                  THEN K_FOR=SUBSTR(K_FOR,1,KIN-1) || K_NODE;
                  ELSE K_FOR=SUBSTR(K_FOR,1,KIN-1) || K_NODE ||
                             SUBSTR(K_FOR,KIN+2,LNKSTR-(KIN+1));
                 FOUND='1'B;
                 END;
            ELSE DO; KIN=KIN - 1; DO WHILE(SUBSTR(K_FOR,KIN,1)='*');
                     KIN=KIN - 1; END;
                     KIN=KIN - 1;
                 END;
          END;
  NEXT_ROW:
         END;
KDERIVE: PROCEDURE;
         ROW(JSAV)='1'B; K_NODE=''; JJ=DR(JSAV);
         DO WHILE(DRADJ(JJ) ¬= 0);
          NUMNODE=DRADJ(JJ); K_NODE=K_NODE || CHARNODE;
          JJ=JJ + 1;
          END;
         NUMNODE=JSAV; K_NODE=CHARNODE || K_NODE;
         DO ASTER = DR(JSAV) TO JJ-1;
          K_NODE='*' || K_NODE;
         END;
END KDERIVE;
         END;
/*                                                                    */
/*          END OF ALGORITHM 3                                        */
/*                                                                    */
         PUT SKIP(4) EDIT('K_FORMULA OF REFERENCE GRAPH DR') (A);
         PUT SKIP LIST(K_FOR);
END BALG1;
```

APPENDIX F

ULLMANN'S REFINEMENT/BACKTRACKING ALGORITHM AND PL/I IMPLEMENTATION

F.1 Algorithm 6

Algorithm 6 finds an isomorphism, if one exists, between two graphs G_A and G_B. The algorithm first constructs a matrix M which represents possible vertex assignment. After making a possible vertex assignment, the algorithm then refines M by using a necessary and sufficient condition for graph isomorphism. This compound condition is based on the adjacency relations of the vertices. If the graphs are isomorphic, then M, after possible backtracking to reassign vertices, is refined to a matrix which specifies an isomorphism between G_A and G_B. The number of vertices and arcs of G_A and G_B are given by pa, qa and pb, qb. The algorithm uses a pb-bit binary vector $\{F_1,\ldots,F_i,\ldots,F_{pb}\}$ to record which columns have been used at an intermediate state of computation: $F_i = 1$ if the i^{th} column has been used. The algorithm also uses a vector $\{H_1,\ldots,H_d,\ldots,H_{pa}\}$ to record which column has been used at which depth: $H_d = k$ if the k^{th} column has been selected at depth d.

Step 0. Construct M^o according to

$$M^o_{ij} = \begin{cases} 1, \text{ if the indegree of the } i^{th} \text{ point in } G_A \text{ is the same as the indegree of the } j^{th} \text{ point in } G_B \text{ and the outdegree of the } i^{th} \text{ point in } G_A \text{ is the same as the outdegree of the } j^{th} \text{ point in } G_B, \\ 0, \text{ otherwise;} \end{cases}$$

Step 1. $M := M^0$; $d := 1$; $H_1 := 0$

for all $i := 1,\ldots,pb$ set $F_i := 0$;

refine M; if exit FAIL then terminate algorithm;

Step 2. If there is no value of j such that $m_{dj} = 1$ and $f_i = 0$

then go to Step 7;

$M_d := M$;

if $d = 1$ then $k := H_1$ else $k := 0$;

Step 3. $k := k + 1$;

if $m_{dk} = 0$ or $f_k = 1$ then go to Step 3;

for all $j \neq k$ set $m_{dj} := 0$;

refine M; if exit FAIL then go to Step 5;

Step 4. If $d < pa$ then go to Step 6 else give output to indicate

that an isomorphism has been found;

Step 5. $M := M_d$;

if there is no $j > k$ such that $m_{dj} = 1$ and $f_i = 0$ then go

to Step 7;

go to Step 3;

Step 6. $H_d := k$; $F_k := 1$; $d := d + 1$;

go to Step 2;

Step 7. If $d = 1$ then terminate algorithm;

$d := d - 1$; $k := H_d$; $F_k := 0$;

go to Step 5;

Refine M. $m_{ij} = 1$ is changed to $m_{ij} = 0$ unless

$$\underset{1 \leq x \leq pa}{(\forall x)} \left((a_{ix} = 1) \Rightarrow \underset{1 \leq y \leq pb}{(\exists y)} (m_{xy} \cdot b_{jy} = 1)\right)$$

and

$$\underset{1 \le x \le pa}{(\forall x)} \left((a_{xi} = 1) \Rightarrow \underset{1 \le y \le pb}{(\exists y)} (m_{xy} \cdot b_{yj} = 1) \right)$$

where $[a_{ij}]$ and $[b_{ij}]$ are the adjacency matrices for graphs G_A and G_B.

F.2 PL/I Source Listing

```
ULLMANN: PROCEDURE OPTIONS(MAIN);
/*                                                                    */
/*    THE MAIN PROCEDURE ULLMANN IMPLEMENTS ALGORITHM 6.  THE         */
/* STATEMENT NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF          */
/* ALGORITHM 6.  IN STEP 0, THE PROCEDURE CALLS UGENGR TO CREATE      */
/* THE ADJACENCY MATRICES FOR GRAPHS A AND B.  CALCMO IS THEN         */
/* CALLED TO CONSTRUCT THE INITIAL MATRIX M.  IN STEPS 1-7,           */
/* ALGORITHM 6 IS IMPLEMENTED.  THE INTERNAL PROCEDURE REFINE IMPLE-  */
/* MENTS THE REFINEMENT CONDITIONS OF ULLMANN.  THIS PROCEDURE IS     */
/* CALLED TO REFINE M AFTER EACH VERTEX ASSIGNMENT.  STEP 4 OF        */
/* ALORITHM 6 IS MODIFIED SO THAT IF M IS LEFT UNCHANGED BY THE       */
/* REFINEMENT PROCEDURE AND M IS SUCH THAT EACH ROW AND EACH COLUMN   */
/* CONTAINS EXACTLY ONE 1, THEN THE THE ALGORITHM HAS SUCCEEDED IN    */
/* DETERMINING GRAPH A IS ISOMORPHIC TO GRAPH B.  THE ISOMORPHISM IS  */
/* DEFINED BY THE MATRIX M.  THE REMAINDER OF THE PROCEDURE           */
/* GATHERS PERFORMANCE INFORMATION.  THE ASSEMBLY LANGUAGE ROUTINE    */
/* ASMTME IS USED TO OBTAIN EXECUTION TIMES FOR SECTIONS OF A         */
/* PROCEDURE.                                                         */
/*    THE PROCEDURE CAN HANDLE TWO GRAPHS HAVING UP TO 64 VERTICES.   */
/* IF MORE VERTICES ARE NEEDED, THEN THE DIMENSIONS OF A AND B MUST   */
/* BE INCREASED.  THE VARIABLE AND ARRAY NAMES CORRESPOND TO THOSE    */
/* USED BY ALGORITHM 6.                                               */
/*                                                                    */
  DECLARE (A(64,64),B(64,64),M(64,64)) BIT(1) EXT,
         (NA,NB,NPA,NPB) FIXED BIN(15) EXT,
         BCOL(64,64) BIT(1) EXT,MD(64,64,64) BIT(1);
  DECLARE (TMCALCM(25),NO_GRAPH_PRS) FIXED BIN(31) EXT;
  DECLARE (TMULLM(25),NOREF(25),NOBACK(25),MNULLM,MNCALCM,MNREF,
         MNBACK,NOGR,TOTAL) FIXED BIN(31),(MEULLM,MECALCM,MEREF,
         MEBACK,METOT) FLOAT DEC(16);
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
/*                                                                    */
/* READ IN TWO GRAPHS AND CREATE CORRESPONDING ADJACENCY MATRICES     */
/*                                                                    */
         GET FILE(GRAPH) LIST(NGPRS,NA,NB,NPA,NPB);
         GET LIST(NO_GRAPH_PRS);
         MNCALCM=0; MNULLM=0; MNREF=0; MNBACK=0;
         TMCALCM=0; TMULLM=0; NOREF=0; NOBACK=0; NOGR=NO_GRAPH_PRS;
  STEP0: CALL UGENGR;
         IF NA ¬= NB | NPA ¬= NPB THEN GO TO NO_ISOMORPHISM;
         BEGIN;
         DECLARE F(NB) BIT(1),
          (SUCCEED,FAIL) BIT(1),(H(HA),D,NOWDS) FIXED BIN(15);
```

```
/*                                                                    */
/* CALL CALCMO TO CONSTRUCT INITIAL MATRIX M                          */
/*                                                                    */
        CALL CALCMO; NOWDS=CEIL(NA/32);
/*                                                                    */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                           */
/*                                                                    */
        ITME= -1; CALL ASMTME(ITME);
/*                                                                    */
/*          THE FOLLOWING CODE REPRESENTS ALGORITHM 6                 */
/*                                                                    */
  STEP1: D=1; H(1)=0; F='0'B;
         CALL REFINE(B,BCOL,M);
         IF FAIL THEN GO TO NO_ISOMORPHISM;
  STEP2: DO I=1 TO NB;
          IF M(D,I) &  ¬F(I) THEN GO TO STEP2_1;
         END;
         GO TO STEP7;
  STEP2_1:
         MD(D,*,*)=M;
         IF D=1 THEN K=H(1); ELSE K=0;
  STEP3: K=K + 1;
         IF( ¬M(D,K) | F(K))THEN GO TO STEP3;
         DO J=1 TO NB; IF J ¬= K THEN M(D,J)='0'B; END;
         CALL REFINE(B,BCOL,M);
         IF FAIL THEN GO TO STEP5;
  STEP4: IF SUCCEED
          THEN DO;
               PUT SKIP(4) EDIT
                ('THE FOLLOWING DEFINES AN ISOMORPHISM FROM A TO B')
                (A); PUT SKIP LIST(' ');
                PUT EDIT((I DO I=1 TO NA)) (F(3));
                PUT SKIP LIST(' ');
                DO I=1 TO NA; DO J=1 TO NA;
                 IF M(I,J) THEN DO; PUT EDIT(J) (F(3)); GO TO IEND;
                 END; END; IEND: END;
               GO TO CHECK_NO_GRAPHS;
               END;
          ELSE GO TO STEP6;
  STEP5: M=MD(D,*,*);
         I=K+1;
         DO J=I TO NB;
         IF (M(D,J) & ¬F(J)) THEN GO TO STEP5_1;
         END;
         GO TO STEP7;
  STEP5_1: GO TO STEP3;
  STEP6: H(D)=K; F(K)='1'B; D=D + 1;
         GO TO STEP2;
  STEP7: IF D=1 THEN GO TO NO_ISOMORPHISM;
         D=D - 1; K=H(D); F(K)='0'B;
         NOBACK(NO_GRAPH_PRS)=NOBACK(NO_GRAPH_PRS) + 1;
         GO TO STEP5;                                                 */
 /*
```

```
/*                                                                    */
REFINE: PROCEDURE(B,BCOL,M);
/*                                                                    */
/*    THE PROCEDURE REFINE IMPLEMENTS THE REFINEMENT CONDITIONS       */
/* USED BY ALGORITHM 6.  EACH ROW OF M, EACH ROW OF B, AND EACH       */
/* COLUMN OF B IS STORED IN CEIL(N/32) WORDS.  IN ORDER TO EXPLOIT    */
/* THE LIMITED PARALLELISM OF THE IBM 370/158, THE REFINEMENT         */
/* CONDITIONS ARE IMPLEMENTED BY ORING THE APPROPRIATE ROW OF M       */
/* WITH THE APPROPRIATE ROW OR COLUMN OF B.  THE PROCEDURE RETURNS    */
/* TO THE MAIN PROCEDURE ULLMANN WHEN ONE OF THE FOLLOWING OCCURS:    */
/* NO MORE ELEMENTS OF M ARE CHANGED; OR A ROW OF M BECOMES ALL       */
/* ZEROS, FAIL; OR EACH ROW AND EACH COLUMN OF M CONTAINS EXACTLY     */
/* ONE 1, SUCCEED.                                                    */
/*                                                                    */
  DECLARE B(64,64) BIT(1),ROWB(64,2) BIT(32) DEF B,
         M(64,64) BIT(1),ROWM(64,2) BIT(32) DEF M,
         BCOL(64,64) BIT(1),COLB(64,2) BIT(32) DEF BCOL,
         CHANGE BIT(1),COL(NA) BIT(1),STRCOL BIT(NA) DEF COL,
         ZEROS FIXED BIN(15),BINZERO BIT(32) INIT((32)'0'B) STATIC;
         NOREF(NO_GRAPH_PRS)=NOREF(NO_GRAPH_PRS) + 1;
/*                                                                    */
         FAIL='0'B; CHANGE='1'B;
         DO WHILE(CHANGE);
          CHANGE='0'B; COL='0'B; SUCCEED='1'B;
          DO I=1 TO NA;
           ZEROS=0;
           DO J=1 TO NA;
           IF ¬M(I,J) THEN GO TO NEXT_M;
           DO IX=1 TO NA;
            IF A(I,IX) THEN DO;
                            DO IY=1 TO NOWDS;
                            IF (ROWM(IX,IY) & ROWB(J,IY)) ¬= BINZERO
                              THEN GO TO COND1_OK;
                            END;
                            M(I,J)='0'B; CHANGE='1'B; GO TO NEXT_M;
                            END;
  COND1_OK: IF A(IX,I) THEN DO;
                            DO IY=1 TO NOWDS;
                            IF (ROWM(IX,IY) & COLB(J,IY)) ¬= BINZERO
                              THEN GO TO COND2_OK;
                            END;
                            M(I,J)='0'B; CHANGE='1'B; GO TO NEXT_M;
                            END;
  COND2_OK: END;
  NEXT_M:  IF ¬M(I,J)
            THEN ZEROS=ZEROS + 1;
            ELSE COL(J)='1'B;
           END;
           IF ZEROS=NB THEN DO; FAIL='1'B; RETURN; END;
           IF ZEROS ¬= NB - 1 THEN SUCCEED='0'B;
          END;
         END;
         IF SUCCEED THEN IF ¬STRCOL='0'B
```

```
                              THEN ;
                               ELSE SUCCEED='0'B;
END REFINE;
/*                                                                    */
/*                                                                    */
        END;
  NO_ISOMORPHISM:
        PUT SKIP(2) EDIT('NO ISOMORPHISM EXISTS FOR GIVEN GRAPHS')
         (A);
        IF ITME= -1 THEN; ELSE GO TO TOTALING;
/*                                                                    */
/*            END OF ALGORITHM 6                                      */
/*                                                                    */
/* THE REMAINDER OF THE PROCEDURE GATHERS PERFORMANCE INFORMATION     */
/*                                                                    */
  CHECK_NO_GRAPHS:
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME=1; CALL ASMTME(ITME);
        TMULLM(NO_GRAPH_PRS)=ITME;
  TOTALING:
        MNCALCM=MNCALCM + TMCALCM(NO_GRAPH_PRS);
        MNULLM=MNULLM + TMULLM(NO_GRAPH_PRS);
        MNBACK=MNBACK + NOBACK(NO_GRAPH_PRS);
        MNREF=MNREF + NOREF(NO_GRAPH_PRS);
        NO_GRAPH_PRS=NO_GRAPH_PRS - 1;
        IF NO_GRAPH_PRS ^= 0
         THEN GO TO STEP0;
         ELSE DO; PUT PAGE EDIT('ANALYSIS OF RUN USING ',NOGR,
              ' GRAPH PAIRS OF ',NA,' VERTICES EACH') (A,F(2),A,F(2),
              A); PUT SKIP(2) EDIT('NUMBER','BACKTRACK','REFINE',
              'CALCMO','ULLMANN','TOTAL') (COL(1),A,COL(15),A,COL(34),
              A,COL(56),A,COL(75),A,COL(95),A); J=1;
              DO I=NOGR TO 1 BY -1;
              TOTAL=TMCALCM(I) + TMULLM(I);
              PUT SKIP EDIT(J,NOBACK(I),NOREF(I),TMCALCM(I),TMULLM(I),
              TOTAL) (F(2),COL(10),F(11),COL(30),F(11),COL(50),F(11),
              COL(70),F(11),COL(90),F(11));
              J=J + 1;
              END;
              TOTAL=MNCALCM + MNULLM;
              PUT SKIP(2) EDIT
               ('TOTAL',MNBACK,MNREF,MNCALCM,MNULLM,TOTAL)
               (A,COL(10),F(11),COL(30),F(11),COL(50),F(11),
               COL(70),F(11),COL(90),F(11));
              MEBACK=FLOAT(MNBACK)/NOGR; MEREF=FLOAT(MNREF)/NOGR;
              MECALCM=FLOAT(MNCALCM)/NOGR; MEULLM=FLOAT(MNULLM)/NOGR;
              METOT=FLOAT(TOTAL)/NOGR;
              PUT SKIP(2) EDIT
               ('MEAN',MEBACK,MEREF,MECALCM,MEULLM,METOT)
               (A,COL(10),F(15,3),COL(30),F(15,3),COL(50),
              F(15,3),COL(70),F(15,3),COL(90),F(15,3));
```

```
            PUT SKIP(2) EDIT('***EACH ABOVE UNIT=26.04166 X 10 -6',
             ' SEC.') (A,A);
            END;
END ULLMANN;


UGENGR: PROCEDURE;                                                    */
/*                                                                    */
/*    THE PROCEDURE UGENGR BUILDS ADJACENCY MATRICES FOR GRAPHS A     */
/* AND B.  ADJACENCY LISTS FOR BOTH GRAPHS ARE PRINTED.               */
/*                                                                    */
  DECLARE (A(64,64),B(64,64),BCOL(64,64)) BIT(1) EXT,
         (NA,NB,NPA,NPB) FIXED BIN(15) EXT;
/*                                                                    */
         A='0'B; B='0'B; BCOL='0'B;
/*                                                                    */
/* BUILD ADJACENCY MATRIX AND PRINT ADJACENCY LIST FOR GRAPH A        */
/*                                                                    */
         PUT PAGE EDIT('ADJACENCY LIST FOR GRAPH A') (A);
         LAST=0;
         DO I=1 TO NPA;
          GET FILE(GRAPH) LIST (II,J); A(II,J)='1'B;
          IF II ^= LAST
           THEN DO; PUT SKIP EDIT(II) (F(2)); LAST=II; END;
          PUT EDIT(J) (F(3));
         END;
         IF II ^= NA THEN PUT SKIP EDIT(NA) (F(2));
/*                                                                    */
/* BUILD ADJACENCY MATRIX AND PRINT ADJACENCY LIST FOR GRAPH B        */
/*                                                                    */
         PUT PAGE EDIT('ADJACENCY LIST FOR GRAPH B') (A);
         LAST=0;
         DO I=1 TO NPB;
          GET FILE(GRAPH) LIST (II,J); B(II,J)='1'B;
          BCOL(J,II)='1'B;
          IF II ^= LAST
           THEN DO; PUT SKIP EDIT(II) (F(2)); LAST=II; END;
          PUT EDIT(J) (F(3));
         END;
         IF II ^= NB THEN PUT SKIP EDIT(NB) (F(2));
END UGENGR;


CALCMO: PROCEDURE;                                                    */
/*                                                                    */
/*    THE PROCEDURE CALCMO CONSTRUCTS THE INITIAL MATRIX M            */
/* ACCORDING TO:  M(I,J)=1, IF THE INDEGREE AND OUTDEGREE OF VERTEX   */
/* I IS EQUAL TO THE INDEGREE AND OUTDEGREE OF VERTEX J, OTHERWISE,   */
/* M(I,J)=0.                                                          */
/*                                                                    */
  DECLARE (A(64,64),B(64,64),M(64,64)) BIT(1) EXT,
         (NA,NB) FIXED BIN(15) EXT;
  DECLARE (TMCALCM(25),NO_GRAPH_PRS) FIXED BIN(31) EXT;
```

```
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
/*                                                                    */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                           */
/*                                                                    */
        ITME= -1; CALL ASMTME(ITME);
/*                                                                    */
/*           THE FOLLOWING CODE CONSTRUCTS INITIAL MATRIX M           */
/*                                                                    */
        BEGIN;
        DECLARE (DAI(NA),DAO(NA),DBI(NA),DBO(NA)) FIXED BIN(15);
        M='0'B;
        DAI=0; DAO=0; DBI=0; DBO=0;
        DO I=1 TO NA;
         DO J=1 TO NA;
         IF A(I,J)
          THEN DO; DAO(I)=DAO(I) + 1; DAI(J)=DAI(J) + 1; END;
         IF B(I,J)
          THEN DO; DBO(I)=DBO(I) + 1; DBI(J)=DBI(J) + 1; END;
         END;
        END;
        DO I=1 TO NA;
         DO J=1 TO NA;
         IF DAI(I)=DBI(J) & DAO(I)=DBO(J)
          THEN M(I,J)='1'B;
         END;
        END;
        END;
/*                                                                    */
/*           END OF INITIAL M CONSTRUCTION                            */
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME=1; CALL ASMTME(ITME);
        TMCALCM(NO_GRAPH_PRS)=ITME;
END CALCMO;
```

APPENDIX G

SCHMIDT AND DRUFFEL'S BACKTRACKING ALGORITHM AND PL/I IMPLEMENTATION

G.1 Algorithms

G.1.2 Algorithm 7

Algorithm 7 is Floyd's shortest path algorithm. Initially m[i,j] is the length of a direct path from vertex i to vertex j i.e., m[i,j] = 1 if there is an arc from vertex i to vertex j. If no arc exists, then m[i,j] is initially 10^{10}. At the completion of the algorithm, m[i,j] is the length of the shortest path from vertex i to vertex j. If none exists m[i,j] is 10^{10}. Matrix M contains the shortest path in formation.

```
procedure shortest path (m,n); value n; integer n; array m
begin
integer i,j,k; real inf, s; inf := 10^10
for i := 1 step 1 until n do
for j := 1 step 1 until n do
if m[j,i] < inf then
for k := 1 step 1 until n do
if m[i,k] < inf then
begin s := m[j,k] + m[i,k];
if s < m[j,k] then m[j,k] := s
end
end shortest path
```

G.1.2 Algorithm 8

Algorithm 8 generates the initial partition for the sets of vertices of graphs G^1 and G^2. First, the row and column characteristic matrices XR^1 and XC^1 for G^1, and XR^2 and XC^2 for G^2, are constructed from the distance matrices $D^1 = [d^1_{ij}]$ and $D^2 = [d^2_{ij}]$. Next, the characteristic matrices X^1 for G^1 and X^2 for G^2 are respectively formed by composing XR^1 with XC^1 and XR^2 with XC^2. The initial partition which is represented by the class vectors $C^1_0 = [c^1_i]$ and $C^2_0 = [c^2_i]$ is then generated by assigning the same class to all vertices having identical X^1 and X^2 rows.

Step 1. Set $c^1_i = 0$ $(1 \leq i \leq N)$.

Step 2. Compute the row and column characteristic matrices, XR^1 and XC^1 for G^1, and XR^2 and XC^2 for G^2.

$$XR^1 = \{xr^1_{im} \mid (1 \leq i \leq N) \wedge (1 \leq m \leq N-1) \wedge xr^1_{im} = [\{d^1_{ij} \mid (1 \leq j \leq N) \wedge (d_{ij} = m)\}]\}.$$

$$XR^2 = \{xr^2_{sm} \mid (1 \leq s \leq N) \wedge (1 \leq m \leq N-1) \wedge xr^2_{sm} = [\{d^2_{st} \mid (1 \leq t \leq N) \wedge (d^2_{st} = m)\}]\}.$$

$$XC^1 = \{xc^1_{jm} \mid (1 \leq j \leq N) \wedge (1 \leq m \leq N-1) \wedge xc^1_{jm} = [\{d^1_{ij} \mid (1 \leq i \leq N) \wedge (d^1_{ij} = m)\}]\}.$$

$$XC^2 = \{xc^2_{tm} \mid (1 \leq t \leq N) \wedge (1 \leq m \leq N-1) \wedge xc^2_{tm} = [\{d^2_{st} \mid (1 \leq s \leq N) \wedge (d^2_{st} = m)\}]\}.$$

Step 3. Compose XR^1 with XC^1 to form X^1 and XR^2 with XC^2 to form X^2.

Step 4. CLASS = 0.

Step 5. CLASS = CLASS + 1.

Step 6. Select some integer $k \in \{1,2,\ldots,N\}$ such that $c^1_k = 0$. If none, stop.

Step 7. Determine two sets of integers:

$$W^1 = \{i \mid x^1_{im} = x^1_{km} \ (1 \leq m \leq N - 1)\}.$$

$$W^2 = \{i \mid x^2_{im} = x^1_{km} \ (1 \leq m \leq N - 1)\}.$$

Step 8. Make class assignments:

c^1_i = CLASS, $i \notin W^1$. c^2_i = CLASS, $i \notin W^2$.

Step 9. Go to Step 5.

G.1.3 Algorithm 9

Algorithm 9, the backtracking algorithm, tests graphs G^1 and G^2 for isomorphism. The algorithm selects possible vertex assignments between the two graphs and checks these assignments for consistency. If the assignment is not consistent then the algorithm backtracks to try another vertex assignment. Since a new class vectors, C^1 and C^2, and class count vectors, K^1 and K^2, are generated at each level of the vertex assignment tree, an index t is added to these vectors. Thus, C^1_t refers to the class vector for graph G^1 generated at level t. The class count vector K is defined such that the element k_i is the number of vertices in class i. D^1 and D^2 are the distance matrices for graphs G^1 and G^2.

Step 1. Set t = 0. Set $p_i = 0$ $(0 \leq i \leq N)$. (p_t is the vertex in Graph G^1 chosen at level t.)

Step 2. Let C^1_0 and C^2_0 be the class vectors from Algorithm 8, and let K^1_t and K^2_t be the class count vectors.

Step 3. The t = N, conclude that the graphs are isomorphic, present the mapping, and STOP.

Step 4. Set $i = p_t$.

Step 5. If $i \neq 0$, go to Step 7.

Step 6. Choose some integer i for which v_i^1 has not been mapped at a lower level. If a c_{it}^1 exists with a unique unmapped vertex, choose i. Set $p_t = i$.

Step 7. Choose some r such that $c_{it}^1 = c_{rt}^2$ and for which there has been no mapping yet chosen for v_r^2.

Step 8. If such an r exists, go to Step 10.

Step 9. Decrement t. If $t < 0$, conclude that the graph have no isomorphism; otherwise, go to Step 7.

Step 10. Compose C_t^1 with row i of D^1 to yield $\hat{C}^1$. Compose C_t^2 with row r of D^2 to yield $\hat{C}^2$. Generate $\tilde{C}^1$ and $\tilde{C}^2$ by substituting a unique integer for each unique term of $\hat{C}^1$ and $\hat{C}^2$. Compose $\tilde{C}^1$ with column i of D^1 to yield $\dot{C}^1$. Compose $\tilde{C}^2$ with column r of D^2 to yield $\dot{C}^2$. Generate C_{t+1}^1 and C_{t+1}^2 by substituting a unique integer for each unique term of $\dot{C}^1$ and $\dot{C}^2$.

Step 11. Compute the class count vectors K_{t+1}^1, K_{t+1}^2.

Step 12. If $K_{t+1}^1 \neq K_{t+1}^2$, go to Step 7.

Step 13. Increment t.

Step 14. Go to Step 3.

G.2 PL/I Source Listing

```
DRUFFEL: PROCEDURE OPTIONS(MAIN);                                    */
/*                                                                   */
/*    THE MAIN PROCEDURE DRUFFEL IMPLEMENTS ALGORITHM 9.  THE        */
/* STATEMENT NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF         */
/* ALGORITHM 9.  IN STEP 0, THE PROCEDURE CALLS DGENGR TO CONSTRUCT */
/* THE ADJACENCY MATRICES FOR GRAPHS G1 AND G2.  DIST IS THEN       */
/* CALLED TO GENERATE THE CORRESPONDING DISTANCE MATRICES USING     */
/* FLOYD'S ALGORITHM 7.  IN STEPS 1-14, ALGORITHM 9 IS IMPLEMENTED. */
/* IN STEP 1, THE PROCEDURE DALG1 (ALGORITHM 8) IS CALLED TO        */
/* CONSTRUCT THE INITIAL PARTITION FOR THE SETS OF VERTICES.  THE   */
/* INTERNAL PROCEDURE CHOOSE IS CALLED IN STEP 6 TO CHOOSE A VERTEX */
/* IN THE SMALLEST CLASS SUCH THAT THE VERTEX HAS NOT BEEN PREVIOUS-*/
```

```
/* LY ASSIGNED.  THE REMAINDER OF THE PROCEDURE GATHERS PERFORMANCE */
/* INFORMATION.  THE ASSEMBLY LANGUAGE ROUTINE ASMTME IS USED TO    */
/* OBTAIN THE EXECUTION TIME FOR SECTIONS OF A PROCEDURE.           */
/*    THE PROCEDURE CAN HANDLE TWO GRAPHS OF UP TO 64 VERTICES.     */
/* IF MORE VERTICES ARE NEEDED, THEN THE DIMENSIONS OF ALL ARRAY    */
/* NAMES DIMENSIONED 64 MUST BE CHANGED.  VARIABLE AND ARRAY NAMES  */
/* ROUGHLY CORRESPOND TO THOSE USED BY ALGORITHM 9.                 */
/*                                                                  */
  DECLARE (G1(64,64),G2(64,64), NG1,NG2,NPG1,NPG2) FIXED BIN(15) EXT,
         (K1,INDEX1(0:64,64),LIST1(0:64,64)) FIXED BIN(15) EXT,
         (K2,INDEX2(0:64,64),LIST2(0:64,64)) FIXED BIN(15) EXT,
         (C1(0:64,64),C2(0:64,64)) FIXED BIN(15) EXT,
         NO_ISO BIT(1) EXT;
  DECLARE (T,VI,LAST,C1IT,CLASS,R) FIXED BIN(15);
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
  DECLARE (TMDIST(25),TMDALG1(25),TMDRUF(25),NO_GRAPH_PRS)
         FIXED BIN(31) EXT;
  DECLARE (BACKTR(25),MNBACK,MNDIST,MNDALG1,MNDRUF,NOGR,TOTAL)
         FIXED BIN(31), (MEBACK,MEDIST,MEDALG1,MEDRUF,METOT) FLOAT;
/*                                                                  */
/* READ IN TWO GRAPHS AND CREATE ADJACENCY MATRICES G1 AND G2       */
/*                                                                  */
         GET FILE(GRAPH) LIST(NGPRS,NG1,NG2,NPG1,NPG2);
         GET LIST(NO_GRAPH_PRS);
         BACKTR=0; TMDIST=0; TMDALG1=0; TMDRUF=0;
         NOGR=NO_GRAPH_PRS; MNBACK=0; MNDIST=0; MNDALG1=0; MNDRUF=0;
  STEP0: CALL DGENGR;
         IF NG1 ¬= NG2 | NPG1 ¬= NPG2
           THEN GO TO NO_ISOMORPHISM;
/*                                                                  */
/* CALL DIST (FLOYD'S ALGORITHM 7) TO CONSTRUCT DISTANCE MATRICES   */
/*                                                                  */
         PUT PAGE EDIT('DISTANCE MATRIX FOR G1') (A);
         CALL DIST(G1,NG1);
         PUT PAGE EDIT('DISTANCE MATRIX FOR G2') (A);
         CALL DIST(G2,NG2);
/*                                                                  */
/*           THE FOLLOWING CODE REPRESENTS ALGORITHM 9              */
/*                                                                  */
/* CALL DALG1 (ALGORITHM 8) TO CREATE INITIAL PARTITION FOR SETS    */
/* OF VERTICES OF G1 AND G2                                         */
  STEP1: CALL DALG1;
         IF NO_ISO THEN GO TO NO_ISOMORPHISM;
/*                                                                  */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                         */
         ITME= -1; CALL ASMTME(ITME);
  STEP2: BEGIN;
         DECLARE (P(0:NG1),MAP1(NG1)) FIXED BIN(15), MAP2(NG2) BIT(1);
         T=0; P=0; MAP1=0; MAP2='0'B;
  STEP3: IF T=NG1
           THEN DO; PUT SKIP(4) EDIT
                ('THE FOLLOWING DEFINES AND ISOMORPHISM FROM G1 TO G2')
                (A); PUT SKIP EDIT((I DO I=1 TO NG1)) ((64)F(3));
```

```
              PUT SKIP EDIT((MAP1(I) DO I=1 TO NG1)) ((64)F(3));
              GO TO CHECK_NO_GRAPHS;
              END;
STEP4: I=P(T);
STEP5: IF I ¬= 0
         THEN DO; VI=I; GO TO STEP7; END;
STEP6: CALL CHOOSE; P(T)=VI;
STEP7: C1IT=C1(T,VI); R=INDEX2(T,C1IT);
       IF MAP2(R)
         THEN DO;
STEP7_1:       DO WHILE(LIST2(T,R) ¬= 0);
               R=LIST2(T,R);
STEP8:         IF ¬MAP2(R) THEN GO TO STEP10;
               END;
               END;
         ELSE GO TO STEP10;
STEP9: T=T - 1;
       BACKTR(NO_GRAPH_PRS)=BACKTR(NO_GRAPH_PRS) + 1;
       IF T < 0 THEN GO TO NO_ISOMORPHISM;
              ELSE DO; VI=P(T); R=MAP1(VI); MAP2(R)='0'B;
                       GO TO STEP7_1;
                   END;
STEP10: CLASS=0; II=0;
       DO I=1 TO NG1;
          C1(T+1,I)=0; INDEX1(T+1,I)=0; LIST1(T+1,I)=0;
          C2(T+1,I)=0; INDEX2(T+1,I)=0; LIST2(T+1,I)=0;
       END;
STEP10_1: CLASS=CLASS + 1; II=II + 1;
       DO K=II TO NG1;
        IF C1(T+1,K)=0 THEN GO TO STEP10_2;
       END;
       GO TO STEP13;
STEP10_2: C1(T+1,K)=CLASS; K1=1;
       INDEX1(T+1,CLASS)=K; LAST=K;
       DO I=K+1 TO NG1;
        IF C1(T+1,I)=0
         THEN IF C1(T,K)=C1(T,I) & G1(VI,K)=G1(VI,I)
                 & G1(K,VI)=G1(I,VI)
                 THEN DO; C1(T+1,I)=CLASS; K1=K1+1;
                          LIST1(T+1,LAST)=I; LAST=I;
                      END;
       END;
       K2=0;
       DO I=1 TO NG2;
        IF C2(T+1,I)=0
         THEN IF C1(T,K)=C2(T,I) & G1(VI,K)=G2(R,I)
                 & G1(K,VI)=G2(I,R)
                 THEN DO; C2(T+1,I)=CLASS; K2=K2+1;
                          IF K2=1
                           THEN INDEX2(T+1,CLASS)=I;
                           ELSE LIST2(T+1,LAST)=I;
                          LAST=I;
                      END;
```

```
        END;
  STEP11_12: IF K1 ^= K2
         THEN GO TO STEP7_1;
         ELSE GO TO STEP10_1;
  STEP13: MAP1(VI)=R; MAP2(R)='1'B; T=T + 1;
  STEP14: GO TO STEP3;
/*                                                                    */
/*                                                                    */
CHOOSE: PROCEDURE;
/*                                                                    */
/*    THE PROCEDURE CHOOSE IMPLEMENTS THE STRATEGY USED IN CHOOSING */
/* A VERTEX OF G1 TO ASSIGN TO A VERTEX OF G2.  THE STRATEGY         */
/* CONSISTS OF CHOOSING A VERTEX IN THE SMALLEST CLASS SUCH THAT     */
/* THE VERTEX HAS NOT BEEN PREVIOUSLY ASSIGNED.  THE STRATEGY HAS    */
/* THE EFFECT OF REDUCING THE BREADTH OF SEARCH, BUT POSSIBLY        */
/* PERMITS A GREATER DEPTH.                                          */
/*                                                                    */
  DECLARE (SIZE,LINK) FIXED BIN(15),CLASS_STATUS(NG1) BIT(1);
/*                                                                    */
         SIZE=1; J=1; CLASS_STATUS='0'B;
         IF INDEX1(T,2)=0
           THEN DO; LINK=1; GO TO LINK_V; END;
  FIND_SIZE:
         DO I=J TO NG1;
          IF CLASS_STATUS(C1(T,I)) THEN GO TO NXT_V;
          LINK=C1(T,I);
          DO II=1 TO SIZE;
           IF LINK=0
             THEN DO; CLASS_STATUS(C1(T,I))='1'B;
                      GO TO NXT_V;
                  END;
           LINK=LIST1(T,LINK);
          END;
          IF LINK=0 THEN GO TO FIND_V;
  NXT_V: END;
         SIZE=SIZE + 1; J=1; GO TO FIND_SIZE;
  FIND_V: LINK=I;
  LINK_V: IF MAP1(LINK)=0
           THEN DO; VI=LINK; RETURN; END;
           ELSE DO; LINK=LIST1(T,LINK);
                    IF LINK=0
                     THEN DO; J=I+1; CLASS_STATUS(C1(T,I))='1'B;
                              GO TO FIND_SIZE;
                          END;
                     ELSE GO TO LINK_V;
                END;
END CHOOSE;
/*                                                                    */
/*                                                                    */
         END;
  NO_ISOMORPHISM:
         PUT SKIP(2) EDIT('NO ISOMORPHISM EXISTS FOR GIVEN GRAPHS')
          (A);
```

```
        IF ITME= -1 THEN; ELSE GO TO TOTALING;
/*                                                                    */
/*            END OF ALGORITHM 9                                      */
/*                                                                    */
/* THE REMAINDER OF THE PROCEDURE GATHERS PERFORMANCE INFORMATION     */
/*                                                                    */
  CHECK_NO_GRAPHS:
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME=1; CALL ASMTME(ITME);
        TMDRUF(NO_GRAPH_PRS)=ITME;
  TOTALING:
        MNDIST=MNDIST + TMDIST(NO_GRAPH_PRS);
        MNDALG1=MNDALG1 + TMDALG1(NO_GRAPH_PRS);
        MNBACK=MNBACK + BACKTR(NO_GRAPH_PRS);
        MNDRUF=MNDRUF + TMDRUF(NO_GRAPH_PRS);
        NO_GRAPH_PRS=NO_GRAPH_PRS - 1;
        IF NO_GRAPH_PRS >= 0
         THEN GO TO STEP0;
         ELSE DO; PUT PAGE EDIT('ANALYSIS OF RUN USING ',NOGR,
              ' GRAPH PAIRS OF ',NG1,' VERTICES EACH') (A,F(2),A,F(2),
              A); PUT SKIP(2) EDIT('NUMBER','BACKTRACK','DIST','DALG1'
              ,'DRUFFEL','TOTAL') (COL(1),A,COL(15),A,COL(34),A,
              COL(56),A,COL(75),A,COL(95),A); J=1;
              DO I=NOGR TO 1 BY -1;
              TOTAL=TMDIST(I) + TMDALG1(I) + TMDRUF(I);
              PUT SKIP EDIT(J,BACKTR(I),TMDIST(I),TMDALG1(I),TMDRUF(I)
              ,TOTAL) (F(2),COL(10),F(11),COL(30),F(11),COL(50),F(11)
              ,COL(70),F(11),COL(90),F(11));
              J=J + 1;

              END;
              TOTAL=MNDIST + MNDALG1 + MNDRUF;
              PUT SKIP(2) EDIT
                        ('TOTAL',MNBACK,MNDIST,MNDALG1,MNDRUF,TOTAL)
              (A,COL(10),F(11),COL(30),F(11),COL(50),F(11),
              COL(70),F(11),COL(90),F(11));
              MEBACK=FLOAT(MNBACK)/NOGR; MEDIST=FLOAT(MNDIST)/NOGR;
              MEDALG1=FLOAT(MNDALG1)/NOGR; MEDRUF=FLOAT(MNDRUF)/NOGR;
              METOT=FLOAT(TOTAL)/NOGR;
              PUT SKIP(2) EDIT
                        ('MEAN',MEBACK,MEDIST,MEDALG1,MEDRUF,
              METOT) (A,COL(10),F(15,3),COL(30),F(15,3),COL(50),
              F(15,3),COL(70),F(15,3),COL(90),F(15,3));
              PUT SKIP(2) EDIT('***EACH ABOVE UNIT=26.04166 X 10 -6',
              ' SEC.') (A,A);
              END;
END DRUFFEL;


DGENGR: PROCEDURE;
/*                                                                    */
```

```
/*    THE PROCEDURE DGENGR BUILDS THE ADJACENCY MATRICES FOR GRAPHS */
/* G1 AND G2.  ADJACENCY LISTS FOR BOTH GRAPHS ARE PRINTED.        */
/*                                                                 */
  DECLARE (G1(64,64),G2(64,64),NG1,NG2,NPG1,NPG2) FIXED BIN(15) EXT;
/*                                                                 */
        G1=0; G2=0;
/*                                                                 */
/* BUILD ADJACENCY MATRIX AND PRINT ADJACENCY LIST FOR GRAPH G1    */
/*                                                                 */
        PUT PAGE EDIT('ADJACENCY LIST FOR GRAPH G1') (A);
        LAST=0;
        DO I=1 TO NPG1;
         GET FILE(GRAPH) LIST(II,J); G1(II,J)=1;
         IF II ^= LAST
          THEN DO; PUT SKIP EDIT(II) (F(2)); LAST=II; END;
         PUT EDIT(J) (F(3));
        END;
        IF II ^= NG1 THEN PUT SKIP EDIT(NG1) (F(2));
/*                                                                 */
/* BUILD ADJACENCY MATRIX AND PRINT ADJACENCY LIST FOR GRAPH G2    */
/*                                                                 */
        PUT PAGE EDIT('ADJACENCY LIST FOR GRAPH G2') (A);
        LAST=0;
        DO I=1 TO NPG2;
         GET FILE(GRAPH) LIST(II,J); G2(II,J)=1;
         IF II ^= LAST
          THEN DO; PUT SKIP EDIT(II) (F(2)); LAST=II; END;
         PUT EDIT(J) (F(3));
        END;
        IF II ^= NG2 THEN PUT SKIP EDIT(NG2) (F(2));
END DGENGR;


DIST: PROCEDURE(M,N);
/*                                                                 */
/*    THE PROCEDURE DIST IMPLEMENTS FLOYD'S ALGORITHM 7 FOR CON-   */
/* STRUCTING A DISTANCE MATRIX.  INITIALLY M IS THE SAME AS THE AD- */
/* JACENCY MATRIX OF A GRAPH, I.E., M(I,J)=1, IF THERE IS AN ARC   */
/* (I,J), OTHERWISE M(I,J)=N, EXCEPT M(I,I)=0. AT THE COMPLETION OF */
/* THE ALGORITHM, M(I,J) IS THE LENGTH OF THE SHORTEST PATH FROM I */
/* TO J.  IF NONE EXISTS, THEN M(I,J)= MAXIMUM PATH LENGTH +  1.   */
/*                                                                 */
  DECLARE S FIXED BIN(15),M(64,64) FIXED BIN(15), N FIXED BIN(15);
  DECLARE L FIXED BIN(15) EXT;
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
  DECLARE (TMDIST(25),TMDALG1(25),TMDRUF(25),NO_GRAPH_PRS)
        FIXED BIN(31) EXT;
/*                                                                 */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                        */
/*                                                                 */
        ITME= -1; CALL ASMTME(ITME);
/*                                                                 */
        INF=N;
```

```
        DO I=1 TO N;
         DO J=1 TO N;
         IF I ¬= J THEN IF M(I,J)=0 THEN M(I,J)=INF;
         END;
        END;
/*                                                                    */
/*          THE FOLLOWING CODE REPRESENTS FLOYD'S ALGORITHM 7         */
/*                                                                    */
        L=0;
        DO I=1 TO N;
         DO J=1 TO N;
         IF M(J,I) < INF
           THEN DO K=1 TO N;
                IF M(I,K) < INF
                   THEN DO; S=M(J,I) + M(I,K);
                        IF S < M(J,K)
                         THEN DO; M(J,K)=S;
                              IF S > L THEN L=S;
                              END;
                        END;
              END;
         END;
        END;
        L=L + 1;
        DO I=1 TO N;
         M(I,I)=0;
         DO J=1 TO N;
         IF M(I,J)= INF THEN M(I,J)=L;
         END;
        END;
/*                                                                    */
/*          END OF FLOYD'S ALGORITHM 7                                */
/*                                                                    */
        DO I=1 TO N;
        PUT SKIP EDIT((M(I,J) DO J=1 TO N)) ((64)F(2));
        END;
/*                                                                    */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE         */
/*                                                                    */
        ITME=1; CALL ASMTME(ITME);
        TMDIST(NO_GRAPH_PRS)=TMDIST(NO_GRAPH_PRS) + ITME;
END DIST;


DALG1: PROCEDURE;
/*                                                                    */
/*    THE PROCEDURE DALG1 IMPLEMENTS ALGORITHM 8.  THE STATEMENT      */
/* NAMES ROUGHLY CORRESPOND TO THE STEP NUMBERS OF ALGORITHM 8.       */
/* THE PROCEDURE USES THE DISTANCE  MATRICES CONSTRUCTED BY DIST      */
/* TO GENERATE THE ROW AND COLUMN CHARACTERISTIC MATRICES WHICH IN    */
/* TURN ARE USED TO GENERATE THE CHARACTERISTIC MATRICES.  FROM       */
/* THE CHARACTERISTIC MATRICES, THE INITIAL PARTITION IS CONSTRUCTED  */
/* AND STORED AT LEVEL 0 IN THE CLASS VECTORS C1 AND C2.  THE         */
```

```
/* VARIABLE AND ARRAY NAMES CORRESPOND TO THOSE USED BY ALGORITHM 8.*/
/*                                                                  */
  DECLARE(G1(64,64),G2(64,64),L,NG1,NG2) FIXED BIN(15) EXT,
         CLASS FIXED BIN(15),NO_ISO BIT(1) EXT,
         (K1,INDEX1(0:64,64),LIST1(0:64,64)) FIXED BIN(15) EXT,
         (K2,INDEX2(0:64,64),LIST2(0:64,64)) FIXED BIN(15) EXT,
         (C1(0:64,64),C2(0:64,64)) FIXED BIN(15) EXT;
  DECLARE (TMDIST(25),TMDALG1(25),TMDRUF(25),NO_GRAPH_PRS)
         FIXED BIN(31) EXT;
  DECLARE ITME FIXED BIN(31),ASMTME ENTRY(FIXED BIN(31));
/*                                                                  */
/* CALL ASMTME TO SET THE SYSTEM TIMER TO 0                         */
/*                                                                  */
         ITME= -1; CALL ASMTME(ITME);
/*                                                                  */
/*           THE FOLLOWING CODE REPRESENTS ALGORITHM 8              */
/*                                                                  */
  STEP1: DO J=1 TO NG1;
          C1(0,J)=0; C2(0,J)=0; INDEX1(0,J)=0; LIST1(0,J)=0;
          INDEX2(0,J)=0; LIST2(0,J)=0;
          END;
         NO_ISO='0'B;
  STEP2_3: BEGIN;
    DECLARE (XR1(NG1,0:NG1),XR2(NG2,0:NG2),XC1(NG1,0:NG1),XC2(NG2,0:NG2
      )) FIXED BIN(15);
         XR1=0; XR2=0; XC1=0; XC2=0;
         DO I=1 TO NG1;
          DO J=1 TO NG1;
           XR1(I,G1(I,J))=XR1(I,G1(I,J)) + 1;
           XR2(I,G2(I,J))=XR2(I,G2(I,J)) + 1;
           XC1(I,G1(J,I))=XC1(I,G1(J,I)) + 1;
           XC2(I,G2(J,I))=XC2(I,G2(J,I)) + 1;
          END;
         END;
  STEP4: CLASS=0; II=0;
  STEP5: CLASS=CLASS + 1;
  STEP6: II =II + 1;
         DO K=II TO NG1;
          IF C1(0,K)=0
            THEN GO TO STEP7_8;
         END;
  STEP6_1:
/*                                                                  */
/* CALL ASMTME TO GET EXECUTION TIME FOR THIS SECTION OF CODE       */
/*                                                                  */
         ITME=1; CALL ASMTME(ITME);
         TMDALG1(NO_GRAPH_PRS)=ITME;
/*                                                                  */
         RETURN;
  STEP7_8: C1(0,K)=CLASS; K1=1;
         INDEX1(0,CLASS)=K; LAST=K;
         DO I=K + 1 TO NG1;
          IF C1(0,I) ^= 0 THEN GO TO NXT_1;
```

```
          DO J=1 TO L;
          IF XR1(I,J)=XR1(K,J) & XC1(I,J)=XC1(K,J)
            THEN;
            ELSE GO TO NXT_1;
          END;
          C1(0,I)=CLASS; K1=K1 + 1;
          LIST1(0,LAST)=I; LAST=I;
  NXT_1: END;
         K2=0;
         DO I=1 TO NG2;
          IF C2(0,I) ¬= 0 THEN GO TO NXT_2;
          DO J=1 TO L;
          IF XR2(I,J)=XR2(K,J) & XC2(I,J)=XC2(K,J)
            THEN;
            ELSE GO TO NXT_2;
          END;
          C2(0,I)=CLASS; K2=K2 + 1;
          IF K2 ¬= 1
           THEN LIST2(0,LAST)=I;
           ELSE INDEX2(0,CLASS)=I;
          LAST=I;
  NXT_2: END;
         IF K1 ¬= K2 THEN DO; NO_ISO='1'B; GO TO STEP6_1; END;
  STEP9: GO TO STEP5;
         END;
/*                                                                    */
/*           END OF ALGORITHM 8                                       */
/*                                                                    */
END DALG1;
```

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA IN BIRMINGHAM
DISSERTATION APPROVAL FORM

Name of Candidate Virginia Charmane May

Major Subject Information Sciences

Title of Dissertation A New Algorithm, and the Evaluation of Current Algorithms, Concerning Graph Isomorphism.

Dissertation Committee:

Chao-Chih Yang, Chairman

Director of Graduate Program

Dean, UAB Graduate School

Date 9 September 1977