1-1-1995

Graph Isomorphism Algorithms: Investigation Of The Graph Isomorphism Problem

Tracy R. Tolley

Robert W. Franceschini

Mikel D. Petty

INSTITUTE FOR SIMULATION AND TRAINING

Institute for Simulation and Training
Computer Generated Forces

October 30, 1995

Graph Isomorphism Algorithms

Investigation of the Graph Isomorphism Problem

Tracy R. Tolley, Robert W. Franceschini, and Mikel D. Petty

IST • 3280 Progress Drive • Orlando, FL 32826
University of Central Florida • Division of Sponsored Research

IST-TR-95-26

Graph Isomorphism Algorithms

Investigation of the Graph Isomorphism Problem

October 31, 1995

Prepared by
Tracy R. Tolley, Robert W. Franceschini,
and Mikel D. Petty

Reviewed by
Clark R. Karr

Institute for Simulation and Training • Computer Generated Forces

IST-TR-95-26

Support Provided by:
University of Central Florida Division of Sponsored Research and Training In-House Grants Program

Institute for Simulation and Training • 3280 Progress Drive • Orlando, Florida 32826

University of Central Florida • Division of Sponsored Research

Investigation of the Graph Isomorphism Problem

TABLE OF CONTENTS

0. Executive Summary

Graph Isomorphism is the problem of determining whether two graphs are, in a structural sense, the same graph. Determining the computational complexity status of the Graph Isomorphism problem is an unsolved problem in Computer Science. It is unknown whether this problem can be solved quickly (i.e., in polynomial time) or whether the problem belongs to a class of difficult problems (NP-Complete).

As a precursor to possible applications of Graph Isomorphism to simulation, IST examined the current status of the problem. This report documents our Graph Isomorphism Algorithms project.

Our primary simulation interest is Computer Generated Forces (CGF), which is a real-time simulation of behaviors of vehicles and soldiers in a synthetic battlefield environment. Because of time constraints, quick algorithms are needed to apply Graph Isomorphism to CGF problems.

We began our Graph Isomorphism Algorithms project with a literature survey. From this survey, we determined that the problem's complexity has not been resolved, and we identified several algorithms for Graph Isomorphism. Next, we implemented three Graph Isomorphism Algorithms. We conducted performance studies of these algorithms on representative data sets of different classes of graphs, including graphs that are specially constructed to be difficult test cases for Graph Isomorphism algorithms. Finally, we analyzed the results of these tests.

Two of the algorithms we implemented produced good timing results. We believe that a combination of these two algorithms could yield an even better algorithm for isomorphism testing.

As a result of this work, we believe that Graph Isomorphism can be applied with success to simulation. The algorithms that we implemented performed well for a wide range of test cases. We believe that simulation applications can be studied to determine characteristics of graphs to be analyzed, and that information can be exploited to eliminate the chance of comparing graphs that form a difficult case for the algorithms.

1. Introduction

As a precursor to possible applications of Graph Isomorphism to simulation, IST examined the current status of the problem. Graph Isomorphism is the problem of determining whether two graphs are, in a structural sense, the same graph. Determining the computational complexity status of the Graph Isomorphism problem is an unsolved problem in Computer Science. It is unknown whether this problem can be solved quickly (i.e., in polynomial time) or whether the problem belongs to a class of difficult problems (NP-Complete).

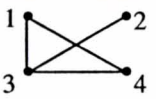
This report documents our Graph Isomorphism Algorithms project. Section 2 provides graph theory background. Section 3 presents the results of our literature survey. Section 4 presents the three algorithms we implemented. Section 5 presents the results of our performance studies of the three algorithms.

2. Background

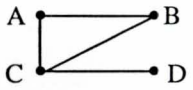
Graph theory is a branch of mathematics devoted to the study of structures known as graphs (Thulasiraman 1992). A graph consists of a set of vertices and a set of edges (edges are pairs of vertices). In spite of this apparent simplicity, graph theory has been applied to a tremendous range of real-world applications, including electrical engineering, chemistry, political science, ecology, genetics, transportation, and information processing (Roberts 1984). Graph theory relates to computer science in two ways; first, graph theory may be applied to many aspects of computer science from data structures to parse trees, and second, many problems in graph theory reveal limits of what can be solved by a computer (Dewdney 1989). Thus, computer scientists often search for algorithmic solutions to graph problems (e.g., (Golumbic 1980) and (Even 1979)).

More formally, a graph G is an ordered pair (V,E) where $V = \{v_1,v_2,...,v_n\}$ is a set of n *vertices* and $E = \{ \{v_{i1},v_{j1}\}, \{v_{i2},v_{j2}\},..., \{v_{im},v_{jm}\}\}$ is a set of m *edges* (i.e., pairs of vertices). We consider only undirected graphs ((a,b) and (b,a) are the same edge {a,b}) and exclude loops and parallel edges. Thus the edges are unordered, the two vertices of any edge are different, and E contains at most one edge $\{v_i,v_j\}$ for any i,j. Although graphs are defined as sets, they are usually represented visually with dots for the vertices and lines connecting the dots for the edges. These figures show two example graphs:

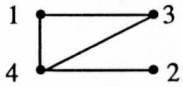
$G_1 = (V_1,E_1)$
$V_1 = \{1,2,3,4\}$
$E_1 = \{\{1,3\},\{1,4\},\{2,4\},\{3,4\}\}$



$G_2 = (V_2,E_2)$
$V_2 = \{A,B,C,D\}$
$E_2 = \{\{A,B\},\{A,D\},\{B,D\},\{C,D\}\}$



Two graphs with the same number of vertices are said to be isomorphic if the vertices of one graph can be relabeled so that it becomes the "same" as the other graph in the sense that the two sets of edges are the same under the labeling. For example, we could relabel the vertices A,B,C,D of G_2 as 1,3,2,4, respectively. The resulting graph (G_3) is:



The edges are renamed: {A,B} as {1,3}, {A,D} as {1,4}, {B,C} as {2,3}, and {C,D} as {2,4}. Thus, the edge set of G_3 is $\{\{1,3\},\{1,4\},\{2,3\},\{2,4\}\}$, which is the same set as E_1. Therefore, G_1 and G_2 are isomorphic.

Graph Isomorphism is the problem of determining whether two arbitrary graphs G_1 and G_2 are isomorphic (Garey 1979). Graph Isomorphism is the problem of determining whether two graphs are, in a structural sense, the same graph. Determining the computational complexity status of the Graph Isomorphism problem is an unsolved problem in Computer Science. It is unknown whether this problem can be solved quickly (i.e., in polynomial time) or whether the problem belongs to a class of difficult problems (NP-Complete).

3. Literature Survey

In this section we present the results of our literature survey of the Graph Isomorphism problem. We briefly mention a few applications of Graph Isomorphism, discuss what is known about the computational complexity of the problem, and discuss some approaches to resolving the problem's status.

3.1. Applications

Applications for Graph Isomorphism can be found in the areas of chemical identification, scene analysis, and construction and enumeration of combinatorial configurations (Corneil 1980). Two applications in Computer Generated Forces (CGF) (Petty 1992) (Petty 1995) concern situational awareness and terrain database correlation.

For situational awareness, tactical situations might be represented by graphs, with vertices representing tactical concepts or objects and edges representing relationships between those concepts (for an example of graph theory used in a CGF task related to situational awareness, see (Petty 1994)). Graph Isomorphism would be used to match the current tactical situation faced by a CGF entity or unit with the tactical situations found in its library of encoded doctrine.

In terrain database correlation, terrain databases for CGF systems often use graphs to represent networks of roads or mobility corridors; typically the CGF systems plan routes on those graphs with A* (Powell 1988) (Stanzione 1989).

Given two terrain databases that purport to be of the same terrain, their road networks can be compared for correlation with Graph Isomorphism.

3.2. Complexity

One open research area in Graph Isomorphism concerns the status of the complexity of the Graph Isomorphism Problem. Graph Isomorphism is known to be in class NP (Garey 1979), but beyond that, little more is certain. Neither a polynomial algorithm for the Graph Isomorphism Problem nor an NP-Completeness proof have been produced to clarify the problem's status. Graph Isomorphism may be a problem that satisfies $P \neq NP$ which implies there is an intermediate class of problems that are polynomially equivalent to Graph Isomorphism (the ISO-Complete class) (Köbler 1993).

Polynomial Graph Isomorphism algorithms exist for restricted classes of graphs. For examples, polynomial algorithms have been developed for trees and planar graphs (Read 1977).

3.3. ISO-Complete and NP-Complete

One approach to resolving Graph Isomorphism is to study other ISO-Complete problems and NP-Complete problems similar to Graph Isomorphism. Some polynomially equivalent problems include computing the automorphism of a graph, deciding directed graph isomorphism, determining cospectral graph isomorphism, and determining whether a graph is self-complementary (Booth 1979). (Basin 1989) presents another problem in the ISO-Complete class: Equality of Terms Containing Associative-Commutative Functions and Commutative Binding Operators. Counting all isomorphisms between graphs is also ISO-complete (Mathon 1979).

Some NP-Complete problems similar to Graph Isomorphism are Subgraph Isomorphism, Automorphism With Restrictions, and Largest Common Subgraph (Lubiw 1981), (Eppstein 1994). By researching either ISO-complete problems or some NP-Complete problems, there is hope of making some progress toward resolving the complexity of the Graph Isomorphism problem.

3.4. Algorithms

Because of the unresolved nature of the Graph Isomorphism Problem, researchers have been driven to explore every aspect of Graph Isomorphism. Summaries of research on Graph Isomorphism are found in (Read 1977) (Gati 1979) (Miller 1979) (Babai 1994).

3.4.1. Unsuccessful Algorithms

Perhaps the best indication of the difficulty and interest in this problem comes from the incorrect or unsuccessful algorithms proposed. For example, two related algorithms for graph isomorphism (Corneil 1970) and (Corneil 1972) and determining automorphism partitioning are based on a false conjecture. A 25-node counter-example can be found in (Mathon 1978). Another attempted solution was to transform graph isomorphism to linear programming (Prabhu 1984)

3.4.2. Difficult Test Cases

By researching particularly difficult cases of graphs (for example, strongly regular graphs), researchers have tried to get a better foothold on Graph Isomorphism (Read 1977) (Mathon 1978). Strongly regular graphs have regular degree and any two adjacent (nonadjacent) vertices are adjacent to exactly λ (respectively μ) other vertices (Corneil 1980).

3.4.3. Algorithms

The current best graph isomorphism algorithm implementation is the *nauty* program (McKay 1981) (Babai 1994). We did not discover this algorithm until late in our research, so it was not used in our experiments described later in this report.

By far the most popular algorithmic approach is to modify the brute force algorithm using invariant information. A graph invariant is a property of a graph that is preserved by isomorphism. Some examples of graph invariants are number of vertices, number of edges, and degree sequence (Thulasiraman 1992). Note that if two graphs are isomorphic then any of these invariants hold, but none of these invariants is sufficient to prove that two graphs are isomorphic. Thus, researchers have been searching for a *complete* graph invariant that is computable in polynomial time; this would yield a polynomial algorithm for Graph Isomorphism (Read 1977).

Vertex classification algorithms reduce the number of permutations of the vertices to be considered. For example, if vertices of a graph G are partitioned into classes based on their degree, then the number of permutations to consider is $\prod_i (p_i)!$, where p_i is the size of class. If this value is small, it is realistic to enumerate all the permutations; otherwise this provides little improvement over enumerating all the permutations.

To improve this method, the iterative vertex classification or refinement method is used. Refinement works by narrowing down the number of permutations that must be considered at each stage of the isomorphism algorithm. Different invariants can be used at each stage.

The "one feel swoop" method directly produces a classification of the vertices (Read 1977).

Once the vertices are partitioned, we will need to test and enumerate all the permutations to verify for isomorphism. The pair of graphs are given a *possible mapping tree* that shows all possible mappings of the vertices. The isomorphism or non-isomorphism is constructed using a breadth-first or depth-first approach (Read 1977).

The most commonly used approach for solving the graph isomorphism problem uses a backtracking procedure as outlined by the vertex classification algorithm. Two of these algorithms were implemented in our experiment (Schmidt 1976) and (Deo 1977). Related approaches can be found in (Mittal 1968) and (Levi 1974).

Another approach to solving this problem can be found in (Goldberg 1983) and yields an algorithm that is $O(exp(N))$.

3.4.4. Graph Invariants

Frequently, graph invariants (properties of a graph that are preserved under isomorphism) are explored in hopes of improving algorithms for graph isomorphism. Graph invariants do not necessarily provide a solution for all graphs on their own (Corneil 1980).

Because *immanants* (*generalized matrix functions)* are difficult to compute and there are examples of nonisomorphic graphs whose adjacency matrices have the same generalized matrix functions, immanants may not be useful for isomorphic testing (Turner 1968).

Results dealing with the characteristic polynomial can be found in (Prabhu 1984) and (Deo 1989).

4. Algorithms Implemented

For this project, the following three algorithms were implemented to test undirected graphs for isomorphism:

A1: Brute-force iteration of all permutations of a graph to test for isomorphism
A2: "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices" (Schmidt 1976)
A3: "A New Algorithm for Digraph Isomorphism" (Deo 1976)

4.1. Brute Force Algorithm (Algorithm 1 – A1)

A1 uses the traditional brute-force method for determining graph isomorphism; enumerate and test all permutations of the vertices of Graph 2 until an isomorphism mapping is found. This determination is made by verifying that every edge in Graph 1 is present in Graph 2 using the currently enumerated permutation and vice versa. If all permutations are exhausted and an isomorphism has not been found, then the two graphs are not isomorphic. The time complexity of this algorithm is $O(N^2 \cdot N!)$ where N is the number of vertices in graphs 1 and 2.

4.1.1. Algorithm

```
Algorithm1()
begin
      isomorphic = BruteForce(N-1)

      if (isomorphic = TRUE) then
            write "Graphs are Isomorphic."
      else
            write "Graphs are NOT Isomorphic."
end
```

Figure 1 — Algorithm 1

```
BruteForce(level)
begin
      result = FALSE

      if (level = -1) then
            result = CheckEdges()
      else
      begin
            i = 0
            while (i < N and result = TRUE)
            begin
                  if (used[i] = FALSE) then
                  begin
                        used[i] = TRUE
                        perm[level] = i
                        result = BruteForce(level-1)
                        used[i] = FALSE
                  end
                  i = i + 1
            end
      end

      return result
```

Figure 2 — BruteForce

```
CheckEdges()
begin
      diff = TRUE

      for x = 0 to N-1
      begin
            y = 0
            while (y < N and diff = FALSE)
            begin
                  if(adj_matrix1[x][y] !=
                     adj_matrix2[perm[x]][perm[y]]) then
                        diff = TRUE
                  y = y + 1
            end
      end

      return diff
end
```

Figure 3 — CheckEdges

The Boolean array `used` marks which vertices have been mapped. The array `perm` records the vertex mapping. For all vertices x in Graph 1, `perm[x]` contains the label of the mapped vertex in Graph 2.

4.2. Distance Matrix Based Algorithm (Algorithm 2 – A2)

The algorithm presented in (Schmidt et al. 1976), uses the distance matrix of a graph to partition the vertices into classes. For most graphs, this initial partitioning of the vertices considerably reduces the number of permutations that must be considered. After the initial partitioning, the classes must be further refined before an isomorphic/non-isomorphic determination can be made. This refinement is accomplished through a backtracking procedure that uses the distance matrix and the class partition from the previous backtracking stage to repartition the vertices. The final result of this repartitioning will be to either have every vertex as the only member of its class or the graphs will not be isomorphic.

We modified the algorithm from (Schmidt et al. 1976) to work for undirected graphs. Our modified algorithm is described here. Note that the algorithm presented here needs a minor modification in order to handle disconnected graphs because the distance matrix will contain infinite distances.

Calculating the complexity for this algorithm is complicated by the backtracking procedure in the algorithm. The time complexity of this algorithm is $\Omega(N^3)$ and $O(N^2 \cdot N!)$. The upper bound on the number of possible mappings we would need to consider is $\prod_{i}^{k} p_i$ where p_i is the number of vertices in the ith class and k is the number of classes. More detailed discussion of the complexity for this algorithm can be found in (Schmidt et al. 1976).

4.2.1. Initial Partitioning

The initial partitioning of vertices into classes requires several steps. See Figure 4 for an example supporting this discussion. The distance matrix (D), an $N \times N$ matrix, stores the shortest path between every pair of vertices in the graph. The distance matrix is then used to compute the row characteristic matrix (XR), an $N \times (N\text{-}1)$. The element $xr_{im} \in XR$ contains the number of vertices that are a distance m away from vertex i. Undirected graphs require an extra column for infinite distances.

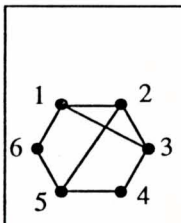
If two vertices have identical rows in XR, they are placed in the same class (C^l). The shading of the rows for XR in Figure 4 represent vertices sharing a common class C^l[i].

Let K^1 be the set of class sizes for Graph 1 and K^2 be the set of class sizes for Graph 2 (i.e., K^1[i] represents the number of vertices in C^1[i]). If these count classes are not equivalent, the graphs are not isomorphic and the algorithm stops. Otherwise, the algorithm proceeds to the backtracking stage. In Figure 4, K^1 and K^2 are equivalent class count sets.

Vertices *1*, *2*, *3*, and *5* are in class number *1* in Graph 1 (i.e., $C^1[1] = C^1[2] = C^1[3] = C^1[5] = 1$) and *a*, *b*, *e*, and *f* are in class number in Graph 2 (i.e., $C^2[a] = C^2[b] = C^2[e] = C^2[f] = 1$), while vertices *4* and *6* are in class number *2* of Graph 1 and *c* and *d* are in class number *2* in Graph 2. From this initial partitioning we know that there must be a mapping as follows:

4 → c and 6 → d (shown in dotted lines in Figure 4) or
4 → d and 6 → c (shown in solid lines in Figure 4)

In this example, the number of permutations is reduced from *N*! (6! = 720) to (4! + 2! = 26) by the initial partitioning.



Graph 1

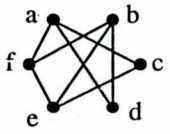
	1	2	3	4	5	6
1	-	1	1	2	2	1
2	1	-	1	2	1	2
3	1	1	-	1	2	2
4	2	2	1	-	1	2
5	2	1	2	1	-	1
6	1	2	2	2	1	-

D^1

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

XR^1

C^1 (1,1,1,2,1,2)
K^1 (4,2,0,0,0,0)



Graph 2

	a	b	c	d	e	f
a	-	2	1	1	2	1
b	2	-	2	1	1	1
c	1	2	-	2	1	2
d	1	1	2	-	2	2
e	2	1	1	2	-	1
f	1	1	2	2	1	-

D^2

	1	2	3	4	5
a	3	2	0	0	0
b	3	2	0	0	0
c	2	3	0	0	0
d	2	3	0	0	0
e	3	2	0	0	0
f	3	2	0	0	0

XR^2

C^2 (1,1,2,2,1,1)
K^2 (4,2,0,0,0,0)

Figure 4 — Example of Initial Partitioning

4.2.2. Backtracking

Let D^1 be the distance matrix for Graph 1 and D^2 the distance matrix for Graph 2. Because the initial partitioning does not necessarily separate all vertices into unique classes, two vertices in the same class must be further partitioned by verifying their mapping for consistency.

The mapping of two vertices is *consistent* if the elements of row *i* of D^1 have corresponding elements in row *r* of D^2 (for all previously mapped vertices). This does not prove that any further mappings are consistent. During a given step in the backtracking algorithm, if no consistent mappings can be found, it is necessary to backtrack and try a different mapping (Schmidt et al. 1976).

Consider the example graphs presented in Figure 4. From the initial partitioning algorithm, we know that class 2 contains the least number of elements, so we will pick a vertex (*i*), from Graph 1 in class 2. In order to further partition the vertices of our graphs, we need to check the consistency of our mapping. This is achieved through a juxtaposition procedure that uses the class vector (*C*) and the *i*th row in the distance matrix for the given graph.

For example, choosing vertex 4 from Graph 1 we have the following juxtaposition:

Distance (Row 4)	Class (Graph 1)	Juxtaposition
2	1	21 (1)
2	1	21 (1)
1	1	11 (2)
–	2	–2 (3)
1	1	11 (2)
2	2	22 (4)

Choosing vertex c from Graph 2 in the same class as vertex 4, we have:

Distance (Row c)	Class (Graph 2)	Juxtaposition
1	1	11 (2)
2	1	21 (1)
–	2	–2 (3)
2	2	22 (4)
1	1	11 (2)
2	1	21 (1)

Now assign new labels are assigned to each of the juxtaposition results: 21→ 1, 11→ 2, –2→ 3, and 22→ 4 (in parenthesis above). The new classes produced are: C^1 = [1,1,2,3,2,4] and C^2 = [2,1,3,4,2,1]. Thus, $K^1 = K^2$ = [2,2,1,1,0,0] and the backtracking algorithm continues to the next stage. Notice that selecting vertex 4 and applying this repartitioning has created two new classes.

The algorithm will continue selecting vertices in Graph 2 until either K^1 is equivalent to K^2 or no vertex can be found in Graph 2 that satisfies this condition. If no such vertex is found the algorithm backtracks to a previous stage and chooses a new vertex from Graph 2. If there is no such previous backtracking stage then the graphs are not isomorphic.

4.2.3. Algorithm

The procedure ***CalculateDistanceMatrix(graph)*** found in Figure 5, calculates the distance matrix for graph, while the procedure ***CalculateCharacteristicMatrix(graph)*** computes the characteristic matrix for graph. ***InitialPartition(c1,c2,k1,k2)*** assigns the vertices in Graph 1 and Graph 2 a class number based on the vertices having equivalent rows in the Characteristic Matrix. ***Backtrack(C1,C2,K1,K2,t)*** is shown in Figure 6.

```
Algorithm2()
begin
      CalculateDistanceMatrix(graph1)
      CalculateDistanceMatrix(graph2)
      CalculateCharacteristicMatrix(graph1)
      CalculateCharacteristicMatrix(graph2)
      InitialPartition(InitC1, InitC2, InitK1, InitK2)
      done = FALSE
      isomorphic = FALSE

      if (InitK1 != InitK2) then isomorphic = FALSE
      else
            Backtrack(InitC1, InitC2, InitK1, InitK2,0)

      if (isomorphic = TRUE) then
            write "Graphs are Isomorphic."
      else
            write "Graphs are NOT Isomorphic."
end
```

Figure 5 — Algorithm 2

```
Backtrack(class1,class2,count1,count2,t)
begin
       have_mapped = TRUE

       if (t = N) then
       begin
              isomorphic = TRUE
              done = TRUE
              /* If class1[i] = class2[r] then i in  */
              /* Graph 1 is mapped to r from Graph 2 */
       end
       else if (done = FALSE) then
       begin
              v1 = ChooseVtx1(class1,count1)

              do
                     v2 = PickByClass(class2,class1[v1])

                     if (v2 does not exist) then
                     begin
                            if (t != 0) then have_mapped = FALSE
                            else
                            begin
                                   isomorphic = FALSE
                                   done = TRUE
                            end
                     end
                     else if (Consistent(newC1,newC2,newK1,newK2,v1,v2)) then
                     begin
                            SaveClass(OldClassVectors, NewClassVectors)

                            /* Mark v1 and v2 as mapped */
                            if (!Backtrack(class1,class2,count1,count2,t+1)) then
                            begin
                                   P[t].is_mapped_to=v2
                                   mapped_g2[v2]=-1

                                   RestoreClass(NewClassVectors, OldClassVectors)

                                   isomorphic = FALSE
                            end
                     end
              while (done = FALSE and have_mapped = TRUE)

       return (have_mapped)
end
```

Figure 6 — Backtrack

In Figure 6, ***ChooseVtx1(class1,count1)*** returns either a new unmapped vertex from the smallest sized class or if stage `t` of the algorithm has been visited previously then the vertex that was chosen earlier is used. Thus, once a vertex is selected from Graph 1 at stage `t`, that vertex will always be used at stage `t`. ***PickByClass(class,given_class)*** returns a vertex that has not been mapped or considered at a previous stage `t` as a candidate for mapping to `v1`. in the `given_class`. ***Consistent(newC1,newC2,newK1,newK2,v1,v2)*** checks to see if the vertices are consistent by creating new classes as outlined in Section 4.2.2 and confirming that the class count arrays for the two graphs are equivalent. ***SaveClass(OldClassVectors)*** stores the `OldClassVectors` (`class1`, `class2`, `count1`, and `count2`) while ***RestoreClass(OldClassVectors)*** restores the previous class vectors. This restoration is needed because if the mapping of `v1` to `v2` has failed then the class partition created by ***Consistent*** is no longer valid. The Boolean variable `have_mapped` keeps track of whether or not we can find a mapping (`v1` → `v2`) at stage `t`.

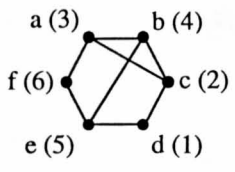
4.3. DFS-Based Algorithm (Algorithm 3 – A3)

A3 starts by finding the degree sequence of each vertex. Then, using a depth-first search, it produces an edge-visitation list for Graph 1. A backtracking algorithm is used to assign a mapping to the vertices. A given stage in the backtracking procedure for A3 will construct an induced subgraph of size k for Graph 1 and try to find an isomorphic induced subgraph in Graph 2. The algorithm description presented here includes modifications for undirected graphs. The average case complexity for this algorithm is $O(N^{logN})$ (Deo et al. 1977).

4.3.1. Initial Setup

Initially the vertices are partitioned into classes based on the degree of the vertices. Since isomorphism preserves degree sequence, if the degree sequences of the two graphs are not equivalent then the graphs are not isomorphic (Trudeau 1976).

A depth-first search on Graph 1 assigns labels to vertices based on their order of visitation. This labeling, represented by the array *num*, replaces the original vertex labeling (see Figure 7).



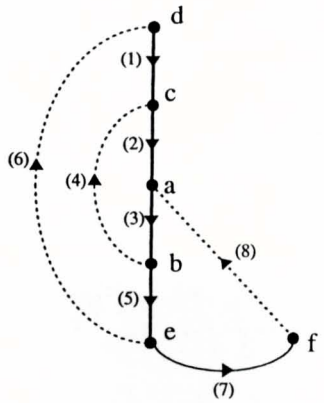
x	num[x]
a	3
b	4
c	2
d	1
e	5
f	6

Figure 7 — num Mapping

Two arrays of size m (m = number of edges + number of trees in DFS forest) are used to keep track of the edges visited. The array *efrom* represents the source vertex (when *efrom[i]* = 0, i is a root vertex in a given connected component) and *eto* the destination vertex. The root vertex is chosen from the smallest degree set. The edges of the graph are then reordered so that all edges of the induced subgraph (of size k) appear before any edge incident to a vertex labeled greater than k. The DFS implementation used here avoids this reordering by storing the edges in order as the vertices are visited. Thus when visiting a given vertex k, all edges from k to previously visited vertices are added before any new vertices are explored. After reordering (if necessary), a list of size m (*order*) is created by letting *order*[i] = MAX(*eto*[i-1], *efrom*[i-1]).

Figure 8 shows an example of the generation of `eto`, `efrom`, and `order` for Graph 1. The tree shown in Figure 8 represents the order of edge visitation by the DFS in Graph 1. Numbers in parenthesis indicate the order in which edges are traversed in the DFS and dotted lines indicate back edges (an edge to a previously visited vertex).

efrom	0	1	2	3	2	4	1	5	3
eto	1	2	3	4	4	5	5	6	6
order	0	1	2	3	4	4	5	5	6



DFS Tree for Graph 1

Figure 8 — Edge Visitation

4.3.2. Backtracking

By recursing through all the edges in Graph 1, A3 builds induced subgraphs of Graph 1 and of Graph 2 and verifies that these induced subgraphs are isomorphic to each other. Consider the *i*th edge to be matched $\{u,v\}$. When v is a root node, we must choose an unmapped vertex in Graph 2 with the same degree as v. This choice allows us to make an intelligent decision about a matching vertex in Graph 2 that will act as the source node for Graph 2's induced subgraph.

Let $u = k\text{-}1$ and $v = k$ (where k has not been previously mapped) and add this edge to the subgraph (*match_count* is set to 1). Then each edge after that in the *eto* and *efrom* list is added and confirmed to have a matching edge in Graph 2 until an unmapped vertex is seen (k+1). For each edge that is added, *match_count* is increased by 1.

Before we map the new vertex, k+1, we must verify that the mapping formed by the previously mapped vertex (k) was a correct one. This is done by considering vertex, i, adjacent to k and verifying that there is an edge that corresponds to this edge for every previously mapped i (*match_count* is decreased by 1). We should not have any more or less edges than the number of edges added between the first occurrence of k in the *efrom* list and the first occurrence of k+1 in the *efrom* list because of the nature of the ordering of the edges in the lists. If this is true, we have verified the isomorphism of the two induced subgraphs and we may map k+1 into an unmapped vertex in Graph 2 with the same degree. If the two induced subgraphs are not isomorphic, then we must map a new vertex from Graph 2 into k (Deo et al 1977).

4.3.3. Algorithm

```
Algorithm3()
begin
      CalculateDegree(graph1)
      CalculateDegree(graph2)

      isomorphic = FALSE

      ZeroOutMappings(pair1, pair2)

      if (degree1 != degree2) then isomorphic = FALSE
      else
      begin
            while (Unvisited())
            begin
                  source = ChooseFromMinDegreeSet(degree1)
                  DFS(source)
            end

            Match(1)
      end

      if (isomorphic = TRUE)then
            write "Graphs are Isomorphic."
      else
            write "Graphs are NOT Isomorphic."
end
```

Figure 9 — Algorithm 3

Algorithm 3 is shown in Figure 9. The procedure ***CalculateDegree(graph)*** calculates the degree sequence for `graph` and the function ***Unvisited()*** returns whether there are any vertices in Graph 1 that have not been visited by the DFS. ***ZeroOutMappings(pair1,pair2)*** assigns 0 to all elements of `pair1` and `pair2`. ***ChooseFromMinDegreeSet(degree)*** chooses a vertex that is in the smallest degree set and ***DFS(source)*** executes a depth-first search from vertex `source`.

```
Match(edge)
begin
        u = efrom[edge]
        v = eto[edge]

        if (isomorphic = FALSE) then
        begin
                if (edge > total_number_edges) then isomorphic = TRUE
                else if (u = 0) then
                begin
                        y = ChooseVtx(degree1[v],degree2)
                        Map(v,y)
                        mc = 0
                        isomorphic = FALSE
                        Match(edge+1)

                        if (isomorphic = FALSE) then Unmap(v,y)
                end
        end
        else if (pair[v] = -1) then
        begin
                z = pair1[order[edge]]

                for w = 1 to N
                begin
                        if (adj_matrix[w][z] = 1 and pair2[w] != 0) then
                                mc = mc - 1

                        if (mc != 0)
                                return
                        else
                        begin
                                y = ChooseAdjVertex(pair1[u],degree1[u],degree2)
                                Map(v,y)
                                mc = 1
                                isomorphic = FALSE
                                Match(edge + 1)

                                if (isomorphic = FALSE) then Unmap(v,y)
                        end
                        else
                        begin
                                verify = adj_matrix[pair1[v][pair1[y]]

                                if (verify = TRUE) then
                                begin
                                        mc = mc + 1
                                        isomorphic = FALSE
                                        Match(edge + 1)
                                end
                        end
                end
        end
end
```

Figure 10 — Backtracking Procedure (Match)

Match() is shown in Figure 10. The procedure ***ChooseVtx(given_degree, degree2)*** picks an unmapped vertex in Graph 2 with degree equal to `given_degree`. ***Map(v,y)*** assigns a mapping `v` (in Graph 1) to `y` (in Graph 2), while ***Unmap(v,y)*** removes this mapping. **ChooseAdjVertex*(vtx2,given_degree,degree2)*** returns an unmapped vertex in Graph 2 that is adjacent to `vtx2` and in the `given_degree`.

5. Test Results

The test results outlined in this paper were conducted on a 60 MHz Pentium with 16M of RAM using the WATCOM development environment (V10.0) with Rational Systems' DOS/4GW DOS Extender.

For this section of the paper A1 will refer to the implementation of Algorithm 1, A2 the implementation of Algorithm 2, and A3 the implementation of Algorithm 3. A1, A2, and A3 were tested with both simple test cases (path, cycle, complete, wheel, and star graphs) and more complicated graphs constructed by using graphs presented by Mathon (Mathon 1978). For A2 and A3 the times represented in all figures and tables are computed by running the algorithm on the trial 100 times and then dividing this time by 100. A1's time is the result of a single execution of the trial.

5.1. Test I - Path, Cycle, Complete, Wheel, and Star Graphs

The first test consisted of testing the three algorithms with path, cycle, complete, wheel, and star graphs of size 1 to 100 nodes. These graphs were compared against a randomly generated permutation of the given graph.

5.1.1. Test I - The Results

This section contains the results for the test outlined above.

5.1.1.1. Path & Cycle

A1, A2, and A3 exhibit roughly the same behavior for paths or cycles (see Figure 11 and Figure 12). A3 yields very good performance, while A1 is off the chart at 7 or 8 nodes. A2 is roughly polynomial for paths and cycles. The similar performance of these two cases (path and cycle) can be explained by the similarity in the graphs (a cycle is a path with an edge joining the two endpoints of the path).

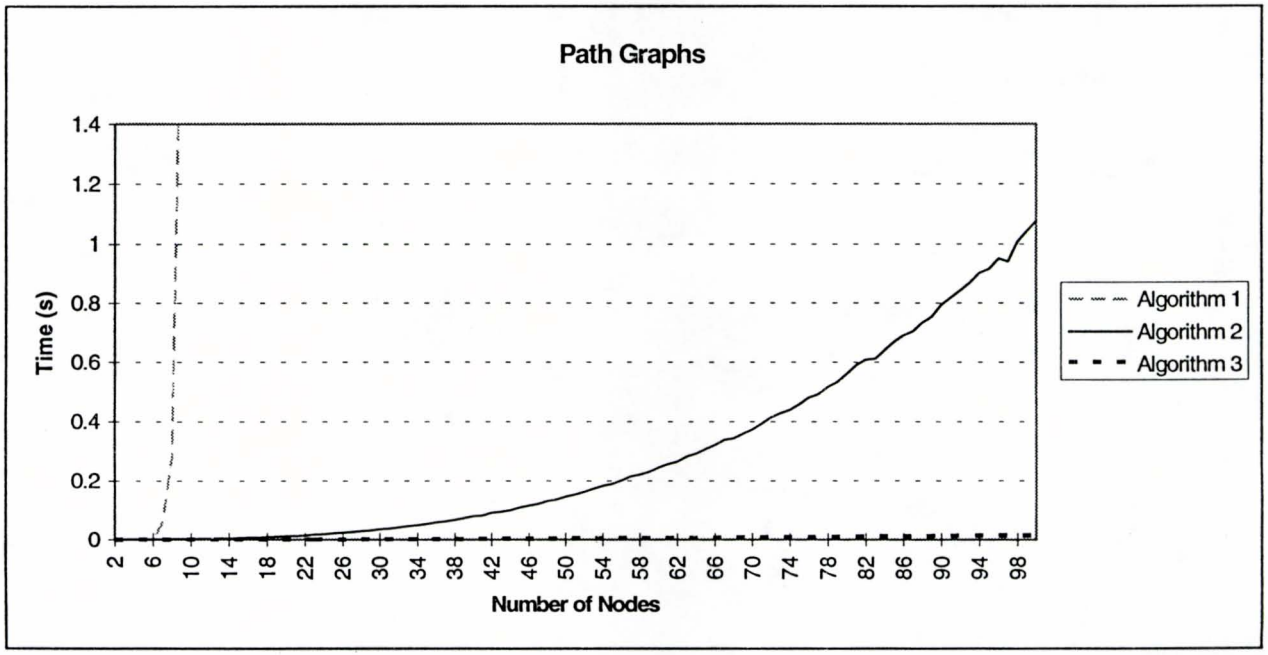


Figure 11 — Path Graphs

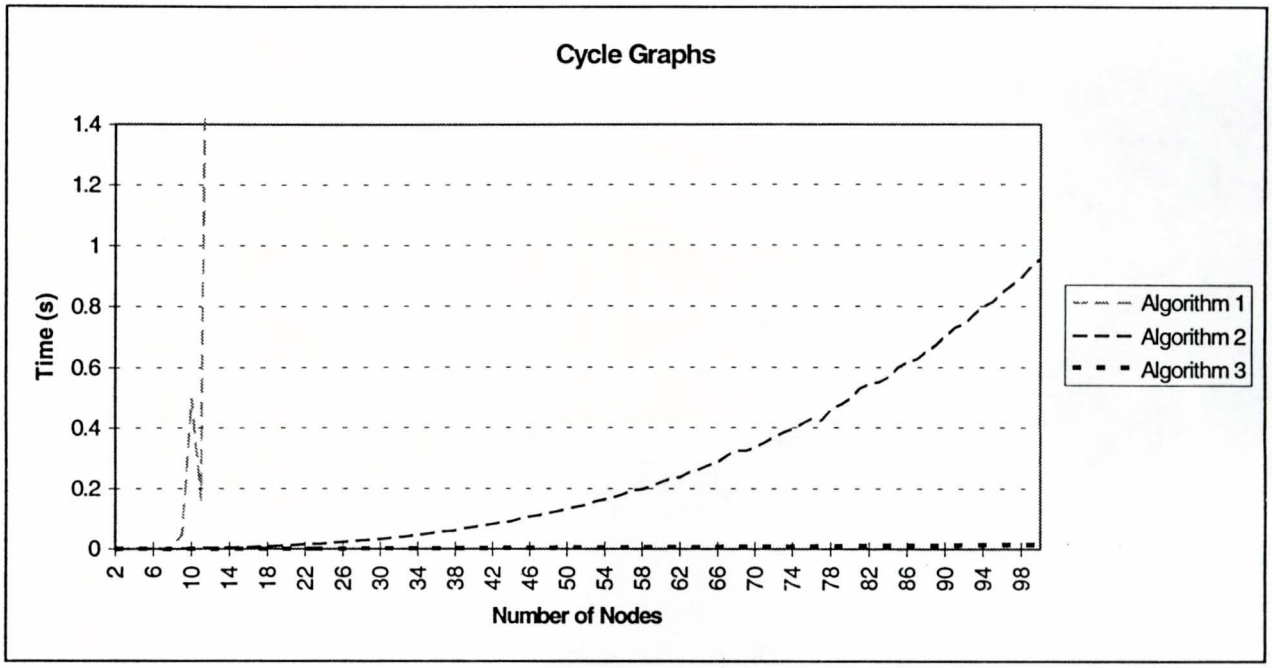


Figure 12 — Cycle Graphs

5.1.1.2. Star & Wheel

A star on N vertices consists of one center node and N-1 neighboring vertices. A wheel graph is a cycle on N-1 vertices with a single vertex added that neighbors the existing N-1 vertices (Roberts 1984). Figure 13 shows an example of a star and a wheel graph on 6 vertices.

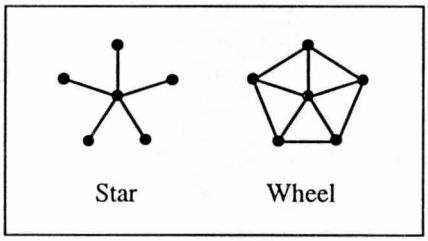


Figure 13 — Example of Star & Wheel Graphs

From Figure 14 and Figure 15, we can see that A1, A2, and A3 have roughly the same behavior for either star or wheel graphs. Note that once an algorithm has correctly mapped the center node in a star graph, it has found the correct permutation. This lucky match provides better results in some of the test cases and thus the sharp drops in the performance charts for star graphs. Notice that A3 performs better than A1 and A2.

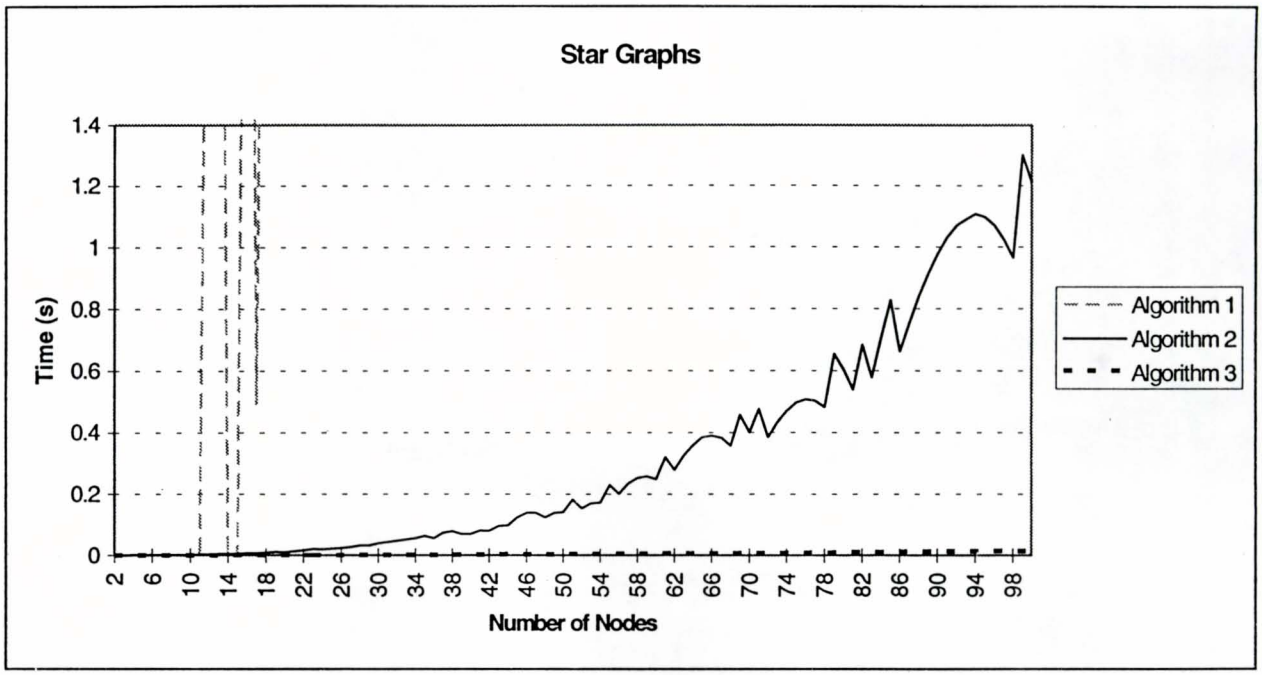


Figure 14 — Star Graphs

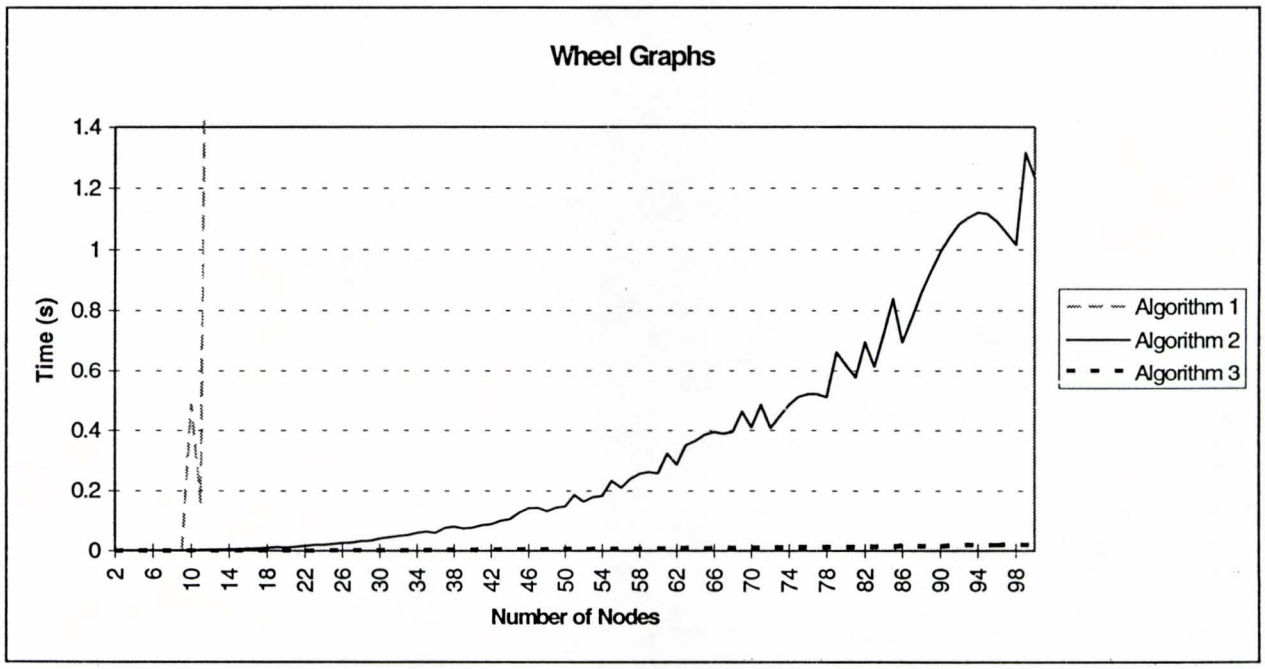


Figure 15 — Wheel Graphs

5.1.1.3. Complete

Because there is only one adjacency matrix that represents a complete graph, when A1 picks a permutation for Graph 2 it will always be the correct one. The complete graph test case highlights the extra overhead involved in A2 and A3 to make these algorithms more efficient for general graphs (see Figure 16).

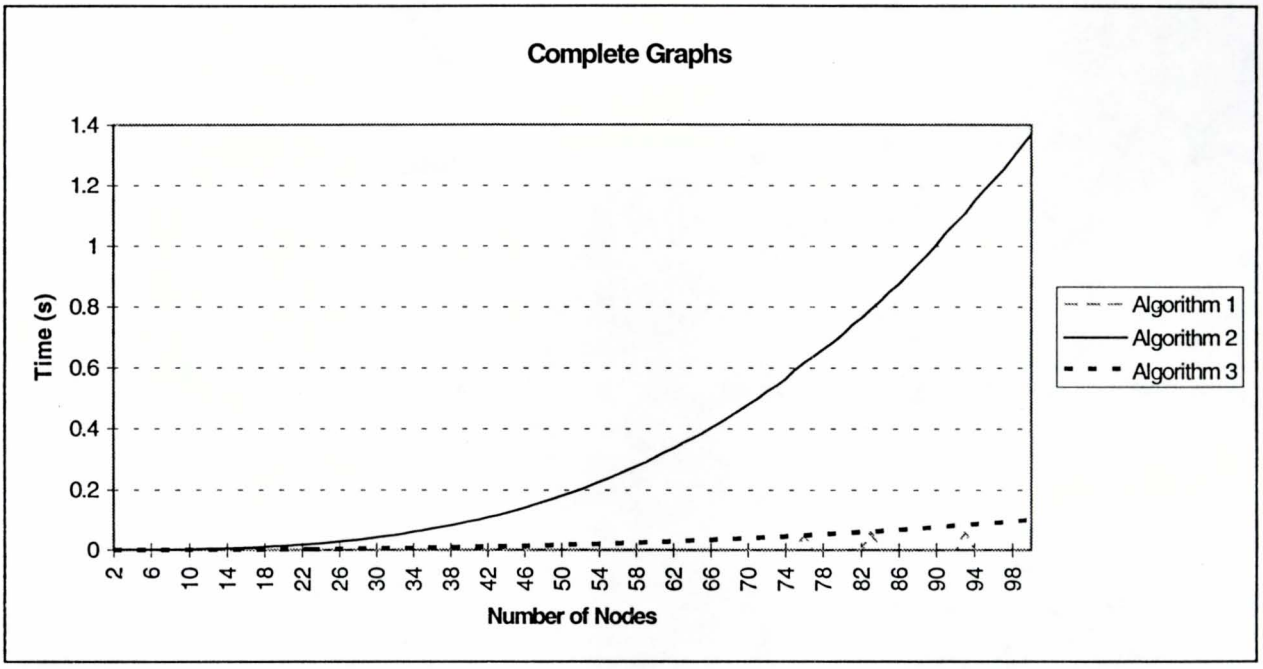


Figure 16 — Complete Graphs

5.2. Test II - More Challenging Graphs

5.2.1. Description of Test

To produce more challenging test cases for A2 and A3, a testing procedure using graphs presented by Mathon is applied. Mathon illustrates 20 example graphs of sizes 25, 29, 35, and 50 nodes (Mathon 1978). The testing procedure consists of taking all graphs in a given node size category and generating from each graph two other graphs. The first graph represents a random remapping of the given graph and the second the complement of the given graph. The complement of a graph G has the same vertex set, but an edge is present in the complement, only if it is not present in the edge set of graph G. Then all graphs in a given node size were tested against each other for isomorphism. For example, let graphs A and B be original graphs from Mathon's paper. The following graphs are produced:

> A, Acomp (A complement), Amap (random permutation of A), B, Bcomp (B complement), Bmap (random permutation of B)

The graphs created above are paired off to produce the following test cases (X-Y assigns Graph X as Graph 1 and Graph Y as Graph 2).

> A–A, A–Acomp, A–Amap, A–B, A–Bcomp, A–Bmap
> Acomp–A, Acomp–Acomp, Acomp–Amap, Acomp–B, Acomp–Bcomp, Acomp–Bmap
> etc.

Note that these test cases include symmetric test cases, such as A–Acomp and Acomp–A. Both these test cases must be considered because the graph chosen as Graph 1 can affect the final results of the test. This is particularly true in A3 which orders the edges it considers based on a DFS of Graph 1.

5.2.2. The Results

Applying the Brute-force algorithm to this sample graphs is out of the question because of the lack of efficiency of this approach. The figures in these sections show results for A2 runs represented by circles and A3 runs by triangles. These results are plotted on a logarithmic scale with each triangle or circle representing a single test case as describe in Section 5.2.1.

5.2.2.1. 25-Node Graphs

A2 and A3 were tested on 144 25-Node test cases as outlined above. The results of this test are plotted in Figure 17. Notice that times for A1 stay clustered around the mean, while A3 has times much greater than A2 and also much lower. This behavior persists throughout the larger node graphs. From Table 1, we can see that A2's overall performance is better than A3's.

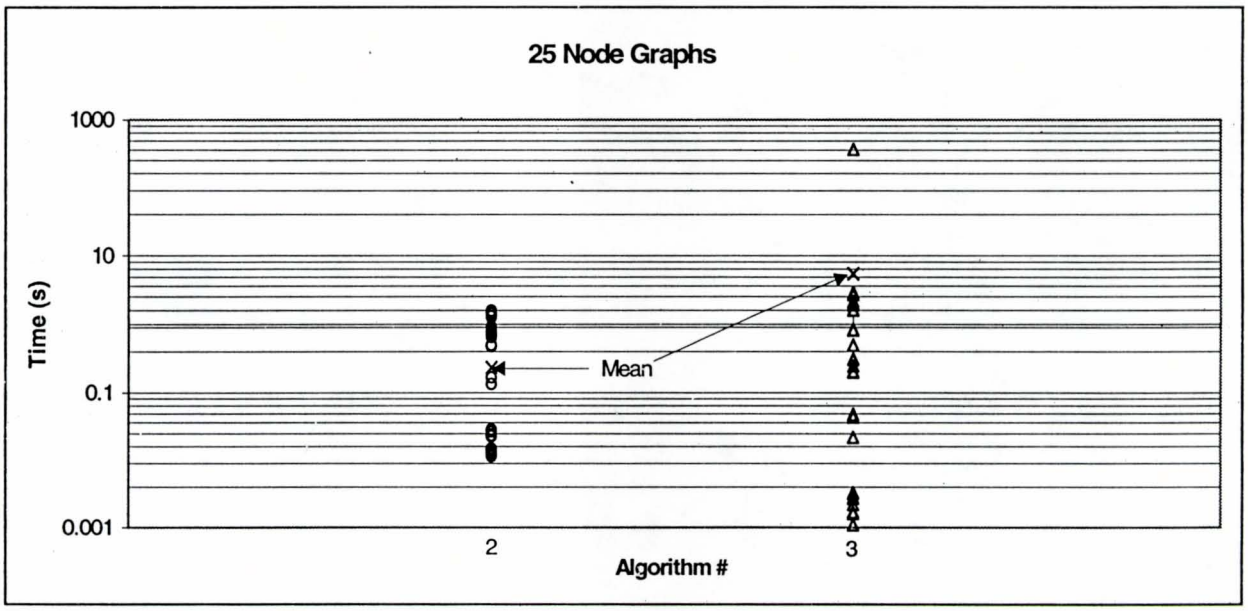


Figure 17 — 25-Node Graphs

	A2	A3
Max	1.556	365.692
Min	0.0115	0.0011
StdDev	0.384392	42.90433
Avg	0.233131	5.406794

Table 1 — 25-Node Statistics

Note that a particularly challenging test case (365.692 seconds) for A3 can be found by using the complement of A_{25} and the complement of B_{25} as test cases (A_{25} and B_{25} can be found in Table 1 of Mathon's "Sample Graphs for Isomorphism Testing" (Mathon 1978).

5.2.2.2. 29-Node Graphs

This test group contained 576 29-Node test cases. Surprisingly, A2 and A3 performed better overall on these graphs than the 25-Node graphs (see Table 2).

	A2	A3
Max	3.5943	5.8633
Min	0.0209	0.00005
StdDev	0.9174219	1.60005967
Avg	0.19108021	0.1472

Table 2 — 29-Node Graphs

Note that in Figure 18, we see the same distribution of A2 and A3 that we had for 25-Node graphs.

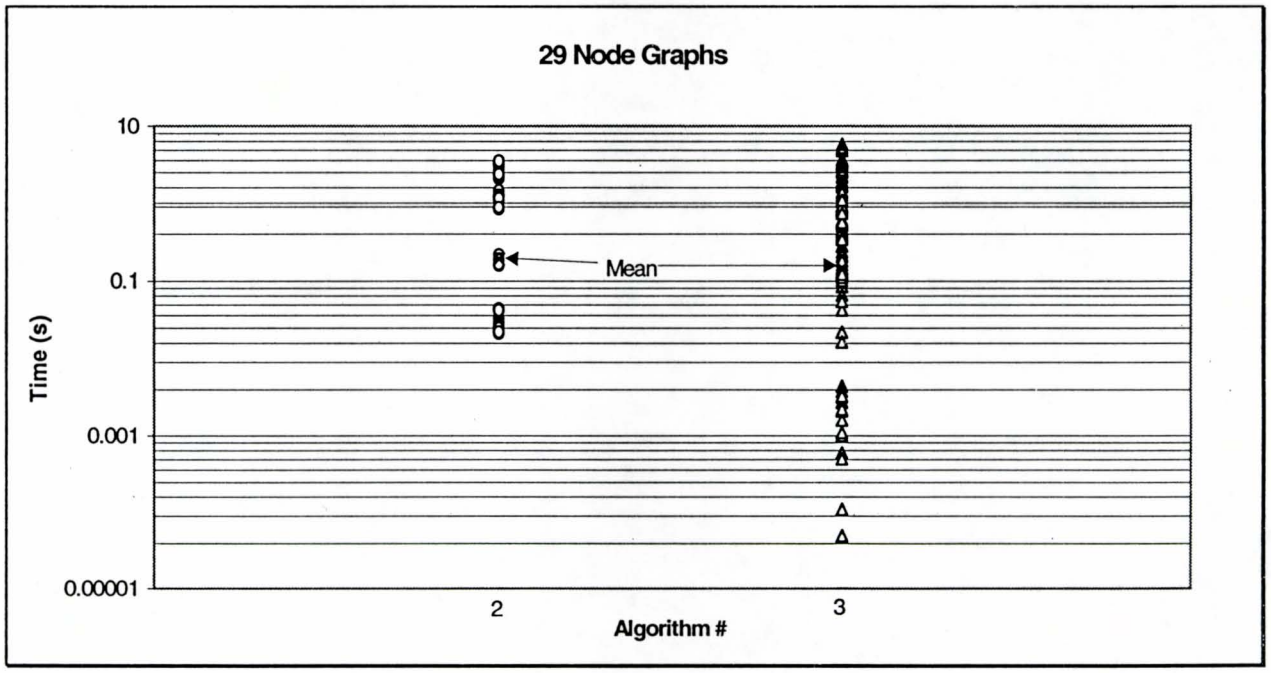


Figure 18 — 29-Node Graphs

5.2.2.3. 35-Node Graphs

This test group contained 144 35-Node test cases. As seen in Table 3, the average performance for A2 and A3 for the 35-Node graphs was not as good as the 29-Node, but it was still better than the 25-Node graphs.

	A2	A3
Max	2.6496	24.0191
Min	0.0351	0.0005
StdDev	0.65839	4.786871
Avg	0.243841	1.362854

Table 3 — 35-Node Statistics

Again in Figure 19 we have similar distribution to 25-Node and 29-Node graphs.

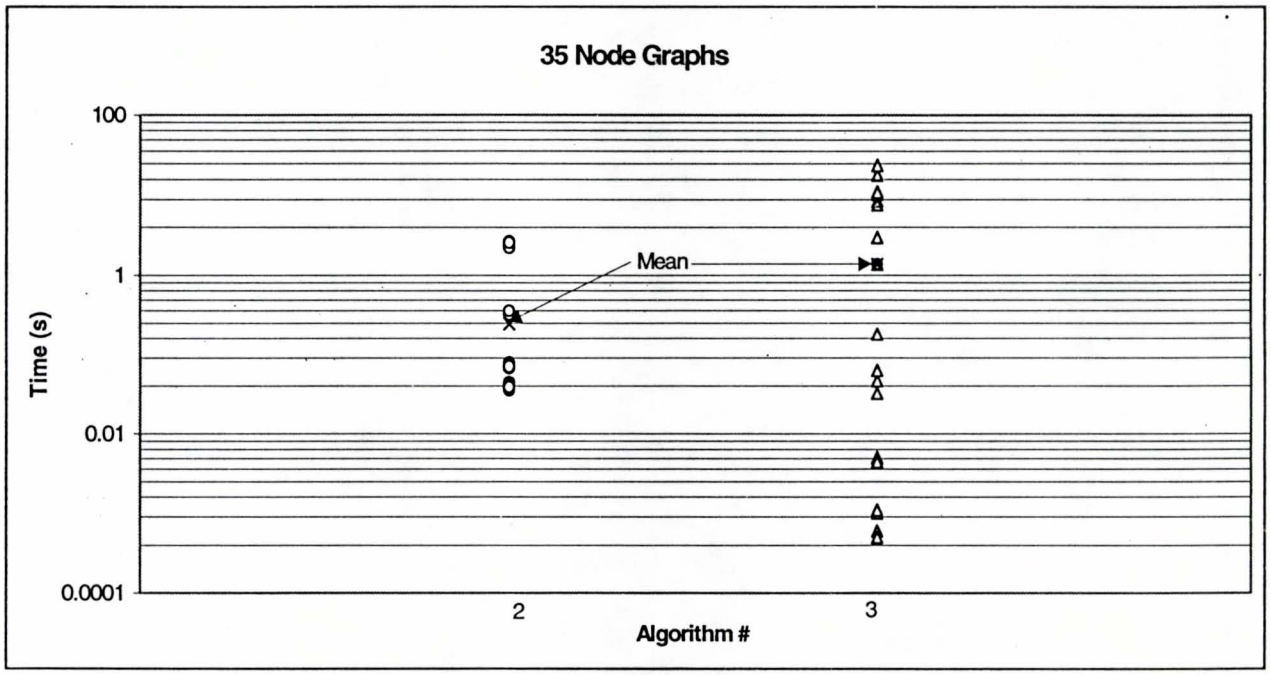


Figure 19 — 35-Node Graphs

5.2.2.4. 50-Node Graphs

For 6 of the 50 node cases, A3 fails to return an answer within a day, for these test cases A3 has been assigned an arbitrary maximum time (1000 seconds) for plotting purposes (see Figure 20).

	A2	A3
Max	16.6623	1000
Min	0.084	0.0016
StdDev	5.606483	200.3471
Avg	2.956804	42.62331

Table 4 — 50-Node Statistics

Note that the test cases that did not finish for A3 were the only test cases for A2 that took approximately 16 seconds to complete (the maximum time of any test cases for A2). A2 either had performance less than 0.1 seconds or times greater than 10 seconds (these longer times occurred in 30 of the 144 test cases).

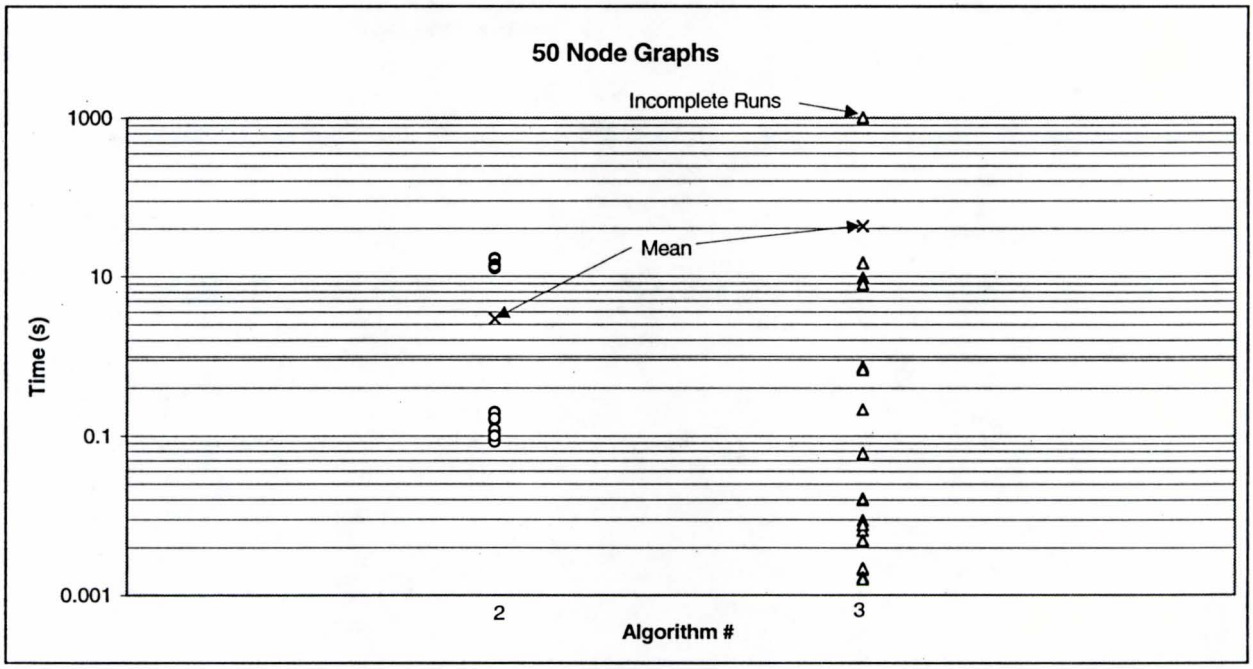


Figure 20 — 50-Node Graphs

5.3. Conclusion

A2 tends to exhibit excellent behavior with Mathon's graphs, but pays a penalty for the extra overhead when dealing with more general graphs. A3 has lower times overall (See minimum values for A3 in Table 1-4), but has a lot of problems with the graphs designed to challenge Graph Isomorphism Algorithms. A1 performs best only when it can find the isomorphic mapping with the first permutation considered.

6. Future Work

This project raises several interesting research questions for further work:

1. Algorithm 2, based on the distance matrix, produced very good results and was well behaved for all test cases. What is it about the distance matrix that gives this algorithm its strength? Is there any other information not captured in the distance matrix about a graph? If so, what is it and can it be computed and augmented to Algorithm 2?

2. We found one paper (as mentioned earlier) that studied graphs that are particularly difficult for isomorphism testing algorithms. We believe that more work is needed in categorizing and studying difficult graphs.

3. We believe that Algorithms 2 and 3 could be combined in some fashion to produce a new algorithm that has the best characteristics of each algorithm without the disadvantages of each algorithm. One way to do this might be to run each algorithm in parallel (on two different processors) on the same data set; the performance of this new algorithm on a data case would be determined by which algorithm finished the data case first. Investigation of this approach to combining algorithms might yield useful algorithms for real-time processing of graphs in general.

4. We may now turn our attention to applying Graph Isomorphism to simulation, and specifically to CGF. We will propose ways of using Graph Isomorphism for CGF. We will also classify the types of graphs that are used in our CGF applications, to determine whether they form one of the subclasses of graphs that is known to have a polynomial-time algorithm for isomorphism testing.

We hope to address each of these topics in our future work.

7. References

Babai, L.(1994). "Automorphism groups, isomorphism, reconstruction". Chapter 27 of *Handbook of Combinatorics. University of Chicago* TR-94-10.

Basin, D. (1989). "Equality of Terms Containing Associative-Commutative Functions and Binding Operators is Isomorphism Complete", *Cornell University TR-89-1020.*

Booth, K.and Colbourn, C. (1979). "Problems Polynomial Equivalent to Graph Isomorphism". *University of Waterloo CS-77-04.*

Corneil, D. (1972). "An Algorithm for Determining the Automorphism Partitioning of an Undirected Graph", *BIT 12*, pp. 161-171.

Corneil, D. and Gotlieb, C. (1970). "An Efficient Algorithm for Graph Isomorphism". *Journal of the Association for Computing Machinery*, Vol. 17, No. 1, Jan., pp. 51-64.

Corneil, D. and Kirkpatrick D. (1980). "A Theoretical Analysis of Various Heuristics for the Graph Isomorphism Problem", *SIAM J. Comput.*, Vol 9, No. 8, May 1980. pp. 281-297.

Deo, N., Davis, J. and Lord, R. (1977). "A New Algorithm for Digraph Isomorphism", *BIT*, Vol. 17, pp. 16-30.

Deo, N., Harary, F., and Schwenk, A. (1989). "An Eigenvector Characterization of Cospectral Graphs Having Cospectral Joins", *Combinatorial Mathematics: Proceedings of the Third International Conference, Vol. 555 of the Annals of the New York Academy of Sciences*, May 1989, pp. 159-166.

Dewdney, A.K. (1989). *The Turing Omnibus.* Computer Science Press, Rockville MD, 415 pages.

Eppstein, D (1994). "Subgraph Isomorphism in Planar Graphs and Related Problems", *Department of Information and Computer Science TR 94-25*, University of California, Irvine, CA.

Even, S. (1979). *Graph Algorithms.* Computer Science Press, Potomac MD, 249 pages.

Garey, M.R. and Johnson, D.S. (1979). *Computers and Intractability.* W.H. Freeman, New York NY, 340 pages.

Gati, G. (1979). "Further Annotated Bibliography on the Isomorphism Disease", *Journal of Graph Theory*, Vol. 3, No. 3, Summer 1979, pp 95-109.

Goldberg, M. (1983). "A Nonfactorial Algorithm for Testing Isomorphism of Two Graphs", *Discrete Applied Mathematics 6*, pp. 229-236.

Golumbic, M.C. (1980). *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York NY, 284 pages.

Köbler, J., Schöning, U., and Torán, J. (1993). *The Graph Isomorphism Problem: Its Structural Complexity*, Birkäuser, Boston, pp.1-4.

Levi, G. (1974). "Graph Isomorphism: A Heuristic Edge-Partitioning-Oriented Algorithm", *Computing 12*, pp. 291-313.

Lubiw, A. (1981). "Some NP-Complete Problems Similar to Graph Isomorphism", *SIAM J. Comput.*, Vol. 10, No. 8, pp. 11-21.

Mathon, R. (1979). "A Note on the Graph Isomorphism Counting Problem", *Information Processing Letters*, Vol 8, number 3, pp. 131-132.

Mathon, R. (1978). "Sample Graphs for Isomorphism Testing", *Proc. 9th S-E Conf. Combinatorics, Graph Theory, and Computing*, pp. 499-517.

McKay, B. (1981). "Practical Graph Isomorphism", *Congressus Numerantium*, Vol. 30, pp. 45-87.
`ftp://dcssoft.anu.edu.au/pub/nauty19`

Miller, G (1979). "Graph Isomorphism, General Remarks", *Journal of Computer and System Sciences 18*, pp. 128-142.

Mittal, H. (1988). "A Fast Backtracking Algorithm for Graph Isomorphism", *Information Processing Letters 29*, pp. 105-110.

Petty, M. (1992). "Computer Generated Forces in Battlefield Simulation", *Proceedings of the Southeastern Simulation Conference 1992* (Pensacola, FL, Oct. 22-23). The Society for Computer Simulation, 56-71.

Petty, M. and D. Van Brackle. (1994). "Reconnaissance Planning in Polygonal Terrain", *Proceedings of the 5th International Training Equipment Conference* (The Hague, The Netherlands, Apr. 26-28). 314-327.

Petty, M. (1995). "Computer Generated Forces in Distributed Interactive Simulation", *Distributed Interactive Simulation Systems for Simulation and Training in Aerospace Environment* (Orlando, FL, Apr. 19-20). SPIE Critical Review 58, 251-280.

Powell, D., J. Wright, G. Slentz, and P. Kundsen. (1988). "Representations to Support Reasoning on Terrain", *Proceedings of the U.S. Army Symposium on Artificial Intelligence Research for Exploitation for the Battlefield Environment* (El Paso, TX, Nov. 15-16). 212-222.

Prabhu, G. and Deo, N. (1984). "The Ellipsoid Algorithm and the Graph Isomorphism Problem", *Technique et Science Informatiques 0752-4072/84/05/327-7*, pp. 327-333.

Prabhu, G. and Deo, N. (1984). "On the Power of a Perturbation for Testing Non-Isomorphism of Graphs", *BIT 24*, 302-307.

Read, R. and Corneil, D. (1977) "The Graph Isomorphism Disease", *Journal of Graph Theory, Vol. 1*, pp. 339-363.

Roberts, F. (1984). *Applied Combinatorics*, Prentice-Hall, Inc., New Jersey, pp. 91, 442-446.

Schmidt, D. and Druffel, L. (1976). "A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices", *Journal of the ACM*, Vol. 23, No. 3, July 1976, pp. 443-445.

Stanzione, T. (1989). "Terrain Reasoning in SIMNET Semi-Automated Forces System", *Geo '89 Symposium on Geographical Information Systems for Command and Control.* SHAPE Technical Centre, The Hague, The Netherlands, Oct. 1989.

Thulasiraman K., and Swamy, M. (1992). *Graphs: Theory and Algorithms*, John Wiley & Sons, Inc., New York.

Trudeau, R. (1976). *Dots and Lines*, The Kent State University Press, pp. 41-55.

Turner, J. (1968). "Generalized Matrix Functions and the Graph Isomorphism Problem", *SIAM J. Appl. Math.*, Vol. 16, No. 3, May 1968, pp. 520-526.