8-1-2000

Graph theory and tolerance graphs.

Michael D. Hazelwood

GRAPH THEORY AND TOLERANCE GRAPHS

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Michael D. Hazelwood

August 2000

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the Requirements for the degree Master of Science, University of Nebraska at Omaha.

Committee

Susan Wiedenbeck

Azad Azadmanesh Azadmanesh

Hesham Ali

Chairperson Hesham Ali

Date 7/24/00

GRAPH THEORY AND TOLERANCE GRAPHS

Michael D. Hazelwood, MS

University of Nebraska, 2000

Advisor: Dr. Hesham H. Ali

Graphs are diagrams made up of nodes and edges. The nodes are the points on the graph. The edges are the lines connecting the nodes. These graphs are useful in that they allow for the modeling of real world problems into a format that can be readily solved by computers.

Graph theory can be used in fields as diverse as chemistry, transportation, and music. However, graph theory is not being fully utilized because of the level of knowledge required to use it. The first of three goals of this thesis is to make graph theory accessible to a larger audience by developing a graphical application. This application allows a user to create a graph, apply a graph algorithm, and display the results through a graphical user interface.

The second goal of this thesis is to implement the most useful graph algorithms. This includes basic algorithms that have been well researched and can be solved in polynomial time. Advanced algorithms for the class of graphs known as perfect graphs will also be implemented.

The third goal is to add to graph theory to make it more practical. A relatively new class of graphs known as tolerance graphs allows for variations that occur in real

world problems. The nodes on a tolerance graph correspond to intervals on a real line. Each interval has a tolerance value. Edges are drawn between two nodes if the intersection of the two intervals is greater than the tolerance of either interval.

This thesis examines known algorithms for tolerance graphs. There are still some open problems dealing with tolerance graphs. Among them are the problem of recognizing tolerance graphs and converting a known tolerance graph into a tolerance representation. These two problems will be explored within this thesis.

Table of Contents

Table of Figures

Chapter 1 Introduction

1.1 Motivation

A problem that all computer scientists face is how to solve real world problems using computers. Many of those real world problems can be generalized as an arrangement of objects and some relationships between those objects. Such problems include finding the shortest route between two cities and finding the best assignment of a company's workers to a list of tasks. These types of problems are the focus of a branch of computer science known as graph algorithms.

A great deal of research has been performed in the field of graph theory. This research has led to a large number of algorithms that can solve a wide variety of problems in fields as diverse as chemistry, transportation, and even music [WW90].

Although many graph algorithms have been developed, they are not being fully utilized because of the amount of work required to implement those algorithms. The first of the three goals of this thesis is to make those graph algorithms more easily useable. This will be achieved through the creation of a graphical application, which will allow a user to apply graph algorithms by simple pointing and clicking. This application should make graph algorithms accessible to a larger audience.

The second goal of this thesis is to implement the most useful graph algorithms. This includes basic graph algorithms that have been well researched and can be solved in polynomial time. Advanced algorithms for the class of graphs known as perfect graphs will also be implemented.

The third goal of this thesis is to add a piece to graph theory that will make it more practical. A relatively new type of graph is called tolerance graphs. These graphs add an element of flexibility, which make them more useful in some real-world applications. There are several open problems associated with tolerance graphs. This thesis includes work toward solving two of those problems.

1.2 Graph Theory

Before continuing discussion of this thesis, some basic information must be covered. First, graph theory can be defined as the study of using graphs, and the algorithms based on those graphs, to solve problems. Solving a problem using graph theory involves several steps. First, the problem is modeled into a diagram called a graph. The graph represents the objects in the problem and the relationship between those objects. For example, a graph could be created to model the local streets in a city. It could contain all of the intersections in the city and the streets that connect those intersections.

Graphs are actually diagrams made up of nodes and edges. The nodes (also called vertices) are the points on a graph. The edges are lines connecting the nodes. Figure 1.1 shows a graph that contains 5 nodes and 8 edges. A node usually represents an object, while an edge represents some relationship between two objects. In the city streets example, each of the nodes could represent an intersection, and the edges could represent a street that connects two intersections.

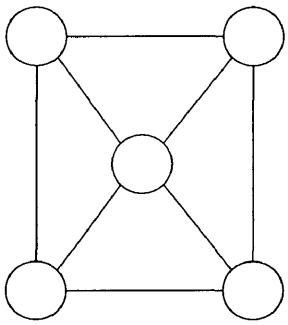


Figure 1.1 – Graph Representing 8 Streets and 5 Intersections

These graphs can be modified to better represent objects and relationships. Weights can be assigned to the nodes and/or edges to represent characteristics of the objects and relationships like cost or size. Returning to the city streets, each edge could be weighted based on the length of the associated street. An algorithm could then be used to find the shortest path between two intersections by using the shortest streets.

The edges can be directed to represent a one-way relationship between two nodes. Also, multiple edges can exist between the same two nodes to represent multiple

relationships, possibly with different weights. These properties could be used to represent one-way streets in a city.

1.3 Importance of Graphs

These graphs are useful in that they can be used to model real world problems in a format that can be solved algorithmically. A great deal of work has been done to develop algorithms to solve graph problems. As such, if a real world problem can be represented in graph form, it can likely be solved using a graph algorithm.

This concept is best described though an example. Consider the problem of assigning classes to classrooms. In this problem, two classes cannot be assigned to the same room if they are taught at the same time or at overlapping times. So a solution must be found which assigns the classes to classrooms without two classes being assigned to the same classroom at the same time.

To use graph theory, the problem is first represented in graph form. In this graph, the nodes will represent individual classes. The edges in the graph will represent a conflict between two classes. That is, an edge is drawn between two nodes if the meeting times of the two classes overlap and therefore must not be assigned to the same room. Figure 1.2 below illustrates a possible graph. In this graph, classes 1 and 4 cannot be

assigned to the same room because their meeting times overlap. This is represented in the graph by drawing an edge between nodes 1 and 4.

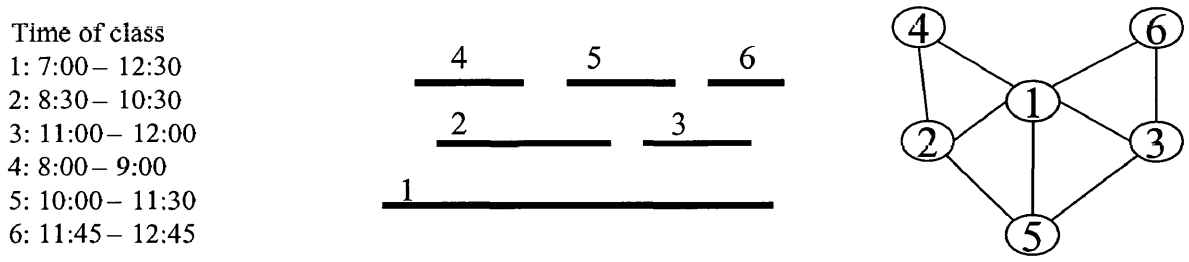


Figure 1.2 – Classrooms Example

Once this problem is transformed into a graph, one of several graph algorithms can be applied to it. For example, a minimum coloring algorithm can be used to find the minimum number of classrooms needed. For the above graph, a minimum of three classrooms would be required ({1}, {2,3}, {4,5,6}).

A maximum independent set algorithm could be applied to the graph to find the most classes that a single instructor could teach. In the above graph, the maximum number of classes a single instructor could teach is three (classes 4, 5, and 6).

While the solutions to the above small example are easy to determine without the use of algorithms, the assignment of a thousand classes would necessitate the use of an algorithmic approach.

Extensive effort has been applied to developing graph algorithms that can be used to solve a large variety of problems. Graph theory can be useful in the fields of

transportation, scheduling, integrated circuit design, computer networks, and distributed processing, to name a few. In transportation, graph theory could be used to find the quickest route between two cities. In scheduling, an optimum assignment of jobs to employees could be computed. Graph theory could be used in integrated circuit design to find an optimal layout of components. In computer networks, graph theory can be used to find the shortest route between two computers and to locate bottlenecks in the network. Finally, in distributed processing, graph theory can be used to assign tasks in a multiprocessor environment.

1.4 Goals and Objectives

This thesis contains three goals in the area of graph theory and graph algorithms. The first goal is to create a graphical application to allow a larger audience to make use of graph theory. Next, the more important graph algorithms will be implemented and made available in the application. The last goal will be to research two of the open problems in a newer class of graphs known as tolerance graphs.

1.4.1 Application

The first goal is motivated by the problem that graph theory is not being fully utilized because of the level of knowledge required. The solution to this problem is through the creation of a user-friendly application. This application will make graph theory

accessible to a larger audience. It will be called the Java-based Graph Algorithm Program or J-GAP. Researchers and students can use this application to model problems by simply adding nodes and edges through a graphical user interface. They can then find the solution to the problem by selecting the appropriate algorithm from a drop-down menu.

Previous graph theory applications had the drawback that they could only execute on a particular platform (like Windows-only or Linux-only). Because J-GAP is written in Java, it has the advantage of being platform independent. The same application can execute on any platform for which a Java virtual machine is available[1]. Graph files created by J-GAP are also platform independent. So, for example, a graph that is created and saved on a Linux workstation can be opened on a Microsoft Windows personal computer.

J-GAP will contain a graphical user interface and will give the user the ability to create a graph, test for characteristics of the graph, and implement many algorithms on the graph. As illustrated below, the graphical interface will be made up of three parts: a drop down menu, a drawing area, and a status bar. By default, the program runs in a 600x400 window, but can easily be resized by dragging on a corner of the window.

[1] A list of platforms that support Java is available at http://java.sun.com/cgi-bin/java-ports.cgi

The drawing area is a scrollable window that is 1280 x 1024 pixels in size. It is in this window that a graph can be created and the results of an algorithm are visualized.

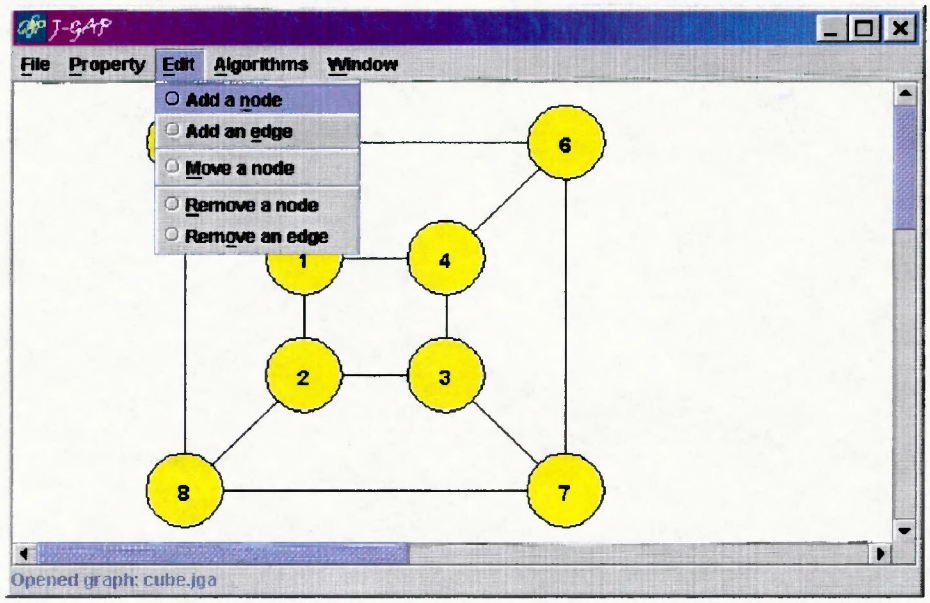


Figure 1.3 – Application Interface

Below the drawing area, along the bottom of the window, is a status bar. This bar may contain instructions based on menu selections. It may also display the results of an algorithm.

The drop-down menu will allow a user to perform functions like saving the current graph, opening a graph file, printing the graph, and exiting the program. It also allows the user to add and remove nodes and edges and to implement algorithms.

1.4.2 Implementation of Algorithms

An extensive amount of algorithms have been developed in the field of graph theory. The second goal of this thesis is to implement a set of the most useful graph algorithms. This will include algorithms that are representative of graph theory as a whole.

1.4.2.1 Classification

Graphs can represent a wide variety of real-world problems. Often an algorithm that solves a given problem for every possible graph requires a large amount of time to complete.

The graphs can be classified based on their properties and on the type of real-world problem they represent. By classifying a graph, a faster algorithm may be used. This is because a graph algorithm that is applicable to all graphs often does not execute as quickly as an algorithm designed for the specific type of graph being used. To get an idea of the importance of classifying a graph, one of the problems discussed in chapter 2, known as the maximum clique problem, will be analyzed. Finding a solution to this problem could take from months to years for a large graph. If the graph belongs to one of several classes listed in section 2.5, a better algorithm could be applied to find the solution for the same graph in a matter of only seconds.

Because of the importance of classifying a given graph, classification algorithms will be covered in this thesis and included in the accompanying application. Chapter 2 lists the classes of graphs that will be tested for in this application.

1.4.2.2 Algorithms and Heuristics

Three different types of algorithms will be implemented. First, well-known algorithms will be implemented and made available. The second type of algorithms to be implemented is the less researched algorithms and heuristics that have been discussed, but are not in an easily implementable format. Finally, this thesis will develop new algorithms for the class of tolerance graphs.

1.4.3 Tolerance Graphs

The third goal of this thesis is to research a relatively new class of graphs known as tolerance graphs. Tolerance graphs are a generalization of interval graphs and are defined in chapter 2. These graphs are practical in that they have more flexibility to allow for the variations that occur in real world problems. Because tolerance graphs are new, several important problems have not yet been solved. These open problems include recognizing whether a graph belongs to the class of tolerance graphs. As research is applied to tolerance graphs, more efficient algorithms are being developed to solve

various graph problems. If a graph can be recognized as being a tolerance graph, these more efficient algorithms can be utilized. Since recognizing tolerance graphs is an important open problem, research will be made in the development of an efficient algorithm to recognize tolerance graphs.

Chapter 2 Formal Definitions

This chapter introduces the basic terminology used in the field of graph theory. Components and characteristics of graphs are defined. Then the graph problems covered in this thesis are introduced along with an introduction to the algorithms that solve each problem.

2.1 Components of a Graph

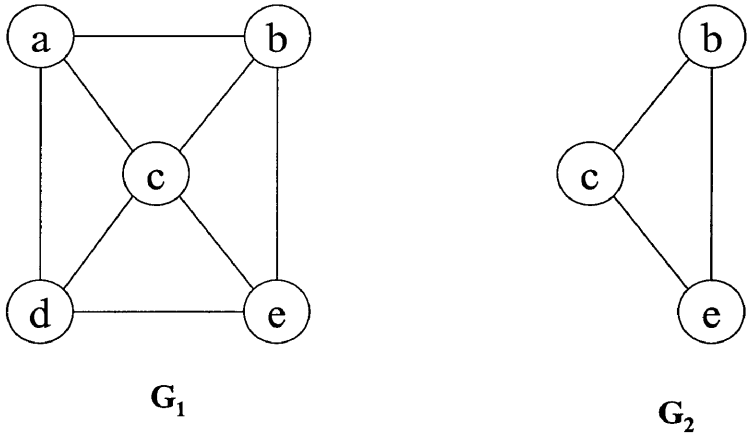
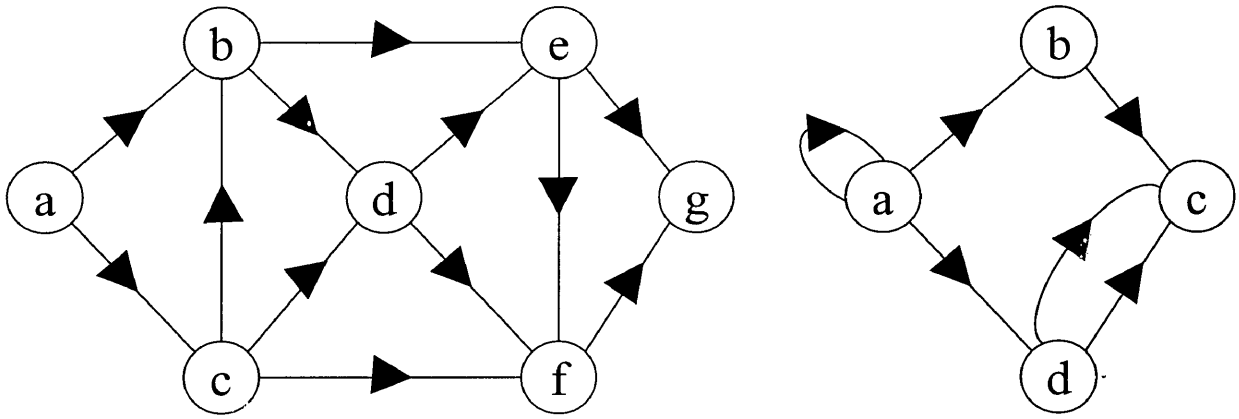


Figure 2.1 – Graph Components

A **graph** is a diagram of vertices (also called nodes) and edges. A graph G is said to be made up of a non-empty set of vertices V(G) and a list of edges E(G). Each edge is represented as a pair of vertices (v_1,v_2). In the above graph, G_1, the set of vertices is $\{a,b,c,d,e\}$, and the list of edges is $\{(a,b),(a,c),(a,d),(b,c),(b,e),(c,d),(c,e),(d,e)\}$. Two vertices are considered **adjacent** if there exists an edge that connects the two vertices. In

G_1, the vertex a is adjacent to the vertex b because the edge (a,b) exists in the graph. Also, the two vertices are **incident** to the edge and the edge is incident to both of the vertices. In G_1, the edge (a,b) is incident to the vertices a an b. The **degree** of a vertex is equal to the number of edges incident to that vertex. In G_1, all of the vertices have a degree of 3 except for vertex c, which has a degree of 4.

If G' is a **subgraph** of a graph G, then G' contains a subset of the vertices and edges in G. Furthermore, an **induced subgraph** G' of a graph G contains a subset of the vertices in G and every edge in G that is incident to the vertices in G'. In figure 2.1, G_2 is an induced subgraph of G_1, which includes the vertices b, c, and e.



G_3 G_4

Figure 2.2 – Directed Graphs

A **directed graph** or a **digraph** is a graph where the edges are ordered pairs of vertices. These directed edges (also referred to as **arcs**) create a one-way connection

between vertices. Thus, the arc (v_1,v_2) connects vertex v_1 to v_2 but does not connect v_2 to v_1. In a directed graph, the **in-degree** of a vertex is equal to the number of edges going to the vertex. Likewise, the **out-degree** of a vertex is equal to the number of edges leaving the vertex. In figure 2.2, vertex *f* of graph G_3 has an in-degree of 3 and an out degree of 1.

A **simple graph** contains no multiple edges and no loops. A graph is said to contain **multiple edges** if two or more edges connect the same two nodes. A **loop** is an edge that connects a vertex to itself. In figure 2.2, the graph G_4 is not a simple graph because there are more than one edges connecting vertex *d* to vertex *c* and there is a self-loop on vertex *a*.

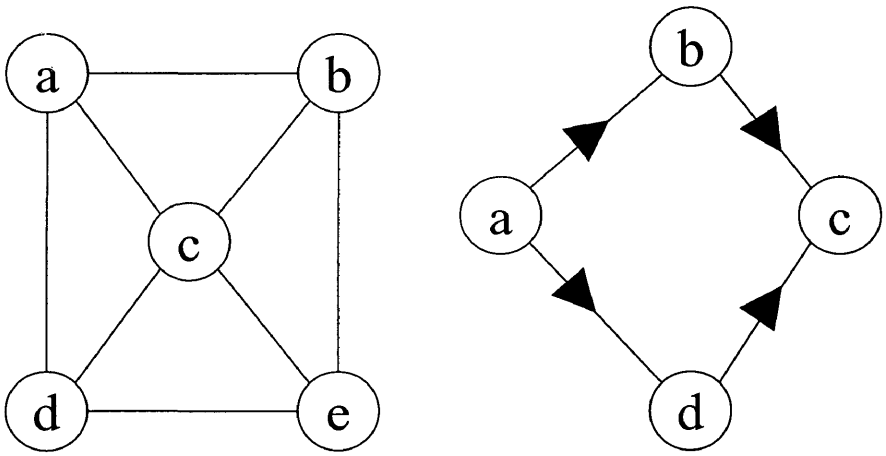
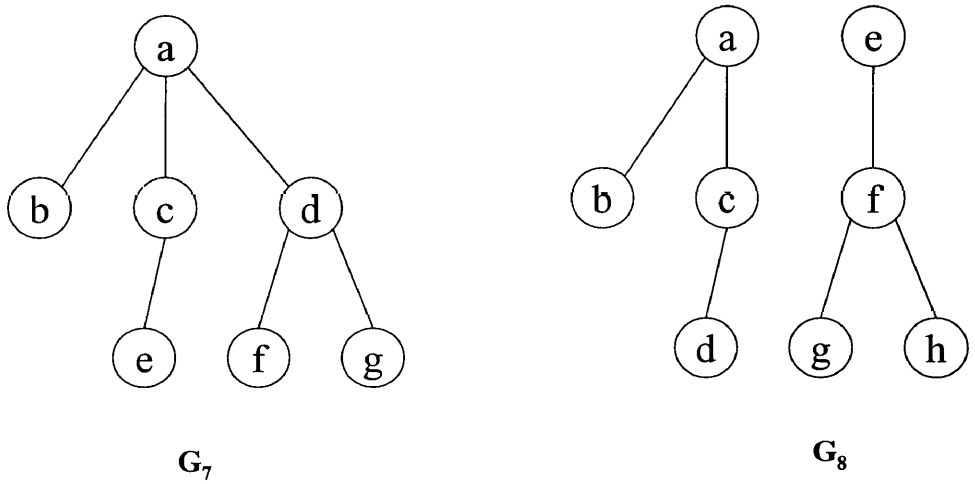


Figure 2.3 – Undirected and Directed Graphs

A **walk** is a series of edges in a graph in which the second vertex of one edge in the series is equal to the first vertex of the next edge. In other words, if a walk contains

an edge from vertex v_1 to v_2, the next edge must start at v_2. In G_5, [(*a*,*b*),(*b*,*a*),(*a*,*c*)] is an example of a walk. The walk can also be identified by the nodes that it includes. The previous walk could also be written as *abac*. A **trail** is a walk in which no edge is used twice. An example of a trail in G_5 is *abcad*. A **path** is a trail in which no vertex is visited more than once. If the first vertex in a walk is equal to the last vertex, then it is called a **closed walk**. Similarly, a **closed trail** is a trail in which the first and last vertices are the same. A **cycle** is a closed trail in which all vertices are unique except for the first and last vertex. One of the cycles in G_5 is *abeda*. An **acyclic** graph is defined as a graph that contains no cycles.

An undirected graph is considered to be **connected** if a path exists between every pair of vertices. If the graph is disconnected, then the connected subgraphs form the **components** of the graph. A digraph is **strongly connected** if for every pair of vertices v_1 and v_2, there exists a path from $v_1$to v_2 and from v_2 to v_1. In figure 2.3, the graph G_6 is not strongly connected because there does not exist a path from vertex *c* to vertex *a*.



G_7 G_8

Figure 2.4 – Trees and Forests

A **bridge** or a **cut-edge** is an edge in a connected graph, which if removed would break the graph into two components. In the graph G_7, the edge (a,b) is a bridge because removing the edge would result in an unconnected graph with two components. Similarly, a **cut-vertex** is a vertex in a connected graph whose removal would break the graph into two or more subgraphs. In the graph G_7, vertex d is a cut-vertex whose removal would result in a graph with three components [WW90].

A **tree** is a connected undirected acyclic graph. Another way to characterize a tree is that it is an undirected graph in which there is only one path between any two vertices. Trees are useful because they can represent hieratical relationships. The graph G_7 is an example of a tree. A **forest** is an undirected graph (not necessarily connected) in which each of the components forms a tree. The graphs G_7 and G_8 are examples of forests.

2.2 NP-completeness

Before looking at different types of graph problems, it is necessary to define terms used to describe the complexity of a given problem. The complexity of a problem can be described as the amount of time required to find a solution.

The amount of time required for an algorithm to execute is often dependent on the size of the input. For example, most graph algorithms execute quicker for small graphs than for larger graphs. The relationship between the execution time relative to the input size is known as the time complexity of an algorithm. One algorithm is better than another if it has a lower time complexity. A polynomial algorithm is an algorithm that has a polynomial relationship between the size of the input and the running time of the algorithm. Exponential algorithms have an exponential relationship between input size and running time. The time required to execute an exponential algorithm quickly becomes infeasible as the input size grows. As such, polynomial algorithms are preferred over exponential algorithms.

In general, problems can fall into one of several categories, the first of which is polynomial or P. Polynomial problems are problems for which a polynomial-time

algorithm is known. In other words, when given inputs of size n, the worst-case running time of the algorithm is $O(n^k)$ for some constant k.

The second category of problems is NP-complete problems. These are the problems for which there is no known polynomial solution. Furthermore, if a polynomial solution is found for one NP-complete problem, then all NP-complete problems can be solved in polynomial time. This is highly unlikely given the large number of NP-complete problems for which no polynomial solution has been found [CL 90].

Since NP-complete problems cannot be solved in polynomial time, one of two options must be chosen. First, an exponential algorithm can be used to find the optimum solution. This solution is only practical for small input sizes. The second option is to use a heuristic. Rather than finding the optimum solution in exponential time, heuristics find a near-optimum solution in polynomial time. In other words, a heuristic can quickly find a solution that is good but not necessarily the best solution.

2.3 Basic Graph Problems

Using the idea of NP-completeness, graph problems can be broken into two categories. Basic graph problems can be solved in polynomial time. Advanced graph problems require exponential time to execute for general graphs. Basic graph algorithms

include finding the minimum spanning tree, the shortest path, the maximum matching, and the maximum network flow.

2.3.1 Shortest Path Algorithms

A shortest path algorithm finds the shortest distance from one node to another node. They may also find the shortest distance from a starting node to all other nodes. The shortest path in an edge-weighted graph is defined as the path with the minimum total edge weights.

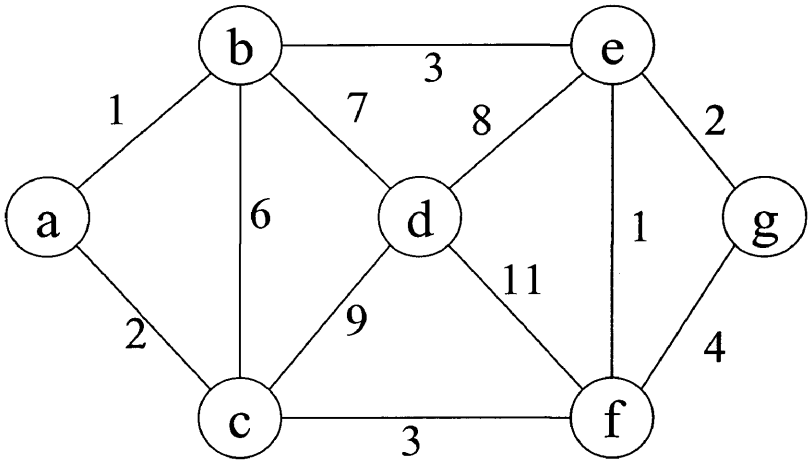


Figure 2.5 – Weighted Graph

An example of where shortest path algorithms are useful is in computer networks where two computers in the network need to communicate. Figure 2.5 illustrates a computer network represented as weighted graph with 7 nodes and edge weights

representing the delay between neighboring routers. In this example, the best path between the routers a and g is $[a,b,e,g]$ which has a total cost of 6.

Two shortest path algorithms are implemented in this thesis. Both of these algorithms are described in chapter 4.

Dijkstra's algorithm computes the shortest path from a single source node to all other nodes in a weighted directed graph in which all of the edge weights are nonnegative.

The Bellman-Ford algorithm also calculates the shortest path in a given weighted directed graph from a given node to all other nodes in the graph. This algorithm allows for negative edge weights. However, a negative-weight cycle is not allowed.

2.3.2 Minimum Spanning Tree

The minimum spanning tree algorithms find a tree that connects all of the nodes in a given graph such that the edges have minimum total weight.

Figure 2.6 illustrates an edge-weighted graph and its associated minimum spanning tree. This tree connects all of the nodes in the graph using the minimum total edge weights.

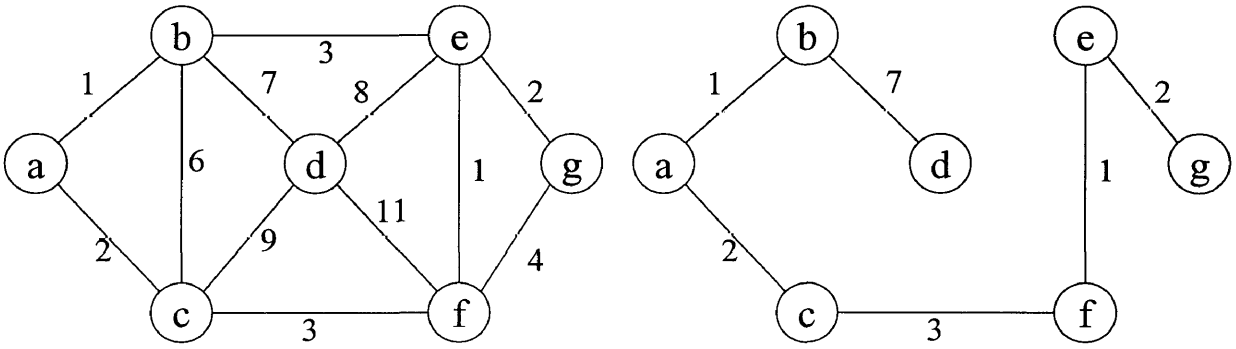


Figure 2.6 – Minimum Spanning Tree

Kruskal's minimum spanning tree algorithm sorts the edges in ascending order of their weights. It then adds each edge to the tree unless doing so would create a cycle.

Prim's minimum spanning tree algorithm starts by adding a single node to the tree. It then adds the smallest edge between a node in the tree and a node not yet in the tree. It repeats adding edges until the complete tree is formed.

2.3.3 Maximum Network Flow

A maximum network flow algorithm determines the maximum number of units that can flow from a source node in a network to a sink node.

The Ford-Fulkerson algorithm operates on the idea of residual graphs and augmenting paths. Every arc (u,v) in a given network is represented by two arcs in a

residual graph: (u,v) and (v,u). The weight of the arc (v,u) represents the flow from node u to node v. The weight of the arc (u,v) represents the residual of the arc, or how much the flow can be increased in the arc of the network.

An augmenting path in the residual graph is a path from the source to the sink that has positive weights on all of its arcs. The algorithm works by repeatedly finding augmenting paths in the residual graph, and increasing the flow along the path until no residual paths exist. At this point, the maximum flow through the network has been determined.

2.3.4 Maximum Matching

The matching problem is to find the maximum set of edges that are pairwise independent. That is, any one node in the graph is incident to at most one of the edges in the matching.

In figure 2.6, a maximum matching is the set of the three edges (a,b), (c,d), and (e,f). Each of the edges in the matching is independent of the others.

The maximum matching problem can be solved in polynomial time. Chapter 4 includes a description of the algorithm used in this thesis.

2.3.5 Transitive Orientation

A transitive orientation applies a direction to each edge of an undirected graph such that if there exists an edge from node u to node v and an edge from node v to node w, there must exist an edge from node u to node w. Not every graph has a possible transitive orientation. A polynomial algorithm will be provided that finds a transitive orientation for a graph if one exists.

The graph G_{12} in figure 2.7 can be transitively oriented as shown in the graph G_{13}. In this orientation, since there is an edge from vertex b to vertex d and an edge from vertex d to vertex a, there must be an edge from vertex b to vertex a if this orientation is transitive. Since this property is maintained for all edges, the orientation is indeed a transitive orientation.

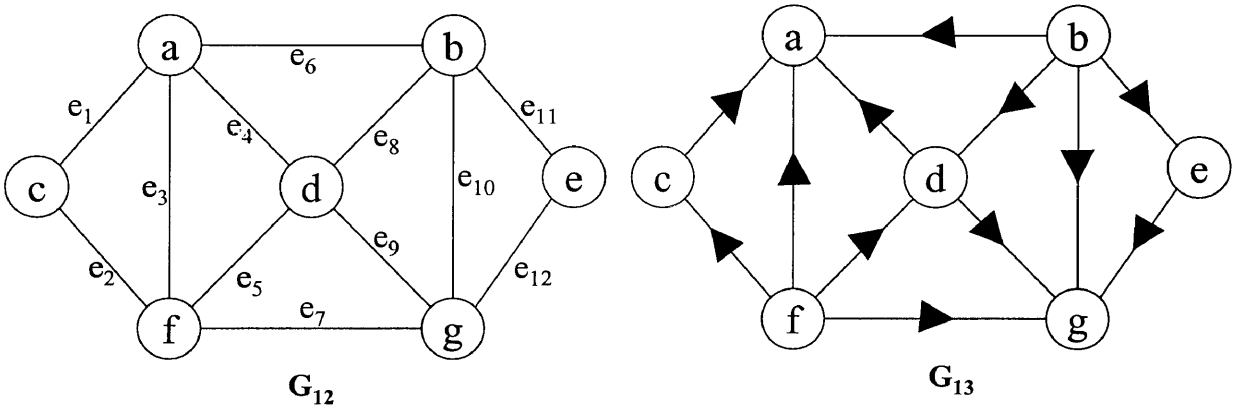


Figure 2.7 – Transitive Orientation

2.3.6 Euler Tour

An Euler tour is a closed trail (the first node is the same as the last node) that includes each edge exactly once. For example, if the edges of a graph represented the streets that a postal worker needed to visit, an Euler tour would be the best path to take so that each street was visited once without backtracking on the same street.

A graph can be easily tested to see if it contains an Euler tour. A directed graph contains an Euler tour if and only if for every node, the in-degree of the node (the number of edges going to the node) is the same as the out-degree of the node (the number of edges leaving the node). For an undirected graph, an Euler tour exists if for every node, the degree of the node is even. After it is determined that a graph contains an Euler tour, an algorithm can be applied to find it.

The graph G_{12} in figure 2.7 does contain an Euler tour. Starting at vertex a, the Euler tour is [e_1, e_2, e_7, e_{12}, e_{11}, e_6, e_3, e_5, e_9, e_{10}, e_8, e_4].

2.4 Advanced Graph Problems

Advanced graph problems are the problems that are known to be NP for graphs in general. As such, a search for the optimal solution requires exponential time. Problems

that fall under this category include finding the maximum clique, the maximum independent set, the minimum coloring and the minimum clique cover.

2.4.1 Maximum Clique Problem

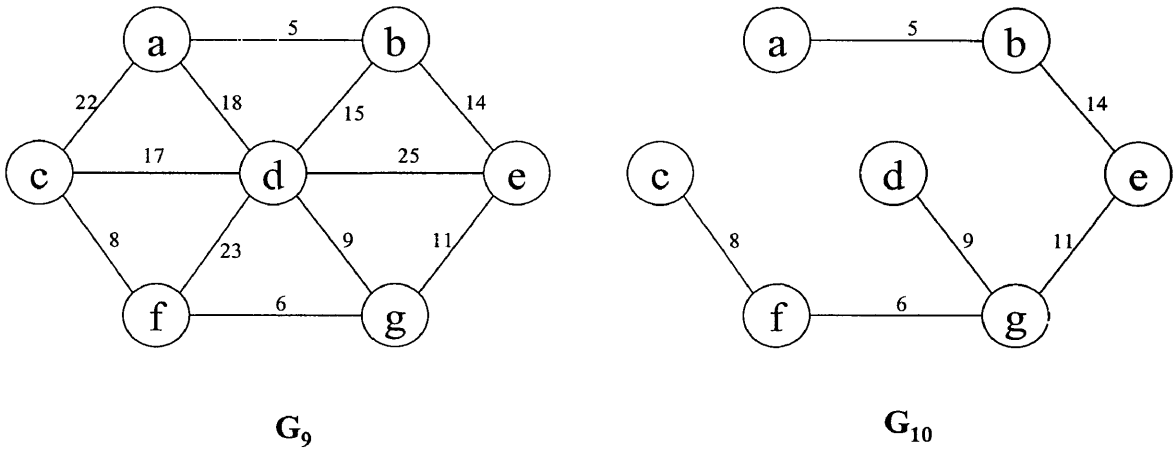


Figure 2.8 – Weighted Graph and Minimum Spanning Tree

A **clique** is a subgraph in which there exists an edge between every pair of vertices. A **maximal clique** for a given graph G is a clique that is not a subset of some larger clique in G. For example, in the graph G_9, the clique $\{a,c\}$ is not maximal because it is part of the larger clique $\{a,c,d\}$. A **maximum clique** is the largest possible clique in a given graph. The size of the maximum clique for a graph G is denoted as $\omega(G)$. Graph G_9 in the above figure contains a maximum clique containing the vertices a,c,d, so $\omega(G_9)=3$.

The maximum clique problem takes exponential time to solve for general graphs. Faster algorithms exist for specific types of graphs (see chapter 3). For other types of graphs, some heuristics can be used to allow for a near optimum solution in polynomial time.

2.4.2 Maximum Independent Set

An **independent set** (also referred to as a stable set) for a graph G is a set of vertices in which for every pair of vertices, no edge exists in G that connects the two vertices. The **maximum independent** set is the independent set with the largest number of vertices for a given graph. The size of the maximum independent set of a graph G is denoted as $\alpha(G)$. A maximum independent set in graph G_9 on page 25 is $\{a,e,f\}$, so $\alpha(G_9) = 3$.

The maximum independent set problem also takes exponential time to solve. It is equivalent to the maximum clique in the graph's complement. As such, this problem can be solved using the algorithms and heuristics design for finding the maximum independent set.

2.4.3 Vertex Cover

A **vertex cover** is a set of vertices such that every edge in the graph is incident to a vertex in the set. That is, for every edge (u,v) in the graph, either vertex u or vertex v must be in the vertex cover set. The vertex cover problem is to find a vertex cover with the minimum number of vertices. The graph in figure 2.9 on page 27 has a minimum vertex cover of $\{1,2,3\}$. Every edge in the graph is connected to one of those three vertices.

The vertex cover problem has been proven to be NP-complete. This means that finding the optimal vertex cover for a given graph requires exponential time. However, heuristics do exist to find a near-optimal solution.

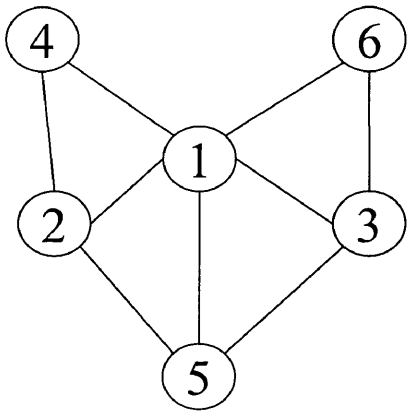


Figure 2.9 – Simple Graph

2.4.4 Minimum Coloring

Proper coloring of a given graph involves assigning a color to each vertex in the graph such that no two adjacent vertices have the same color. A minimum coloring of a graph (also referred to as the chromatic number of the graph) assigns the minimum possible number of different colors to the graph while maintaining proper coloring. The fewest number of colors needed to properly color a graph G is denoted as $\chi(G)$. The problem of finding the minimum coloring for a general graph is NP-complete.

At least three colors are required for the graph in figure 2.9, so $\chi(G)=3$. The minimum coloring would be to assign the first color to vertex 1, the second color to vertices 4,5, and 6, and the third color would be assigned to vertices 2 and 3.

2.4.5 Clique Cover

The clique cover problem is similar to the vertex cover problem. An optimal clique cover contains the minimum number of cliques such that every vertex in the graph is included in at least one of the cliques. For a given graph G, the size of the clique cover is denoted as $\kappa(G)$. The clique cover problem is also an NP-complete problem.

For the graph in figure 2.9, the minimum clique cover is three, so $\kappa(G)=3$. The cliques are {1,3,6}, {2,5}, and {4}.

2.5 Types of Graphs

A graph can be classified based on the characteristics and properties that it possesses. The advantage of knowing that a given graph belongs to a certain class is that often, a more efficient algorithm can be used. So by classifying a graph, problems dealing with the graph can be solved in less time. Many of the problems that are hard to solve for general graphs turn out to be easy for certain classes of graphs. For example, finding the maximum clique in a large graph can take days or even years to compute. However, if the graph is known belong to one of the classes of perfect graphs, then it can be solved in a matter of seconds.

2.5.1 Perfect Graphs

Many of the problems discussed in section 2.2 that are NP-complete in general graphs can be solved in polynomial time for the class of graphs known as perfect graphs.

Perfect graphs are a large class of graphs in which the maximum clique size is equal to the minimum proper node coloring for the graph and all induced subgraphs. Perfect graphs also have the property that the size of the maximum independent set is equal to the minimum clique covers of the graph all of its induced subgraphs. The complement of a perfect graph is also perfect.

Because of the properties of perfect graphs, more optimal algorithms are available to solve graph problems such as finding the minimum proper node coloring for the graph. There are several subclasses of perfect graphs. The subclasses are comparability graphs, co-comparability graphs, triangulated graphs, co-triangulated graphs, split graphs, permutation graphs, interval graphs, and tolerance graphs.

Figure 2.10 illustrates the relationship among the classes of perfect graphs. The class of permutation graphs includes all graphs that are both comparability graphs and co-comparability graphs. Intervals graphs are both triangulated graphs and co-comparability graphs. Split graphs are both triangulated and co-triangulated graphs. Each of these classes of perfect graphs is discussed in the following sections.

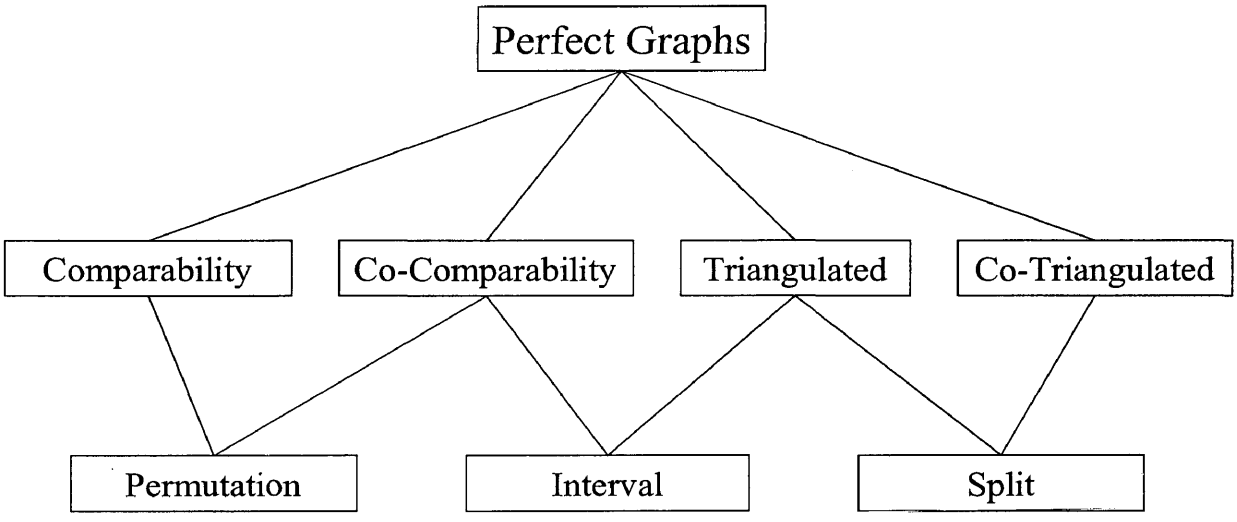


Figure 2.10 – Classes of Perfect Graphs

2.5.1.1 Triangulated Graphs

The first class of graphs that was recognized as being perfect was the class of triangulated graphs. A graph is a triangulated graph (also called a chordal graph) if it contains no chordless cycles of size four or larger. A chord is an edge that connects two non-sequential nodes in a cycle. The following figure illustrates a cycle that contains a chord. In this graph, the nodes *a*, *b*, *c*, and *d* form a cycle of length 4. The edge *ac* is a chord in that cycle because it connects two non-sequential nodes in the cycle (*a* and *c*). So this graph is a triangulated graph because every cycle of size four or larger contains a chord.

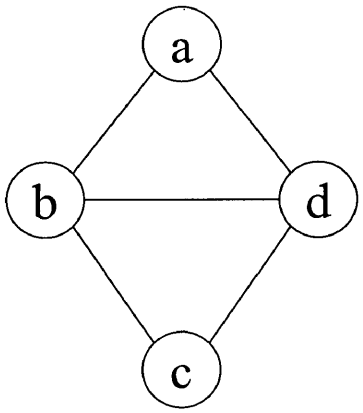


Figure 2.11 – Triangulated Graph

2.5.1.2 Comparability Graphs

A comparability graph is an undirected graph (the edges are not directed) that has a transitive orientation. As previously described, a transitive orientation is achieved by

assigning a direction to each edge in the graph such that if there is an edge from any node n_1 to a node n_2, and an edge from node n_2 to node n_3, then there must exist an edge from node n_1 to n_3.

The following figure shows a transitive orientation of a comparability graph.

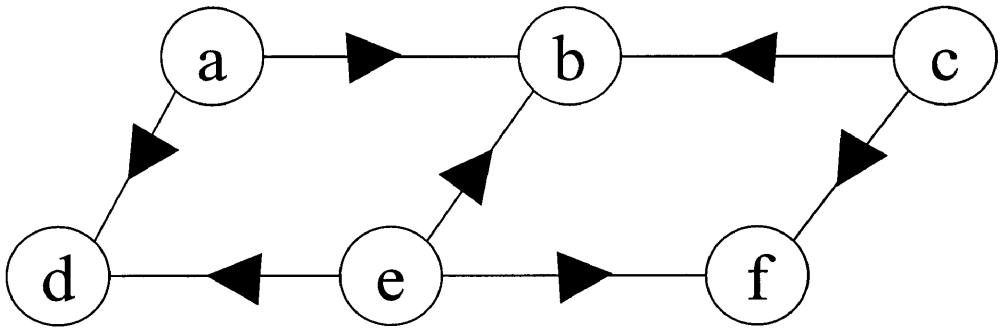


Figure 2.12 – Transitive Orientation

2.5.1.3 Split Graphs

A split graph is a graph where the nodes can be split up in to two groups. The first group is an independent set. That is, there are no edges connecting any two nodes in the first group. The second group of nodes forms a clique. In a clique there is an edge between every pair of nodes. The following figure illustrates a split graph in which the vertices $\{b,c,f,g\}$ form a clique, and the vertices $\{a,d,e,h\}$ form an independent set.

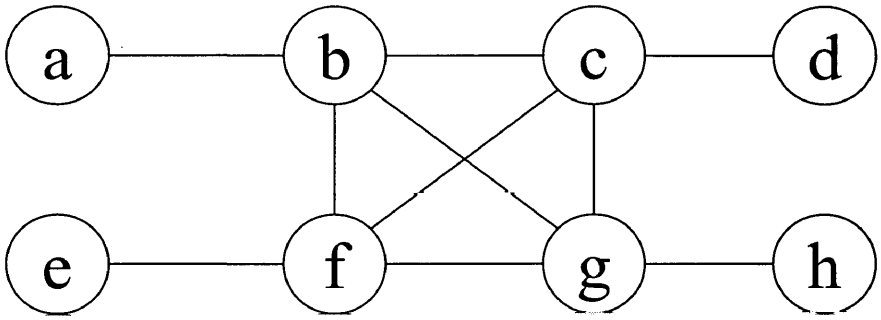


Figure 2.13 – Split Graph

Split graphs are graphs that are both triangulated graphs and co-triangulated graphs.

2.5.1.4 Permutation Graphs

A permutation graph is a graph that is both a comparability graph and a co-comparability graph. A permutation graph is so named because it represents a permutation, or change in the order, of a list of elements. If the order of two elements changes in relation to each other, an edge is drawn between those two nodes. Permutation graphs have important uses in VLSI design.

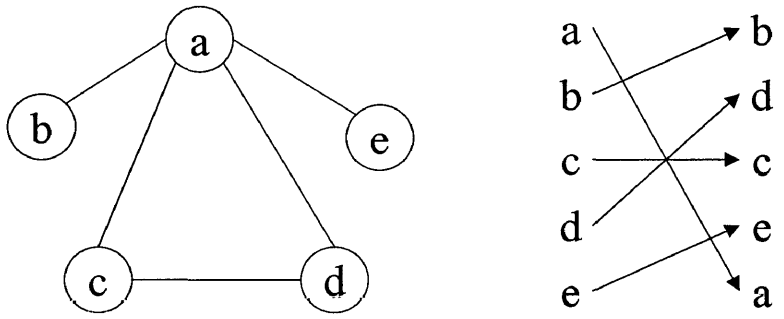


Figure 2.14 – Permutation Graph

In the figure above, the order of the elements [a,b,c,d,e] is changed to [b,d,c,e,a]. The resulting permutation graph is also shown in the above figure. In this example, since the order of elements a and b change, an edge is drawn between those two nodes.

2.5.1.5 Interval Graphs

Another class of graphs is known as interval graphs. Interval graphs represent conflicting intervals on a real line. Each node represents an interval. An edge is drawn between two nodes if their intervals overlap.

The case in chapter 1 of assigning classes to classrooms is an example of an interval graph. The intervals and resultant graph are shown in the following illustration.

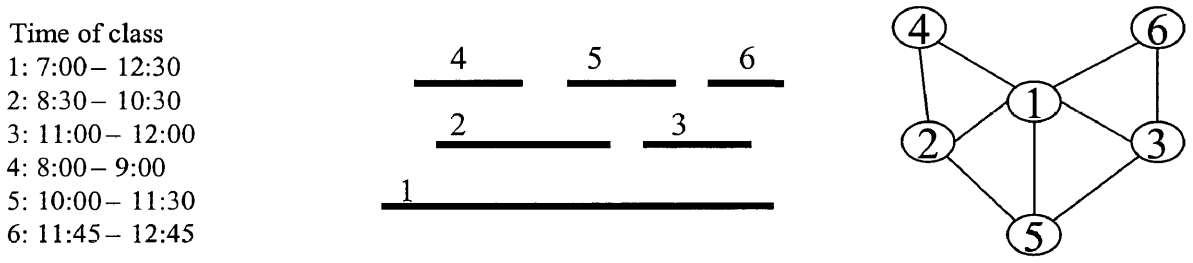


Figure 2.15 – Interval Graph

2.5.2 Tolerance Graphs

The previous classes of graphs are often to too strict to be applied to some real world problems. One example of this is the interval graph. In an interval graph, two intervals conflict even if they intersect by the most minute amount. However, in the real world, an overlap of a small amount is sometimes acceptable. A tolerance graph is a generalization of the interval graph that accommodates this factor.

The class of tolerance graphs can be defined as a generalization of the interval graph in which each interval has a tolerance value. Two intervals would only conflict if the size of their intersection is equal to or greater than the minimum of the two tolerances. That is, interval I_1 and I_2 conflict if the tolerance of I_1 is t_1 and the tolerance of I_2 is t_2 and the intersection of I_1 and I_2 is at least $\min(t_1,t_2)$. The following figure illustrates a set of intervals with tolerances and its associated tolerance graph.

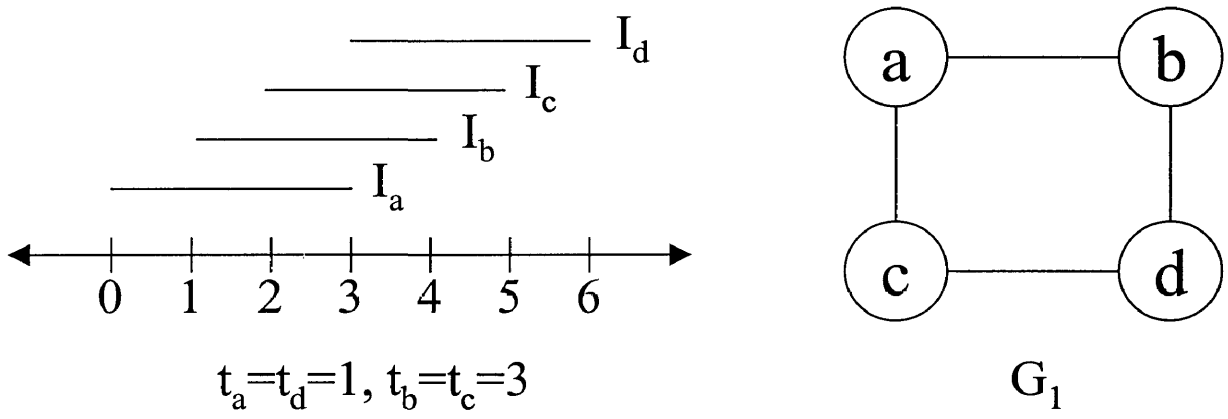


Figure 2.16 – Tolerance Graph

In the above example, the intervals *a* and *d* have a tolerance of 1, and the intervals *b* and *c* have a tolerance of 3. Even though the intervals *b* and *c* overlap, they do not conflict because the size of their overlap is only 2 and they both have a tolerance of three. No edge is drawn between nodes *b* and *c* since their intervals do not conflict.

Tolerance graphs allow for variations that occur in real-world problems. An example of this would be the assignment of workers to positions in an assembly line. For this example, only a single worker can be assigned to each position at a time, so no two employees should be scheduled to work at the same position at the same time. However, if the workers also have some administrative tasks that must be done away from their station, the time that it takes to perform those additional tasks would be their tolerance. So two workers that work at the same station could be scheduled for overlapping times since one could be at the station and the other could be performing his or her necessary administrative tasks.

If the tolerance of each interval is restricted to be no more than the length of the interval, the resultant class is known as **bounded tolerance graphs**. If any tolerance values are greater than the length of the corresponding interval, the tolerance graph is characterized as being an **unbounded tolerance graph**. Figure 2.17 illustrates an unbounded tolerance graph. In this example, the tolerance of the interval d is larger than the intervals size.

Chapter 3 includes an overview of previous research into tolerance graphs. Chapter 6 includes work in identifying whether a given graph belongs to the class of bounded tolerance graphs.

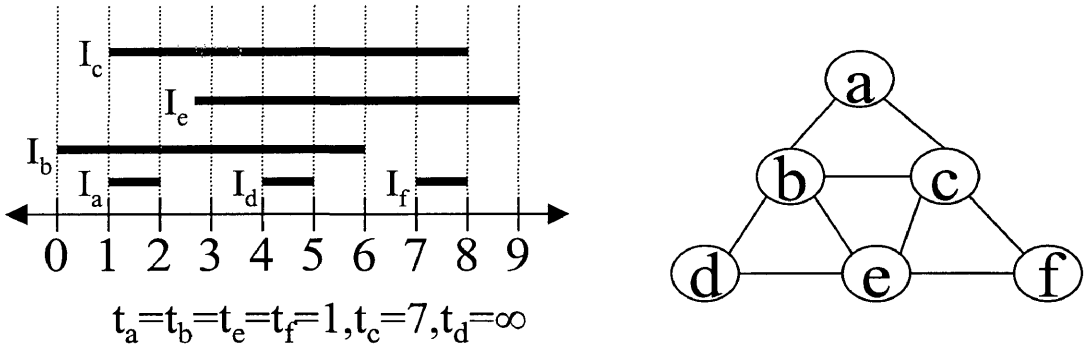


Figure 2.17 - Unbounded Tolerance Graph

The following figure illustrates how the classes of bounded and unbounded tolerance graphs relate to the previously described classes of perfect graphs. All interval

graphs and all permutation graphs are also bounded tolerance graphs. All bounded tolerance graphs are cocomparability graphs.

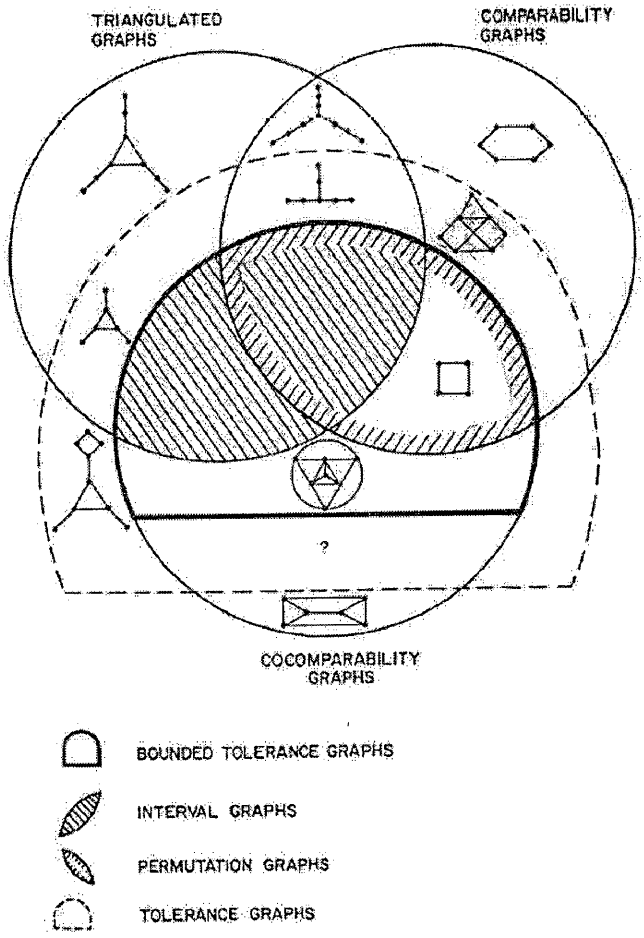


Figure 2.18 – Tolerance Graphs and Other Graphs

Chapter 3 Previous Research

This chapter includes an examination of previous research in the field of graph theory. Major aspects of graph theory to be covered include the class of perfect graphs and tolerance graphs. The discovery of perfect graphs was revolutionary in that a significant number of problems that are known to be NP-hard in general graphs were found to be polynomial in the class of perfect graphs. Next, the research in tolerance graphs will be covered to include its introduction in 1982 by Martin Golumbic and Clyde Monma. Finally, the application developed for this thesis will be compared to other packages available.

3.1 History of Graph Theory

In 1736, Euler came up with the graph as a model for a problem he was working on which he called the seven bridges problem. In this problem, seven bridges connected four pieces of land. Euler wanted to find a way to cross each bridge exactly one time and return to the point where he started. He created a diagram in which each point represented a piece of land and the bridges were represented as lines between those points. The solution to this problem was found by Fleury and is covered in chapter 4. In subsequent years the use of graph theory was applied to the fields of chemistry, electrical engineering, and social science.

Beginning in the 1960s graph theory has been applied to computers. This application provided a means for the computer to solve a wide variety of problems. The process of solving these problems begins with first modeling the problem as a graph. Once the graph is created a corresponding graph algorithm is found that which will solve the problem. Then the computer takes the graph and applies the graph algorithm to it to produce the solution.

3.2 Perfect Graphs

As previously stated, many of the problems that require exponential time to solve in general graphs can be solved in polynomial time for certain classes of graphs. In this respect, one of the most important classes of graphs is the class of perfect graphs. This class includes a large portion of the graphs that represent real world problems. Many significant problems that require exponential time in general graphs can be solved in polynomial time for perfect graphs. These include the maximum clique problem, the maximum independent set problem, the minimum coloring problem, and the clique cover problems introduced in section 2.4.

Extensive research has been applied to the class of perfect graphs since it was discovered by Claude Berge in 1960 [GM80]. Berge defined perfect graphs as the graphs that satisfy the following two properties:

$\omega(G) = \chi(G)$ The size of the largest clique is equal to the minimum proper coloring.

$\kappa(G) = \alpha(G)$ The size of the largest independent set is equal to the minimum clique cover.

Lázló Lovász proved in 1972 that the two above properties are equivalent [GM80]. Because of this, proving that one of the two properties is true for a class of graphs is sufficient to classify the graph as perfect.

Among other research in perfect graphs, of particular importance are the graph problems that are NP for general graphs, but are polynomial for the classes of perfect graphs. The solutions to these problems are examined the following sections.

3.2.1 Triangulated and Co-Triangulated Graphs

Triangulated graphs were proven by Hajnal, Surányi, and Berge to be one of the first classes of perfect graphs [GM80]. As stated in chapter 2, a triangulated graph is a graph in which every cycle of length greater than 3 contains a chord. A graph can be recognized as a triangulated graph through an algorithm designed by D. R. Fulkerson and O. A. Gross in 1965 [GM80]. This algorithm uses the identification of simplicial vertices. A vertex is simplicial if its adjacency set induces a complete subgraph. G. A. Dirvac had proved in 1961 that every triangulated graph contains a simplicial vertex.

The Fulkerson-Gross algorithm repeatedly finds a simplicial vertex and removes it from the graph. This process repeats until either all vertices have be removed from the graph, in which case the graph is triangulated, or no simplicial vertex can be found, in which case the graph is not triangulated.

As simplicial vertices are identified and removed from the graph, they are added to an elimination scheme. For a triangulated graph, the elimination scheme includes all vertices and is therefore a perfect elimination scheme. This elimination scheme is useful in finding a polynomial solution to some problems that are NP for general graphs.

For general graphs, finding the maximum clique involves finding all maximal cliques and picking the one that has the maximum size. This solution requires exponential time. For triangulated graphs, the elimination scheme can be used to find the maximum clique in polynomial time. Ford and Gross proved in 1965 that every maximal clique is in the form $v_i \cup X_v$ where vertex v_i is a vertex and X_v contains all other vertices adjacent to v_i that follow it in the perfect elimination scheme. In other words, if the perfect elimination scheme of a graph is $[v_1,v_4,v_3,v_5,v_2]$, then the maximal clique containing v_1 contains the vertex v_1 and the vertices in $\{v_4,v_3,v_5,v_2\}$ that are adjacent to v_1. By using this property, the maximal cliques can be found in polynomial time. Then by comparing the sizes of the maximal cliques, the maximum clique can be easily found.

The maximum independent set is also found using the perfect elimination scheme. In this algorithm, the independent set is formed by iterating through the elimination scheme (from beginning to end) and adding the vertex to the set if it is not adjacent to any of the other vertices already in the set. The resultant set is the maximum independent set of the triangulated graph.

By finding the complement of the graph, co-triangulated graphs can utilize the recognition algorithm defined for triangulated graphs. In addition, the maximum clique and independent sets can be found using the same technique.

3.2.2 Comparability and Co-comparability Graphs

Comparability graphs (also called transitively orientable graphs) are a class of perfect graphs that have a transitive orientation. In other words, the edges of a comparability graph can be directed such that if the edges (x,y) and (y,z) exist for any x, y, and z, then the edge (x,z) must exist. The recognition of a comparability graph involves attempting to find a transitive orientation of the graph through implication classes. If the undirected edges (x,y) and (y,z) exist, but (x,z) does not exist, then an orientation of (x,y) would force the orientation of (y,z) in the opposite direction. An implication class is made up of an oriented edge and all other edges that are forced by that edge. The recognition algorithm repeatedly finds an unoriented edge and creates an implication class by orienting that edge and all edges forced by that orientation. If the

same edge is oriented in two different directions, then the graph is not a comparability graph.

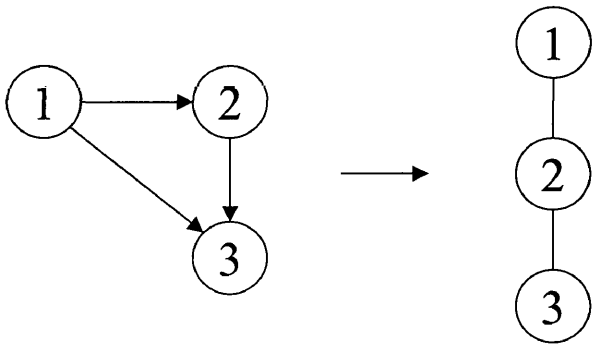


Figure 3.1 – Hasse Diagram

Once a transitive orientation is found for the comparability graph, it is converted into a Hasse Diagram to solve the maximum clique and minimum coloring problems. A Hasse Diagram is a representation of a directed acyclic graph in which the arrow heads are omitted (as the direction of the edge is implied to be from the higher node to the lower node) and the transitive edges are omitted (see figure 3.1).

After the Hasse Diagram for the comparability graph is found, it can be used to easily find the maximum clique. The maximum clique is simply the longest path from a node at the top of the diagram to a node at the bottom of the diagram. Similarly, the maximum node weighted clique is the path from a top node to a bottom node with maximum total node weights.

The minimum proper coloring of the comparability graph is equal to the number of levels in the Hasse Diagram as all of the nodes at the same level can be assigned the same coloring, and nodes at different levels must be assigned different coloring. The maximum independent set is equivalent to the width of the Hasse Diagram where the width is the maximum number of nodes that can occur at the same level in the diagram.

By finding the complement of the graph, the above approaches can also be used to recognize graphs that are cocomparability graphs and then find the maximum clique, the minimum coloring, and the maximum independent set.

3.3 Tolerance Graphs

In 1982, Martin C. Golumbic and Clyde L. Monma introduced the generalization of interval graphs known as tolerance graphs [GM82]. Tolerance graphs included an extra element of flexibility, which enabled them to represent more problems than interval graphs. As previously discussed, each interval is assigned a tolerance value. Two intervals do not intersect unless the size of their overlap is larger that one of their tolerances. This first paper included proof that a tolerance graph in which a constant value is assigned to every interval is also an interval graph. Also, a tolerance graph in which the tolerance of every interval is equal to the size of the interval is a permutation graph.

An interval has an unbounded tolerance if the tolerance is greater than the size of the interval. A bounded tolerance graph is a tolerance graph in which none of the tolerances are unbounded. Golumbic and Monma also proved that all bounded tolerance graphs are contained within the class of co-comparability graphs.

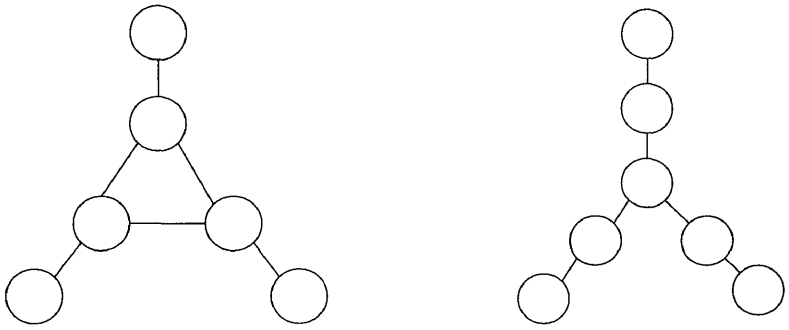


Figure 3.2 - Asteroidal Triple and Tree

It was later proven that tolerance graphs could not contain either a chordless cycle of length 5 or more or the complement of one [GM84]. The same paper also included the proof that tolerance graphs are a class of perfect graphs. This finding led to the polynomial time algorithms for clique, independent set, coloring, and clique cover [NM92]. However, these algorithms required that the tolerance representation be known for the tolerance graph.

There are several graphs that are prohibited from being a subgraph to the classes of tolerance graphs and bounded tolerance graphs. One of those is an asteroidal triple (see figure 3.2). If a graph contains an asteroidal triple as a subgraph, then the graph is

not a tolerance graph. Also, bounded tolerance graphs cannot contain the tree shown in figure 3.2 as an induced subgraph.

A subclass of tolerance graphs known as proper tolerance graphs have a tolerance representation in which no interval is properly contained within another interval. It has been proven that all proper tolerance graphs are also bounded tolerance graphs.

Several generalizations to tolerance graphs have been made including one by Kenneth Bogart and Ann Trenk in which each interval is allowed to have two tolerances: a left tolerance and a right tolerance [BT94]. These graphs are called bitolerance graphs, and the complements of these graphs are bitolerance orders.

Since tolerance graphs were introduced by Golumbic [GM82], several problems have remained unanswered. One such problem is that there is no known algorithm to determine if a given graph is a bounded tolerance graph. Also, if a graph is known to be a bounded tolerance graph, no polynomial algorithm exists for converting the tolerance graph into a tolerance representation. These two problems will be examined in this thesis.

3.4 Interface

Many applications are available that implement various algorithms. The most notable of which is a commercial program called Maple. The J-GAP application developed for this thesis has several advantages over previously available applications. First, most applications do not implement the advanced algorithms included in J-GAP like the algorithms developed for the classes of perfect graphs. Also, J-GAP is platform independent since it is written in Java. Finally, this application includes the capability to recognize most bounded tolerance graphs as discussed in chapter 6.

Chapter 4 Implementation of Graph Algorithms

A number of graph algorithms were implemented in this thesis. These algorithms can be grouped based on their ease of implementation. The first grouping is made up of algorithms that are available in a form that makes them easily implementable. These algorithms usually have accompanying pseudo-code included. The second grouping is algorithms that have been discussed, but are not in a format that makes them easily implementable. These algorithms were often discussed from a mathematical perspective rather than a computer science perspective. Finally, the third grouping of algorithms is the new tolerance graph algorithms that were developed in this thesis.

4.1 Abstract Data Types

All of the algorithms implemented in this thesis make use of several Abstract Data Types (ADTs) that were developed for this thesis to represent the nodes and edges in a graph and to represent the graph itself. All three abstract data types were implemented in the file *Graph.java*.

4.1.1 Graph Abstract Data Type

The first abstract data type defined in *Graph.java* is *Graph*. The *Graph* ADT is made up of several attributes that define the graph, and several methods that manipulate

the graph. The main component of this ADT is the *Nodes* attribute. This attribute is basically a set of nodes (as defined in the Node ADT). Other attributes that are defined are:

- isDirected (*boolean*) – True if the edges in the graph are directed edges.
- weightedNodes (*boolean*) – True if the nodes in the graph are weighted.
- weightedEdges (*boolean*) – True if the edges in the graph are weighted.
- showNodeText (*boolean*) – True if the *text* attribute of the nodes should be displayed in the drawing area. This capability is used by some algorithms to, for example, show the distance of each node from the starting node.
- showEdgeText (*boolean*) – True if the *text* attribute of the edges should be displayed in the drawing area. This capability is also used by some of the algorithms.
- filename (*String*) – This attribute contains the filename of the graph.

All of the methods for adding and removing nodes and edges are defined in this ADT:

- addNode – Adds a new node to the graph. Parameters to this function are the name of the node, the position it should lie in the drawing area, the node's weight, and the color that the node should be drawn in.

- removeNode – Removes an existing node from the graph. One parameter is required: either the name of the node, or the node object itself. This method returns false if the node was not in the graph.

- addEdge – Adds a new edge to the graph. Parameters to this method are the two nodes that the edge is incident to, and optionally the weight of the edge. This method returns false if either of the nodes does not exist in the graph.

- findEdge – Given two nodes, finds an edge object that connects them. The parameters of this method are the two nodes involved.

- adjacent – Given two nodes, returns true if they are adjacent.

- removeEdge – Removes an edge. The parameters of this method are the two nodes incident to the edge.

- makeCopy – Returns an exact copy of the current graph. Some algorithms remove nodes and edges from the graph as it finds the solution. These algorithms use this method to create a working copy of the current graph.

- makeGraphComplement – Returns a new graph that is the complement of the current graph.

4.1.2 Node Abstract Data Type

The second ADT, which is actually a subclass of the Graph class, is the Node ADT. This ADT contains several attributes that define the node, and a couple constructor methods. The main component of this ADT is the Edges attribute. This is simply a list of Edge objects. Other attributes that are defined are:

- position (*Point*) – The coordinates of where the center of this node should be drawn in the drawing area.
- name (*String*) – The name of the node. This is drawn in the middle on the node.
- weight (float) – The weight of the node. By default, this value is 1.0.
- color (*Color*) – The color that the node should be drawn in when it is not selected.
- text (*String*) – Extra text that should be written on the node when the graph's showNodeText attribute is set.
- indeg (*int*) – The number of edges coming into this node. This value is maintained by the methods in the Graph ADT.
- outdeg (*int*) – The number of edges leaving this node.
- alg (*Object*) – An undefined object that can be used by the algorithms. Most algorithms need to attach other attributes to each node in the graph. This is

accomplished by creating an object with all of those extra attributes and attaching that object to each node's *alg* attribute.

4.1.3 Edge Abstract Data Type

The final abstract data type defined in *Graph.java* is Edge. The Edge ADT is made up of several attributes that define the edge, and a few constructor methods. The attributes that are defined are:

- toNode (*Node*) – The node that this edge connects to.
- color (*Color*) – The color that this edge should be drawn in.
- weight (*float*) – The weight of this edge. By default, this value is 1.0.
- isDrawn (*boolean*) – True if this edge should be drawn in the drawing area. An undirected edge in a graph is made up of an edge going from the first node to the second, and another edge going from the second node to the first. Only one of these edges needs to be drawn in the drawing area.
- curveNumber (*int*) – If multiple edges exist between the same pair of nodes, each edge will have a unique curveNumber. This ensures that none of the

multiple edges will be drawn overtop of each other. If curveNumber is zero, the edge is drawn as a straight line.

- counterpart (*Edge*) – For undirected edges, this is a pointer to this edge's counterpart. Since an undirected edge is made up of two directed edges, this pointer is used so that both parts of an edge are deleted together.

- text (*String*) – Any label that should be written next to the edge if the graph's *showEdgeText* attribute is true.

- alg (*Object*) – A generic object that can be used by the algorithms. Similar to the *alg* attribute of the *Node* class, this allows each algorithm to attach miscellaneous attributes to each edge.

4.2 Easily Implementable Algorithms

The first grouping of algorithms implemented in this thesis include those that have already been implemented in pseudo-code or in a programming language. This makes them more easily implemented in this thesis. These algorithms include the Kruskal and Prim minimum spanning tree algorithms, Dijkstra and Bellman-Ford shortest path algorithms, Euler tour algorithms, and the Ford-Fulkerson maximum network flow algorithm.

4.2.1 Kruskal's Minimum Spanning Tree Algorithm

Kruskal's algorithm finds the minimum spanning tree for a connected edge-weighted undirected graph. The pseudo-code for this algorithm is shown below [CL90].

```
MST-KRUSKAL(G,w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3      do MAKE-SET(v)
4  sort the edges of E by nondecreasing weight w
5  for each edge (u,v) ∈ E, in order by nondecreasing weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u,v)}
8                 UNION(u,v)
9  return A
```

This algorithm takes as input the graph (G), made up of a set of vertices V[G] and edges E[G], and the edge weights (w). The algorithm builds and returns the set *A* containing the edges in the minimum spanning tree. Initially, the set *A* is empty, the edges are sorted based on weight, and each vertex is placed in a separate set. MST-Kruskal loops through the edges (from smallest weight to largest) and adds the edge if the endpoints of the edge are in different sets. As edges are added, the two sets containing

the endpoints of the edge are merged. When line 9 is reached, all of the vertices are in a single set, and the set *A* contains the minimum spanning tree.

This algorithm is implemented in the file *MinSpanTree.java*. This file includes the methods kruskal(), findSet(), and Union() which together implement Kruskal's minimum spanning tree algorithm.

4.2.2 Prim's Minimum Spanning Tree Algorithm

Prim's algorithm also finds the minimum spanning tree for a connected edge-weighted undirected graph. This algorithm starts with a tree containing a single vertex and then grows the tree until it contains all vertices. The pseudo-code for this algorithm is listed and described below [CL90].

```
MST-PRIM(G,w)
1  Q ← V[G]
2  for each u ∈ Q
3      do key[u] ← ∅
4  key[r] ← 0
5  π[r] ← NIL
6  while Q ≠ ∅
7      do u ← EXTRACT-MIN(Q)
8           for each v ∈ Adj[u]
9                do if v ∈ Q and w(u,v) < key[v]
10                    then π[v] ← u
```

```
11                              key[v] ← w(u,v)
```

Prim's algorithm takes as input the graph (G), the edge weights (w), and the root vertex (r). This algorithm assigns two values to each vertex. The first value is the key of the vertex. The key is the minimum weighted edge connecting a vertex to the tree. The second value is π. This is the parent of the vertex in the tree. Initially, every vertex is assigned a key of infinity. Then the root vertex is assigned a key of zero, and a parent of nil. This creates a tree that contains only the root vertex.

As vertices are added to the tree, the key and π values of the other vertices is updated. The priority queue Q contains all of the vertices not yet added to the tree. During each iteration, the vertex with minimum key is added to the tree. Upon completion, the algorithm creates the minimum spanning tree of the graph where the parent of a vertex u is $\pi(u)$.

This algorithm is also implemented in file *MinSpanTree.java*. This file includes the methods prim() and extractMin() which together implement Prim's algorithm.

4.2.3 Dijkstra's Single-Source Shortest-Paths Algorithm

Dijkstra's algorithm finds the shortest paths from a single vertex to all other vertices in a weighted directed graph that does not contain edges with negative weights. This algorithm and the Bellman-Ford algorithm use two other procedures shown here.

```
INITIALIZE-SINGLE-SOURCE(G,s)
1  for each vertex v ∈ V[G]
2       do    d[v] ← ∞
3             π[v] ← NIL
4  d[s] ← 0

RELAX(u,v,w)
1  if d[v] > d[u] + w(u,v)
2       then d[v] ← d[u] + w(u,v)
3            π[v] ← u
```

Initialize-Single-Source is used to initialize the graph by assigning two values to each vertex. The distance to a given vertex, d[v], is initialized to infinity for all vertices except the source vertex which is assigned a distance of zero. The predecessor of each vertex, π[v], is initialized to nil. The Relax procedure is used to "relax" each edge. This is a process whereby the distance from the source to vertex u and the weight of the edge (u,v) is compared to the current distance from the source to the vertex v. If a shorter path is found, then d[v] and π[v] are updated.

Now the pseudo-code for Dijkstra's algorithm is presented [CL90].

```
DIJKSTRA(G,W,S)
1   INITIALIZE-SINGLE-SOURCE(G,s)
2   S ← Ø
3   Q ← V[G]
4   while Q ≠ Ø
5       do u ← Extract-Min(Q)
6             S ← S ∪ {u}
7             for each vertex v ∈ Adj[u]
8                   do RELAX(u,v,w)
```

The input parameters to Dijkstra's algorithm are the graph, G, and the source vertex, s. The algorithm creates a set S that contains vertices for which the shortest path has been found, and a priority queue Q that contains the other vertices ordered by their d values. Repeatedly, the vertex in Q with minimum distance is extracted and added to the set S. As vertices are added to S, all of there edges are "relaxed." Upon completion, every vertex contains its distance from the source vertex, s, and the predecessor in the shortest path from s to the vertex.

This algorithm is implemented in the file Dijkstra.java. Methods available in this file include findShortestPaths(),hasNegativeEdges(), InitializeSingleSource(), extractMin(), and relax(). Due to the similarity between the Java code and the pseudo-code, the Java code does not need to be discussed.

4.2.4 Bellman-Ford Single-Source Shortest-Paths Algorithm

The second shortest paths algorithm implemented in this thesis is the Bellman-Ford algorithm. This algorithm also finds the shortest paths from a source node to all other nodes in a directed weighted graph. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm allows for negative-weighted edges. However, cycles that have a negative total edge weight are not allowed. If a negative cycle exists in the graph then no solution exists to the shortest path algorithm and this algorithm returns FALSE. This pseudo-code for the Bellman-Ford algorithm is as follows [CL90]:

```
BELLMAN-FORD(G,W,S)
1  INITIALIZE-SINGLE-SOURCE(G,s)
2  for i ← 1 to |V[G]] - 1
3       do for each edge (u,v) ∈ E[G]
4            do RELAX(u,v,w)
5  for each edge (u,v) ∈ E[G]
6       do if d[v] > d[u] + w(u,v)
7            then return FALSE
8  return TRUE
```

The Bellman-Ford algorithm relaxes each edge n-1 times where n is the number of vertices in the graph. If the graph does not contain any negative cycles then the shortest paths are found after line 4. Lines 5-8 test for a negative cycle by testing to see if any shorter paths can be found.

This algorithm is implemented in the file *BellmanFord.java* and is made up of the methods bf(),InitializeSingleSource(), and relax().

4.2.5 Euler Tour Algorithms

An Euler tour is a closed trail that starts and ends at the same vertex and includes all edges exactly one time. An algorithm is available which test a graph to determine if it contains an Euler tour. Another algorithm finds an Euler tour in a given graph.

Testing whether a graph contains an Euler tour is simple. An undirected graph is Eulerian if and only if every vertex has an even degree. A directed graph is Eulerian if and only if for every vertex v, the out degree of the vertex is equivalent to its in degree. That is, out-deg(v)=in-deg(v) for all v.

The process of finding the Euler tour can begin at any node in the graph. When deciding on the next edge in the tour, an edge (u,v) can be used if that is the only unused edge from the node or if there exists another path of unused edges from node v back to node u.

These Eulerian algorithms are implemented in the file *Euler.java*. The method hasEulerTour() tests whether a given graph contains an Euler tour. The method findEulerTour() labels the edges in a graph with its sequence number in the Euler tour.

4.2.6 Ford-Fulkerson Maximum Network Flow Algorithm

The Ford-Fulkerson algorithm finds the maximum network flow from a source vertex (denoted vertex s) to a destination vertex (denoted vertex t). The following pseudo-code finds the maximum flow along each edge in the network [CL90].

```
FORD-FULKERSON(G,W,S)
1  for each edge (u,v) ∈ E[G]
2       do    f[u,v] ← 0
3             f[v,u] ← 0
4  while there exists a path p from s to t
5       do    cf(p) ← min{cf(u,v) : (u,v) is in p}
6             for each edge (u,v) in p
7                   do    f[u,v] ← f[u,v] + cf(p)
8                         f[v,u] ← -f[u,v]
```

Each iteration of the algorithm creates a residual network where f represents the flow along an edge, and c_f represents the remaining capacity of the edge. Lines 1-3 initialize the flow along each edge to zero. Line 4 repeatedly finds a path from s to t where each edge in the path has a capacity larger than zero. The flow along each edge is then increased in lines 5-8.

This algorithm is implemented in the file *MaxFlow.java* and is initiated by calling the method FordFulkerson() with the graph and the starting and ending vertices. The

method initializeMaxFlow() is used internally to add the residual edges to the graph. Then the method findAugPath() is used to find a path from the starting vertex to the ending vertex. If a path is found, the method increasePathFlow() is called to increase the flow along that path.

4.3 Implementation of Advanced Algorithms

Not all known algorithms are available in a form that makes them easily implementable. Many algorithms exist which have been discussed in texts, but require a larger amount of work to implement in a computer programming language. Often, mathematicians rather than computer scientists developed these algorithms. They therefore make more use of diagrams and human intuition. Converting these into data structures and lines of code is often more complicated than it may initially appear.

The algorithms used in this thesis that fall within this grouping include PQ-trees and maximum matching and some of the Perfect Graph algorithms.

4.3.1 PQ-Trees

A large number of data structures were utilized in this thesis. One of which was a data structure introduced by Kellogg S. Booth and George S. Lueker known as the PQ-tree [BL76]. The PQ-tree is used to find permissible permutations of a set. This data

structure is useful in graph theory because it can be used to find an interval representation of an interval graph.

Converting an interval graph into an interval representation requires three steps. First, all maximal cliques must be found. Next, a valid ordering of those cliques is identified. This step requires that all cliques that contain a common vertex must be consecutive in the ordering. Finally, the list of ordered cliques is converted into the interval representation.

The first step is accomplished be a polynomial time algorithm that finds all maximal cliques in an interval graph, which has been implemented in this thesis. The second step is accomplished by utilizing the PQ-tree data structure. The final step numbers each maximal clique based on its position in the resultant clique ordering. Then each vertex is assigned an interval that begins with the first maximal clique that it is a member of and ends with the first maximal clique that it is not a part of.

A PQ-tree is a tree that contains three types of nodes: P-nodes, Q-nodes, and leaf nodes. A P-node is represented as a circle, and a Q-node is represented as a rectangle. Figure 4.1 contains an example of a PQ-tree. In this example, the root node is a Q-node and one of its children is a P-node.

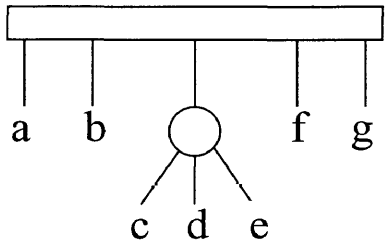


Figure 4.1 - PQ-Tree

The PQ-tree represents all permissible permutations of the set (represented as leaf nodes) because a couple of restrictions are made on how the children of P-nodes and Q-nodes can be reordered. The immediate children of a P-node can be reordered arbitrarily. The immediate children of a Q-node can be reversed.

In the above PQ-tree, the left to right ordering is [a,b,c,d,e,f,g]. Another possible ordering (by reversing the children of the Q-node) is [g,f,c,d,e,a,b]. However, the ordering [a,f,g,b,c,d,e] is not valid because the Q-node requires that the leaf nodes a and b must be consecutive.

Initially, all leaf nodes in a PQ-tree are the children of a root P-node. At this point any ordering of the leaf nodes is valid. Then restrictions are added to the tree requiring that a set of leaf nodes must be consecutive. As these restrictions are added, the structure of the PQ-tree is updated.

Booth and Lueker constructed an algorithm for implementing PQ-trees. This algorithm utilizes a total of 11 templates. Each template is made up of a pattern and its corresponding replacement. As a restriction is added to the PQ-tree, it must match one of the 11 template patterns if can be accommodated.

Booth and Lueker presented pseudo-code for two of the 11 templates. All 11 templates were implemented in Java for this thesis in the file *PQTree.java*.

4.3.2 Maximum Matching

As discussed previously, a matching is a subset of edges such that no two edges are adjacent. The maximum matching problem is to find the largest set of edges where none of the edges are adjacent. Alan Gibbons presented an algorithmic approach to solve this problem [GA85]. This approach makes use of augmenting paths.

For any matching M, an alternating path is defined as a path whose edges alternate between edges in M and edges not in M. An augmenting path is an alternating path whose endpoints are not incident to an edge in M.

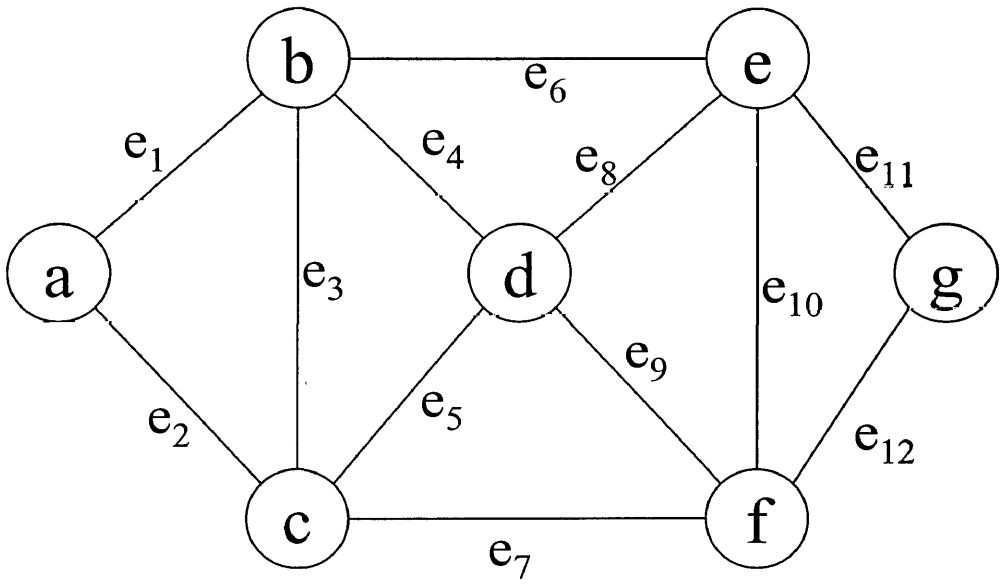


Figure 4.2 - Matching Example

In the above graph, a matching might be $\{e_3,e_{10}\}$. An alternating path for that matching is $[e_3,e_7,e_{10}]$ where the first and third edges are in the matching and the second edge is not. An augmenting path for the matching is $[e_1,e_3,e_5]$ where the first and last edges are not in the matching and the vertices e_1 and e_5 are not incident to the edges in the matching.

If an augmenting path can be found, then a larger matching exists. The larger matching is obtained by reversing the roles of the edges in the augmenting path. For example, in the above graph, the matching was $\{e_3,e_{10}\}$ and the augmenting path is $[e_1,e_3,e_5]$. By reversing the roles of the edges in the augmenting path (e_1,e_3,e_5), a new matching of $\{e_1,e_5,e_{10}\}$ is obtained.

The process of finding a maximum matching now becomes an iterative process of finding an augmenting path and the applying it to find a larger matching. The maximum matching has been found when no augmenting paths exist.

Alan Gibbons presented pseudo-code for finding augmenting paths in a general graph. This pseudo-code, which is listed below, proved difficult to implement. In it, odd cycles are called blossoms. These odd cycles present a problem because they contain two different paths between a pair of vertices. One path has an odd length and the other path has an even length. When a blossom is encountered in this pseudo-code, it is placed onto a stack and replaced in the graph with a single pseudo-vertex. Once an augmenting path is found the blossom is restored.

Gibbons' algorithm also detects Hungarian trees. These are vertices explored by the M-augmenting path search procedure that cannot possibly be in an M-augmenting path. Therefore they are also replaced with a pseudo-vertex to prevent them from being explored again. As with blossoms, Hungarian trees are restored when an augmenting path is found.

The following is the pseudo-code presented by Gibbons [GA85].

```
    MAXIMUM-MATCHING(G)
1.  M ← 0
2.  an augmenting path exists ← true
```

```
    3.  while an augmenting path exists do
            begin
    4.      determine the free vertices {v_i} with respect to M
    5.      for each v_i do
                begin
    6.          empty the stack
    7.          deem each edge of G to be unexplored
    8.          T ← v_i, label v_i outer
L:  9.          MAPS(G)
B:  10.         Place the Blossom found on the stack, shrink it
                    in G and label the resultant vertex outer
                    and if it contains the root label it free.
                    goto L.
H:  11.         Place the Hungarian tree found on the stack and
                    remove it from G.
                end
    12.     Output M.
    13.     an augmenting path exists ← false
    14.     goto S
A:  15.     Identify the M-augmenting path P∈T. Empty the
                stack, expanding G with each popped item.
                If the item is a blossom corresponding to a
                pseudo-vertex on P, interpolate into P the
                appropriate even-length section of the
                blossom.
    16.     Augment M by interchanging the edge roles in P
S:  17.     end


    MAPS(G)
    1.  Choose an outer vertex x∈T and some edge (x,y) not
        previously explored.  Deem (x,y) to have been
        explored.  If no such edge exists goto H.
```

```
2.   If y is free and unlabelled, add (x,y) to T.
     goto A.
3.   if y is outer add (x,y) to T. goto B.
4.   if y is inner goto 1
5.   Let (y,z) be the edge in M with the end-point y.  Add
     (x,y) and (y,z) to T.  Label y inner and z outer.
     goto 1.
```

The *MAPS(G)* procedure searches for an M-augmenting path. If one is found, it jumps to the line labeled A where the new matching is found. If it finds a blossom, it goes to the line labeled B where it is placed on the stack and replaced with a pseudo-vertex in the graph. If it finds a Hungarian tree it is removed from the graph and placed on the stack.

The main *MAXIMUM-MATCHING* procedure repeats lines 3-17 until no more augmenting paths can be found. In finding each augmenting path, line 4 finds all vertices not in the current matching. Lines 5-11 search for augmenting paths that begin at one of the free vertices by calling the *MAPS(G)* procedure. If an augmenting path is found, lines 15 and 16 create the new matching. If no augmenting paths are found, lines 12-14 output the final maximum matching and exit.

4.3.3 Perfect Graphs

Algorithms for classes of perfect graphs have been developed for solving the problems of minimum coloring, maximum clique, maximum independent set, and minimum clique-cover. For triangulated and co-triangulated graphs, the algorithms for the four problems make use of the graph's perfect elimination scheme. For comparability and co-comparability graphs these algorithms make use of the graph's Hasse diagram.

Finding just the size of the minimum coloring, maximum clique, maximum independent set, and minimum clique-cover is more simple because for all perfect graphs, the size of the minimum coloring is equal to the size of the maximum clique and the size of the maximum independent set is equal to the size of the minimum clique-cover. This would mean that only half as many algorithms would have to be implemented. However, this approach would find the size of the minimum coloring, not the color that should be assigned to each vertex. For this reason, separate algorithms were implemented for each of the four problems.

4.3.3.1 Triangulated Graphs

Five different algorithms were implemented for the class of triangulated graphs. Those algorithms are characterization, minimum proper coloring, maximum clique, maximum independent set, and minimum clique cover.

4.3.3.1.1 Characterization of Triangulated Graphs

As stated in chapter 3, a graph is a triangulated graph if it does not contain any chordless cycles of length 4 or larger. The algorithm for identifying whether a graph triangulated successively finds and removes simplicial nodes from the graph. A simplicial node is a node in which all of the nodes that are adjacent to it form a clique. It has been proven that every triangulated graph contains at least two simplicial nodes [GM80]. For a triangulated graph, a simplicial node can be found and removed repeatedly until all nodes in the graph have been removed. The ordering of the simplicial nodes forms what is called an elimination scheme. The elimination scheme of a graph G is denoted $\sigma(G)$.

Finding a perfect elimination scheme (an elimination scheme that includes all nodes in the graph) is sufficient for proving that a graph is triangulated. If at some point, a simplicial node cannot be found, then the graph is not a triangulated graph. The pseudo-code for identifying a tolerance graph is now presented [GM80].

```
   IS-TRIANGULATED(G)
1  σ ← empty list
2  while G ≠ ∅
3       do s ← FIND-SIMPLICIAL-NODE(G)
4          if s = NIL then return FALSE
5          add s to σ
```

```
6          remove s from G
7  return TRUE
```

The process of finding a simplicial node simply loops through the nodes and test to see if all of the neighbors of a node are pair-wise adjacent.

4.3.3.1.2 Maximum Clique in a Triangulated Graph

The elimination scheme found in the previous section can be use to find the maximum clique in a triangulated graph in polynomial time. First the maximal cliques in the graph are found by stepping through the elimination scheme. Then the largest of the maximal cliques is returned as the maximum clique.

The maximal cliques can be found through the following algorithm [GM80]:

```
   FIND-MAXIMAL-CLIQUES(G, σ)
1  for i ← 1 to |σ|
2      do u ← σ(i)
3         Clique ← {u}
4         for j ← i+1 to |σ|
5             do v ← σ(j)
6                if v ∈ adj(u) then Clique ← Clique ∪ {v}
7         print Clique
```

4.3.3.1.3 Minimum Coloring in a Triangulated Graph

The minimum coloring algorithm also takes advantage of the elimination scheme. This algorithm starts at the end of the elimination scheme and assigns the first valid color to a node that does not conflict with any other nodes that have already been assigned a color. The pseudo-code for this algorithm will now be presented [GM80].

```
   FIND-MINIMUM-COLORING(G, σ)
1  for i ← |σ| to 1 step -1
2     do u ← σ(i)
3        color(u) ← 0
4        valid ← FALSE
5        while valid = FALSE
6           do color(u) ← color(u) + 1
7              valid ← TRUE
8              for j ← i+1 to |σ|
9                 do v ← σ(j)
10                   if v ∈ adj(u) and color(u) = color(v)
11                      then valid = FALSE
```

4.3.3.1.4 Maximum Independent Set in a Triangulated Graph

The maximum independent set algorithm works similar to the maximum clique algorithm for triangulated graphs. It steps through the elimination set and generates maximal independent sets. The largest maximal independent set is then returned as the maximum independent set. The pseudo-code for this algorithm will now be presented [GM80].

```
   FIND-MAXIMAL-IND-SET(G, σ)
1  for i ← 1 to |σ|
2     do u ← σ(i)
3        Indset ← {u}
4        for j ← i+1 to |σ|
5           do v ← σ(j)
6              if v ∉ adj(u) then Indset ← Indset ∪ {v}
7                  print Indset
```

4.3.3.1.5 Minimum Clique Cover in a Triangulated Graph

The minimum clique cover works similar to the maximum clique algorithm for triangulated graphs. First the maximal cliques in the graph are found by stepping through the elimination scheme. Then the largest of the maximal cliques is returned as the maximum clique.

The maximal cliques can be found through the following algorithm [GM80]:

```
   FIND-MAXIMAL-CLIQUES(G, σ)
1  c ← 0
2  for i ← 1 to |σ|
3     do u ← σ(i)
4        if clique(u) is not set
5        then c ← c + 1
6           clique(u) ← c
7           for j ← i+1 to |σ|
8              do v ← σ(j)
```

```
9                    if v ∈ adj(u) then clique(v) ← c
```

4.3.3.2 Comparability Graphs

The same five different algorithms were implemented for the class of comparability graphs. Those algorithms are characterization, minimum proper coloring, maximum clique, maximum independent set, and minimum clique cover.

4.3.3.2.1 Characterization of Comparability Graphs

A comparability graph is defined as a graph that is transitively orientable. If the graph can be transitively oriented, then it is a comparability graph. The graph is oriented by first finding an unoriented edge and assigning it an arbitrary orientation. This orientation may force the orientation of other edges in the graph. This process is repeated until all edges have been oriented. If a forced orientation conflicts with a previous orientation then the graph is not a comparability graph. The pseudo-code for this algorithm is now presented [GM80].

```
   IS-COMPARABILITY(G, σ)
1  for each edge e ∈ E
2     do if e is unoriented
3        then ORIENT(e)
4             if not FORCE-REL(G,e)
5                then return FALSE
6  return TRUE
```

4.3.3.2.2 Maximum Clique in a Comparability Graph

Once the Hasse diagram has been found for a comparability graph, the maximum clique can easily be found. The number of levels of the Hasse diagram is equivalent to the size of the maximum clique. Finding the nodes in the maximum clique is equivalent to finding a path from the top level of the Hasse diagram to the bottom level.

4.3.3.2.3 Minimum Clique Coloring in a Comparability Graph

The minimum coloring of a comparability graph is also equivalent to the number of levels in the graph's Hasse diagram. Nodes on different levels of the Hasse diagram must be assigned different colors. Nodes on the same level of the Hasse diagram can be assigned the same color.

The algorithm for finding the minimum coloring first finds the Hasse diagram for the graph. Then every node in the first level of the diagram is assigned the first color. The nodes on the second level are assigned the second color, as so on.

4.3.3.2.4 Maximum Independent Set in a Comparability Graph

The size of the maximum independent set in a comparability graph is equivalent to the width of the graph's Hasse diagram. All of the nodes on any one level of a Hasse diagram are independent. The widest level of the Hasse diagram is the level that contains

the most nodes. The maximum independent set is made up of the nodes in the widest level of the Hasse diagram.

4.4 Summary of Algorithms

Each of the algorithms listed in this chapter vary in terms of complexity and amount of effort to implement. All of these algorithms were implemented in this thesis. They can be selected from the "Algorithms" drop-down menu from within the accompanying application.

Chapter 5 J-GAP Application

5.1 Overview

A Java-based Graph Algorithms Program (J-GAP) was developed for this thesis. It implements all of the graph algorithms discussed in the preceding chapters. The graphical user interface allows a user to create a graph by adding nodes and edges and then selecting from the available graph algorithms.

5.2 Installation

The J-GAP application can be installed on any platform that supports Java version 1.2 or higher. Installation can be broken down into three major steps: installing Java, installing this application, and starting the application.

5.2.1 Installing Java

This application requires a Java Run-Time Environment (JRE) of version 1.2 or higher. The JRE can be downloaded separately or as part of the Java Software Development Kit (SDK), which includes the Java compiler and other utilities. For Microsoft Windows, Linux, and Sun Solaris platforms, the JRE and the JDK can be downloaded from the following website:

· http://java.sun.com/products/index.html

Java is available for other platforms at the following site:

http://java.sun.com/cgi-bin/java-ports.cgi

After downloading the JRE or JDK, install it according to the instructions included on the web site.

Note: After installation, ensure that the java/bin directory is included in the search path. Also, if an older version of Java was previously installed, make sure that the CLASSPATH variable is *not* defined.

5.2.2 Installing this application

This application consists of a single jar (Java archive) file: *jgap.jar*. It is available at http://Mike.Hazelwood.com/graph.

To install this application, simply download and place the *jgap.jar* file in an empty directory.

5.2.3 Starting the application

To start the application in Microsoft Windows, double-click on the jgap.jar file from within the Windows Explorer. For other platforms perform the following:

- Get in the directory where the *jgap.jar* file is located.
- Enter the following command

```
java -jar jgap.jar
```

If the application fails to start, try the following troubleshooting steps:

- Ensure that version 1.2 or higher of Java is correctly installed by executing the following command:

```
java -version
```

- Ensure that the jar file is undamaged by running the following command and watching for errors:

```
jar -tvf jgap.jar
```

5.3 Interface characteristics

As shown in figure 5.1, the graphical user interface is made up of three parts: a drop down menu, a drawing area, and a status bar. By default, the program runs in a 600x400 window. However, it is easily resizable on most platforms by dragging on a corner of the window.

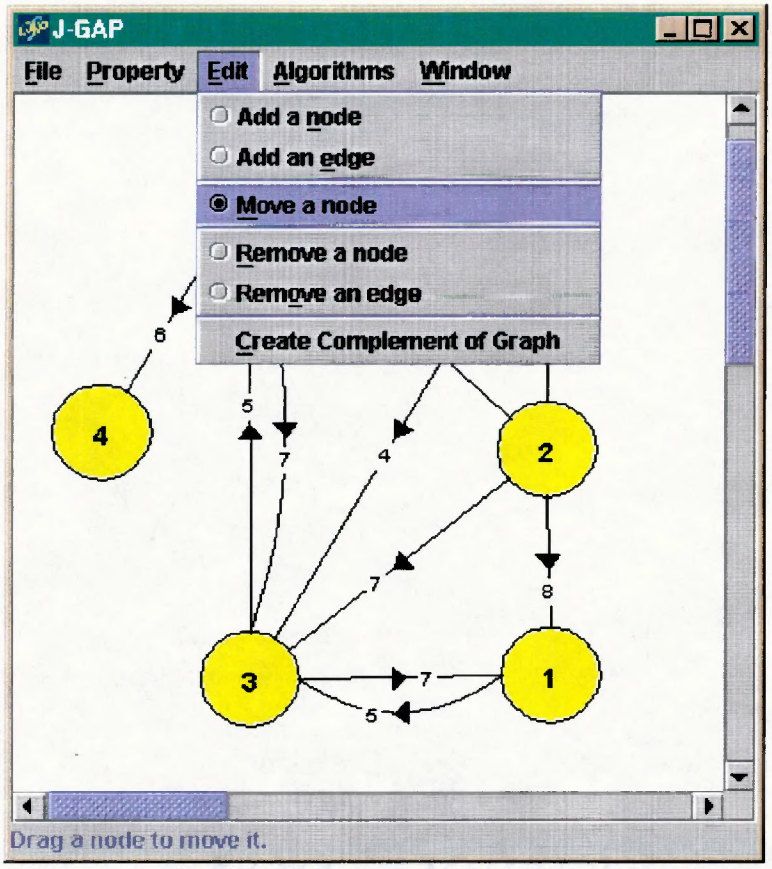


Figure 5.1 – Application Interface

The drawing window is in a scrollable pane and is 1280 x 1024 pixels in size. It is in this window that a graph can be created and the results of an algorithm are visualized. Depending on the action selected in the drop-down menu, one or two nodes may need to be selected. A single node is selected by simply clicking on it with the mouse. Some algorithms require the user to select two nodes like the source and sink in a network flow. Clicking on the first node, holding down the mouse button and dragging to the second node, and then releasing the mouse button accomplish this.

Below the drawing area, along the bottom of the program window, is a status bar. This bar may contain instructions based on menu selections. It may also display the results of an algorithm.

The drop-down menus are organized into five main menus. The following is a description of the functionality in each menu. Where applicable, sample dialog boxes and other screen shots are included.

- File: The first drop-down menu allows the user to create, save, open, and print a graph.

- New – Create an empty graph or a random graph in the drawing area by filling out the following dialog box.

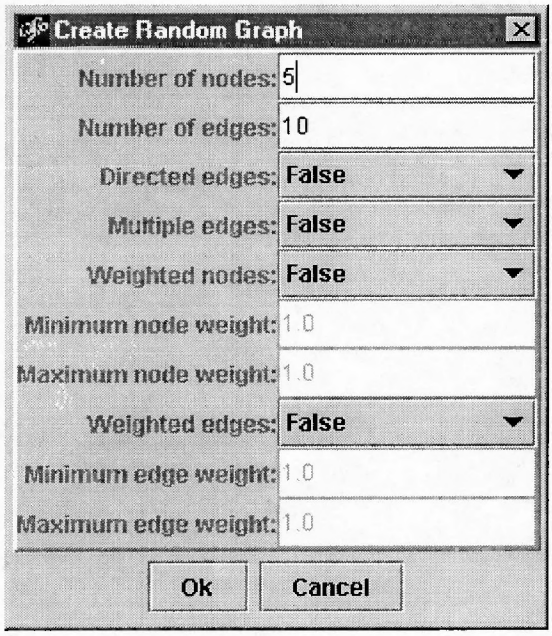


Figure 5.2 – Random Graph Dialog

- Open, Save, Save As – Save a graph to disk and later re-open it. Graph files for this application use the *.jga* extension.

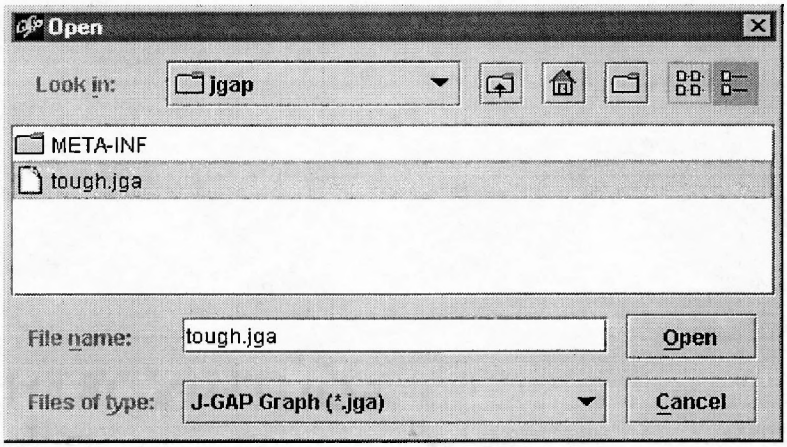


Figure 5.3 – Open File Dialog

- Page Setup, Print – Create a printout of the current graph.
- Exit – Exit the program. The application can also be exited from the title bar. Clicking on the "X" button in the top right will exit the application in Windows.

- Property: The second drop-down menu allows the user to specify properties of the graph that is going to be created. These properties must be selected before creating the graph.
  - Directed edges – All edges in the graph will be directed. An arrow in the middle of the edge indicates the direction of each edge.
  - Nodes have weights – All nodes in the graph will be weighted. The node weight is shown in small text near the bottom of each node. The user will be prompted for the node weight as each node is added.
  - Edges have weights – All edges in the graph will be weighted. The edge weight is shown near the middle of each edge. The user will be prompted for the edge weight as each edge is added.

- Edit: The functions in the third drop-down menu allow the user to create and modify a graph.
  - Add a node – After selecting this item, single clicking in the drawing area will add nodes to the graph.

- Add an edge – After selecting this item, edges can be added by dragging the mouse from one node to another.
- Move a node – When this item is selected all nodes can be moved by dragging them. This is allows the user to rearrange the nodes in the graph to reduce the number of overlapping nodes and edges.
- Remove a node – After selecting this item a node can be removed by clicking on it with the mouse pointer.
- Remove an edge – Select this option and drag from one node to another to remove an edge.
- Create complement of graph – When this option is selected the current graph will be replaced with its complement.

- Algorithms: The fourth drop-down menu contains various algorithms that can be run on a graph. Some algorithms require selection of one or two nodes. Other algorithms are executed once they are selected. If the algorithm selected requires the selection of one or two nodes, instructions will be displayed in the status bar.
    - Test for Class of Perfect Graph – Test whether the current graph is perfect, triangulated, co-triangulated, comparability, co-comparability, permutation, interval, and split.

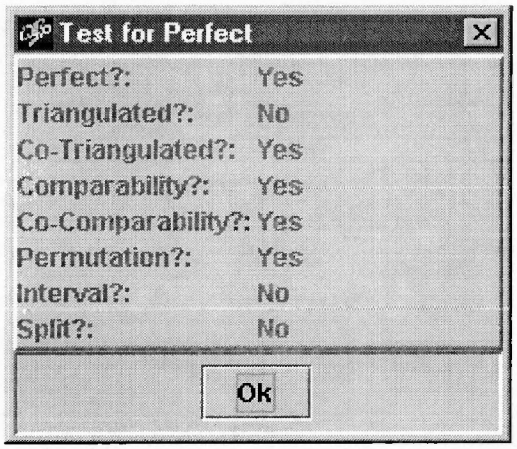


Figure 5.4 – Perfect Graph Dialog

- Find minimum coloring – Assign a minimum coloring to the nodes using an optimum perfect graph algorithm, a heuristic, or an exhaustive search.

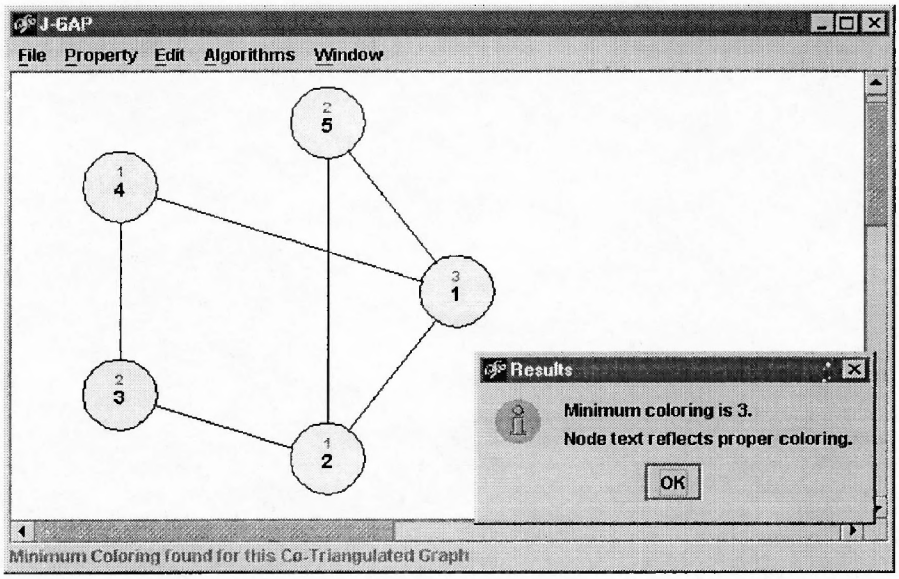


Figure 5.5 – Minimum Coloring Display

- Find maximum clique – Highlight the nodes in the maximum clique using an optimum perfect graph algorithm, a heuristic, or an exhaustive search.

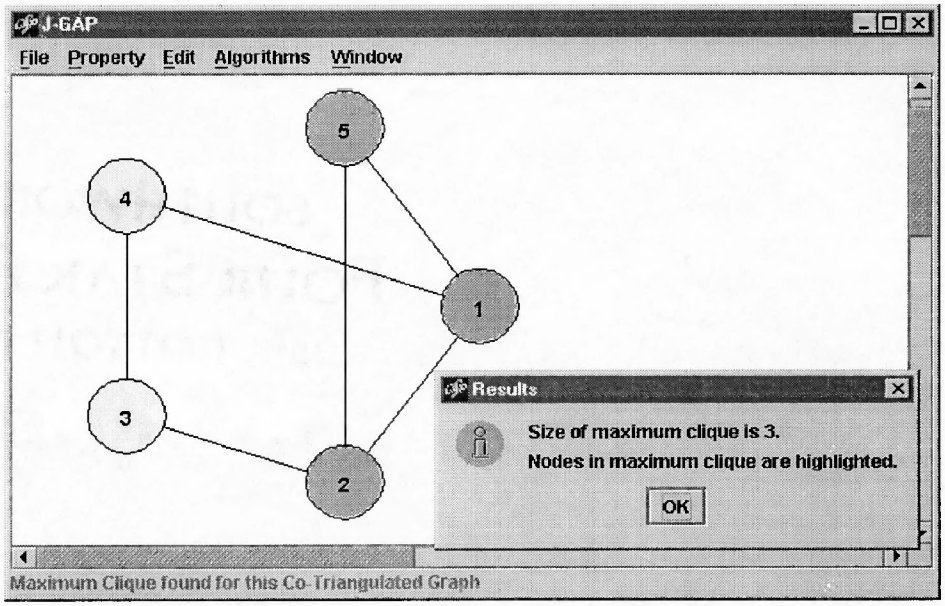


Figure 5.6 – Maximum Clique Example

- Find maximum independent set – Use an optimum perfect graph algorithm, a heuristic, or an exhaustive search to find the maximum independent set.

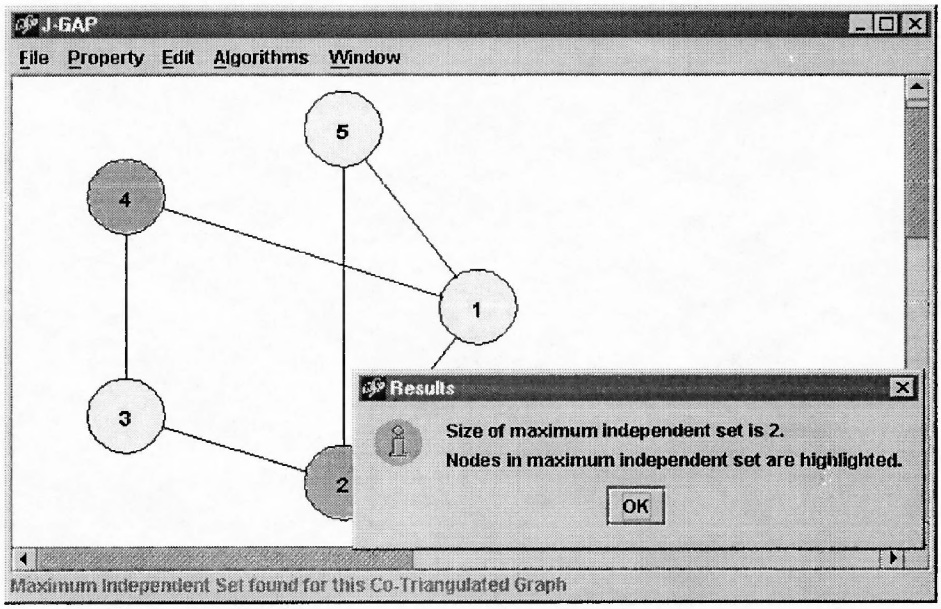


Figure 5.7 – Maximum Independent Set Display

- Find minimum clique cover – Assign a clique number to the nodes using an optimum perfect graph algorithm, a heuristic, or an exhaustive search.

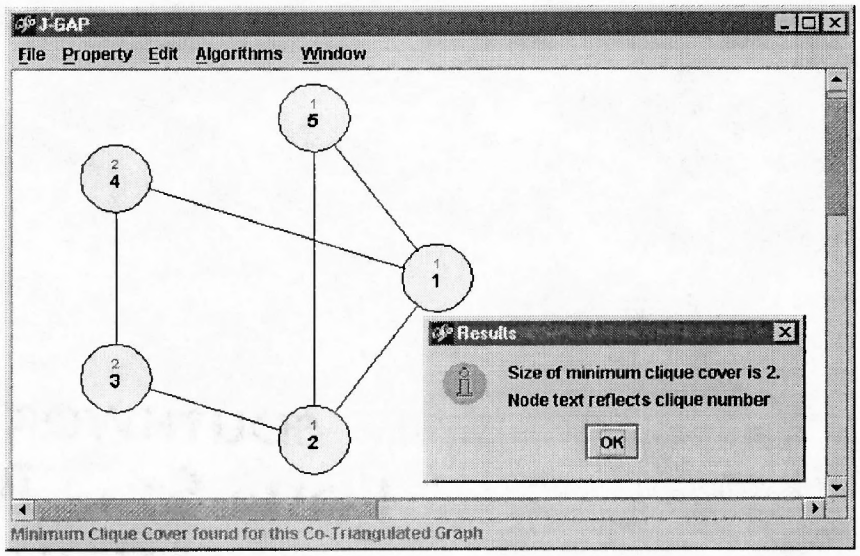


Figure 5.8 – Minimum Clique Cover Display

- Find minimum vertex cover – Highlight the nodes in the vertex cover using either a heuristic or an exhaustive search.

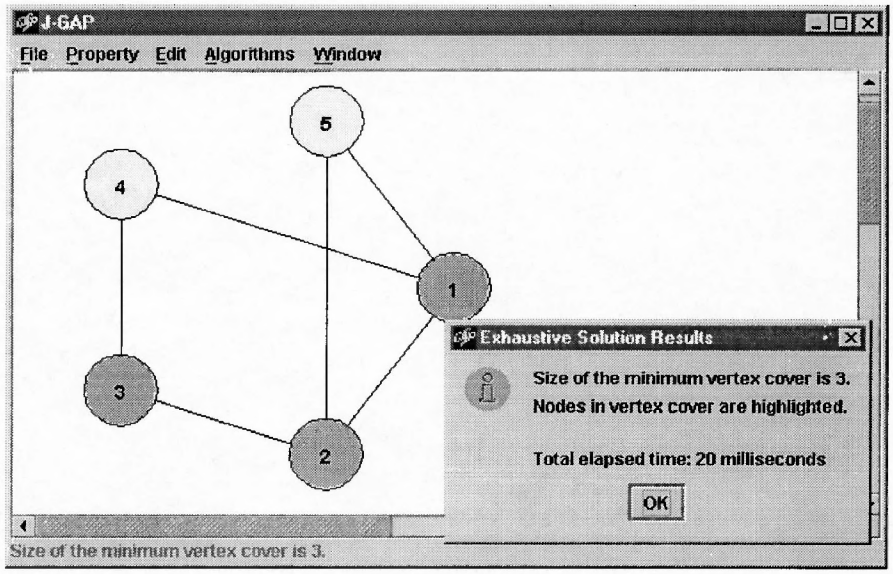


Figure 5.9 – Minimum Vertex Cover Display

- Minimum spanning tree – Highlight the edges in the minimum spanning tree using Kruskal's or Prim's algorithm.

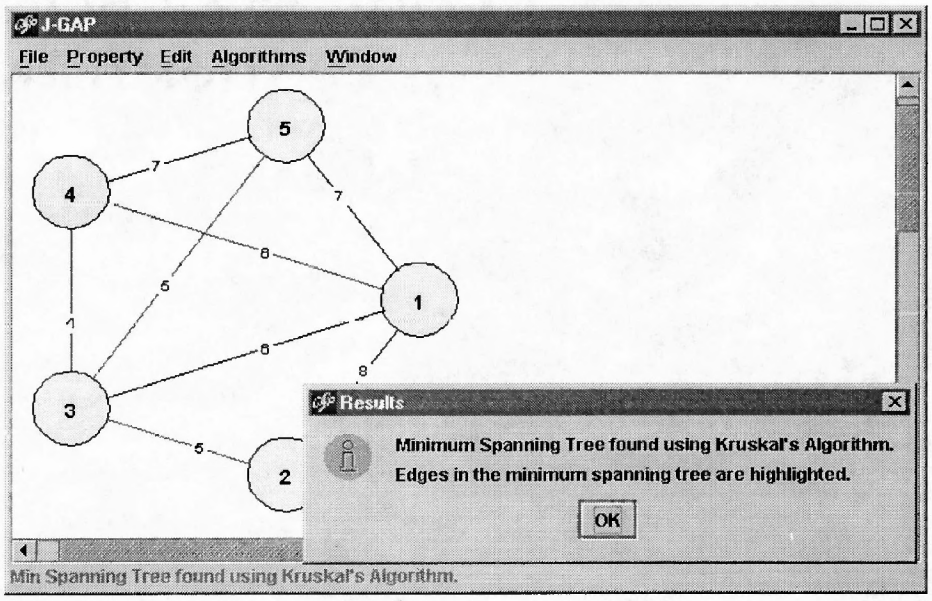


Figure 5.10 – Minimum Spanning Tree Display

- Shortest path – Display the distance from a selected node to all other nodes in the graph using either Dijkstra's algorithm or the Bellman-Ford algorithm. A source node must be selected after selecting one of these algorithms by clicking on the source node.

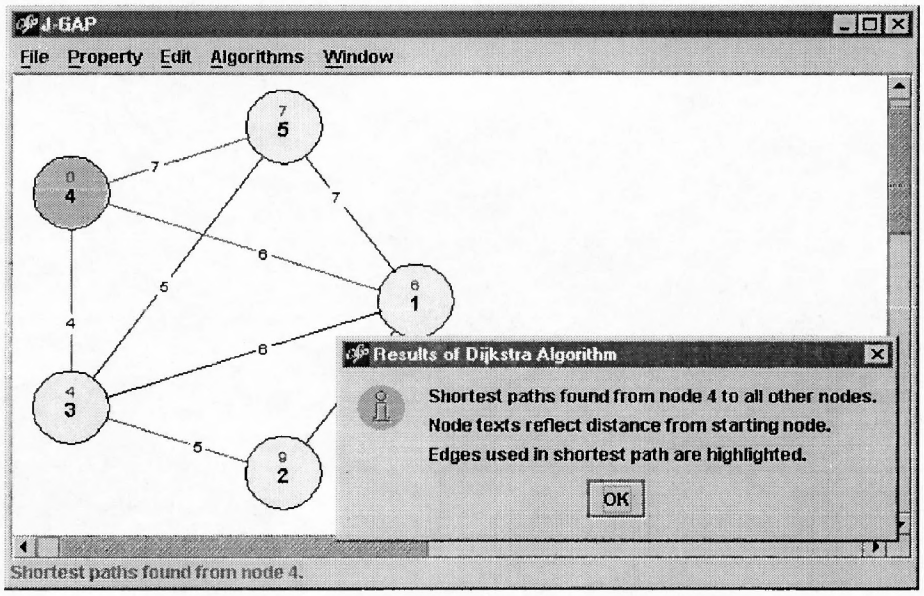


Figure 5.11 – Shortest Paths Display

- Maximum network flow – Determine the maximum network flow along each edge between two selected nodes using the Ford-Fulkerson algorithm. After selecting this algorithm, the source and sink nodes must be selected by clicking on the source node and dragging the mouse to the sink node.

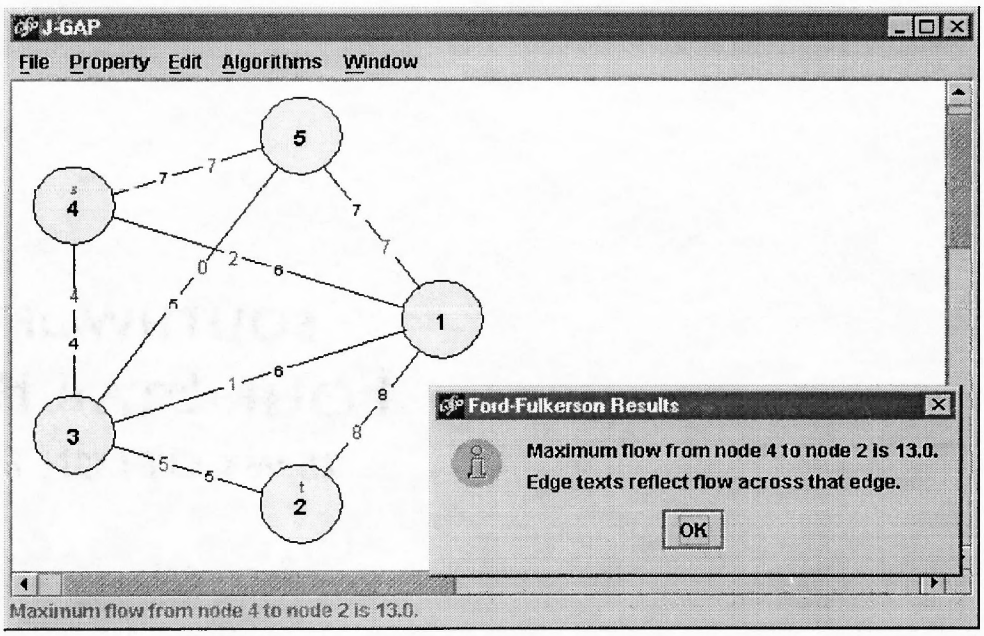


Figure 5.12 – Maximum Network Flow Display

- Euler tour – Determine if the graph contains an Euler tour and if so, label the edges in a graph with its sequence number in the Euler tour.

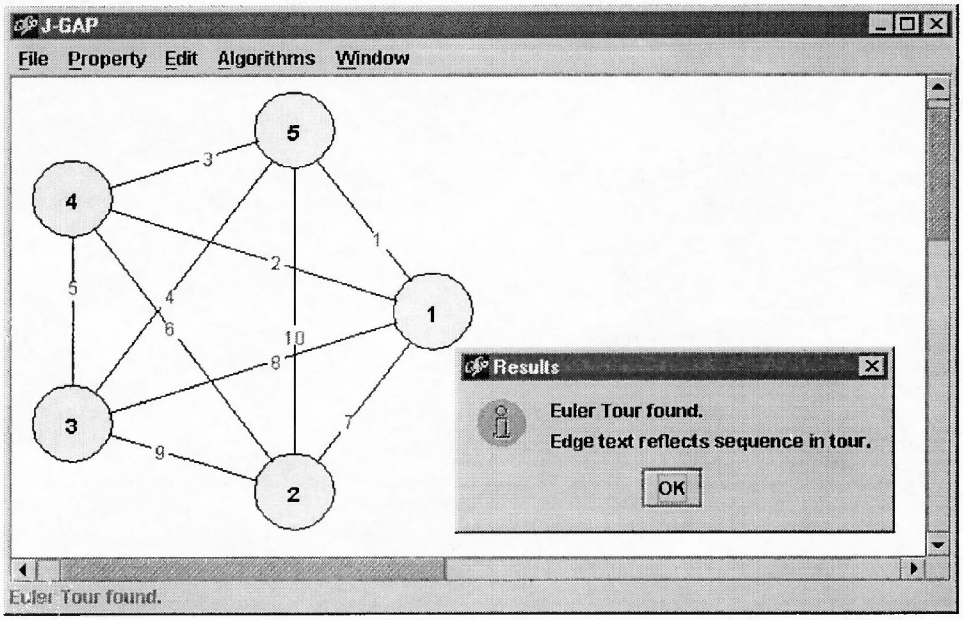


Figure 5.13 – Euler Tour Display

- Maximum matching – Highlight the edges in the maximum matching.

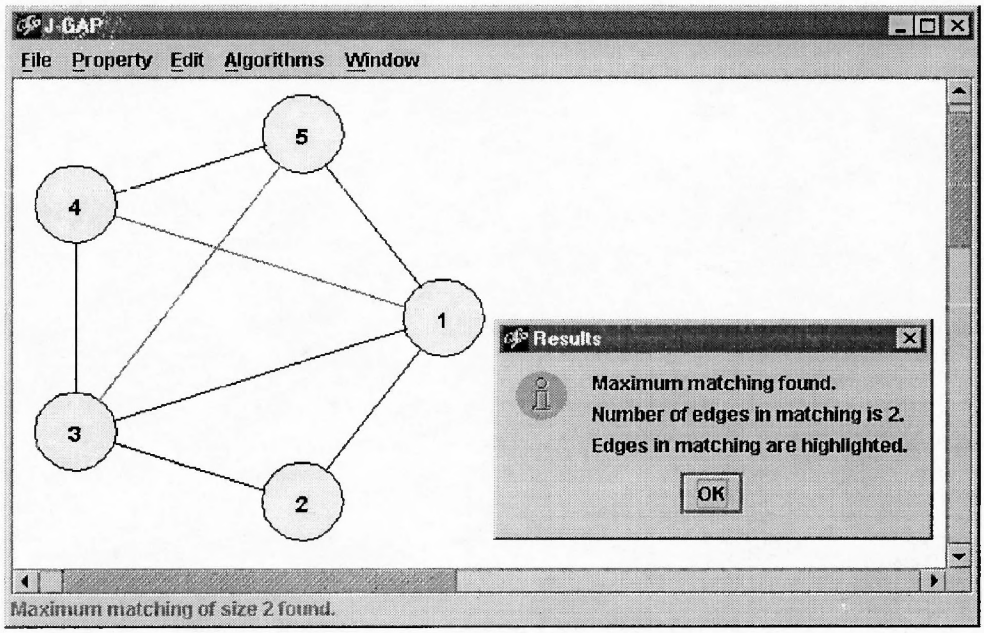


Figure 5.14 – Maximum Matching Display

- Window: The fifth drop-down menu contains functions that modify the display.
  - Refresh display – Redraws the drawing area.
  - Clear algorithm results – Clears highlighted nodes and edges as well as extra text added to the nodes and edges by one of the graph algorithms.
  - Look&Feel – Allows selection of one of three different GUI styles.

Chapter 6 Tolerance Graphs

The chapter includes new research that was accomplished in the area of tolerance graphs. The problem of determining whether a given graph is a bounded tolerance graph is an open problem. A related open problem is finding a bounded tolerance representation of a bounded tolerance graph. Research was performed to find a bounded tolerance representation of a graph. If a bounded tolerance representation could be found for the graph then the graph is known to be a bounded tolerance graph.

6.1 Description of Problem

As previously stated, a tolerance graph is a generalization of an interval graph in which each interval is assigned a tolerance value. Two intervals conflict if the size of their intersection is equal to or greater than one or both of the intervals' tolerances. In a tolerance graph, an edge is drawn between two nodes if their corresponding intervals conflict. Figure 6.1 illustrates a tolerance graph G_1 and its tolerance representation. G_1 is also a bounded tolerance graph since the tolerance of each interval is not larger than the size of the interval.

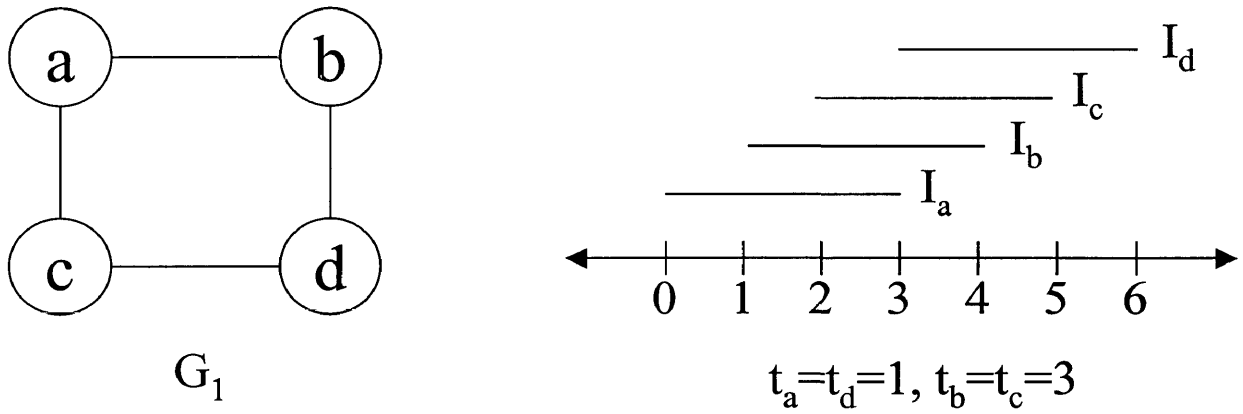


Figure 6.1 – Tolerance Graph and Tolerance Representation

In this example, the size of the intersection of the intervals I_b and I_c is 2. The two intervals do not conflict since this value is smaller than the tolerance of both intervals. However, the intervals I_a and I_b do conflict since their intersection is of size 2, which is larger than the tolerance of I_a.

Converting a bounded tolerance representation to a bounded tolerance graph is a straightforward process of determining which intervals conflict. The reverse process of finding a bounded tolerance representation for a bounded tolerance graph has been an open problem since Golumbic and Monma introduced tolerance graphs in 1982 [GM82]. A related open problem is how to determine whether a given graph is a bounded tolerance graph. These two problems are the focus of the research covered in this chapter.

6.2 Significance of Problem

Finding a bounded tolerance representation for a given graph has both theoretic and practical significance. The theoretic significance is in solving an open problem in the area of tolerance graphs, which would make this relatively new type of graph more complete.

A solution to this problem would have practical significance in that it would allow for the characterization of bounded tolerance graphs. An algorithm that finds a bounded tolerance representation for all bounded tolerance graphs could be used to characterize whether a given graph belongs to the class of bounded tolerance graphs. Once a given graph is determined to be a bounded tolerance graph, efficient algorithms designed for this type of graph could be used such as the maximum independent set and minimum coloring algorithms developed by Narasimhan and Manber [NM92].

6.3 Properties of Tolerance Graphs

There are two properties of tolerance graphs that proved useful in the approaches presented. The first and most important property is that all bounded tolerance graphs are co-comparability graphs [GM84]. In other words, the complement of a bounded tolerance graph is a comparability graph. However, not all co-comparability graphs are

bounded tolerance graphs. Testing whether the given graph is co-comparability is used to quickly rule out a large majority of the graphs that are not bounded tolerance graphs.

The second property that is used to recognize non-tolerance graphs is the property that tolerance graphs (both bounded and unbounded) cannot contain chordless cycles of length 5 or larger. Also, the complement of the tolerance graph cannot contain a chordless cycle of length 5 or larger either. Therefore, all chordless cycles in tolerance graphs are of length 4 or less.

Both of these properties are utilized in the approach described in this chapter. The algorithm for testing whether a graph is a co-comparability graph was covered in chapter 4. The test for chordless cycles of length 5 or larger is integrated in the algorithm developed in section 6.5.

6.4 Special Cases

A couple different approaches were attempted in finding an algorithm to convert the bounded tolerance graph to a tolerance representation. Both approaches took advantage of the special cases of interval graphs and permutation graphs.

6.4.1 Solution for Interval Graphs

As explained in section 6.3, tolerance graphs are co-comparability graphs. If the tolerance graph is also found to be a triangulated graph then it is by definition an interval graph as well. In fact, the entire class of interval graphs is contained within the class of bounded tolerance graphs.

In an interval graph, the intervals conflict if they overlap by even the smallest amount. Because of this, the tolerance of every interval in this class of graphs is zero. An algorithm was presented by Golumbic which converted a given interval graph to an interval representation [GM80]. This algorithm could be easily extended to finding a tolerance representation for the graph by simply setting the tolerance of every interval to zero. The algorithm for finding the tolerance representation of an interval graph is listed below.

```
INTERVAL-REPRESENTATION(G)
1     σ ← perfect elimination scheme of G
2     M ← FIND-MAXIMAL-CLIQUES(G,σ)
3     P ← PQ-Tree(M)
4     for each vertex v ∈ G
5         do R ← set of all cliques in M that contain v
6             PQTREE-ADD-RESTRICTION(P,R)
7     F ← PQTREE-FRONTIER(P)
```

```
8     t ← 0
9     Prev ← ∅
10    for each clique C ∈ F
11        do t ← t + 1
12           Curr ← the set of all vertices in C
13           Old ← Prev - Curr
14           New ← Curr - Prev
15           for each vertex v ∈ Old
16               do start[v] ← t
17           t ← t + 1
18           for each vertex v ∈ New
19               do end[v] ← t
20           Prev ← Curr
21    t ← t + 1
22    for each vertex v ∈ Prev
23        do end[v] ← t
24    for each vertex v ∈ G
25        do tolerance[v] ← 0
```

Lines 1 and 2 create the set *M* containing all of the maximal cliques in the graph using the algorithms for triangulated graphs as discussed in section 4.3.3.1.2. Lines 3-6 then create a PQ-Tree that represents the valid orderings of the cliques. Line 7 stores the

frontier (left to right ordering) of the PQ-Tree in the set *F*. Line 8 initializes a time counter, *t*, to zero, and line 9 initializes the set *Prev* (previous vertices) to empty.

Lines 10-20 loop through the cliques in *F*. First the counter *t* is incremented. Then the nodes that started in this clique are stored in *New*, and the nodes that are not present in this clique but were present in the previous set are stored in *Old*. For each vertex in *Old*, its end value is set to *t*. Then the value of t is incremented and the start value of each vertex in *New* is set to *t*. After line 23, the start and end values for each vertex (and the intervals they represent) have been set. Finally lines 24 and 25 set the tolerance of each interval to zero.

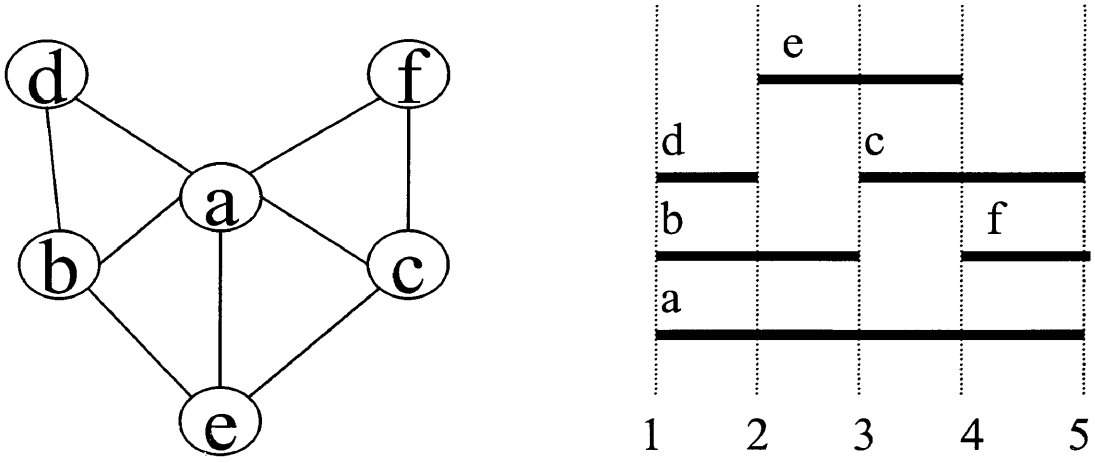


Figure 6.2 – Interval Graph and Interval Representation

In the above interval graphs, the elimination scheme is [d,b,e,c,a,f]. From this elimination scheme the maximal cliques are found to be {d,b,a}, {b,e,a}, {e,c,a}, and {c,a,f}. By creating a PQ-tree that contains each of these cliques and applying

restrictions (for example, the first two cliques must be consecutive since they are the only ones that contain vertex b), a valid ordering of the cliques is found to be [{d,b,a}, {b,e,a}, {e,c,a}, {c,a,f}]. Using the above algorithm, each interval would start with the first clique that contained the corresponding vertex and end with the last clique that contained the vertex. As shown in the above figure, the interval corresponding to vertex *b* would start with the first clique and end just before the third clique.

6.4.2 Solution for Permutation Graphs

Since all bounded tolerance graphs are known to be co-comparability graphs, the subclass of bounded tolerance graphs that are also comparability graphs are by definition permutation graphs as well. Just as with the interval graphs, a permutation graph can be converted into a tolerance graph with a specific tolerance.

Golumbic found that the class of permutation graphs can be represented as tolerance graphs in which the tolerance of every interval is equal to the length of the interval ($t_i = |I_i| \; \forall \; i$) [GM84]. Because of the tolerance of every interval, two intervals will only conflict if one of the intervals completely contains the other.

An algorithm was developed for this thesis, which found a tolerance representation for a known permutation graph. Because permutation graphs are comparability graphs, the solution for this class of graphs takes advantage of the graph's

Hasse diagram. The algorithm processes the vertices in the Hasse diagram a level at a time starting at the bottom level.

One of the problems encountered in this algorithm is illustrated in figure 6.3. This is a simple Hasse diagram that contains only two levels. Since the nodes c, d, and e are on the same level they must have non-conflicting intervals. This is accomplished by assigning non-overlapping intervals for those three nodes. The second level of the Hasse diagram presents a problem. The tolerance interval I_a must completely contain the intervals I_c and I_e but not I_d. This requires that the interval I_d must be placed outside of the intervals I_c and I_e. This would mean that the order of the intervals on a level of the Hasse diagram is dependent on the nodes in all of the higher levels of the diagram.

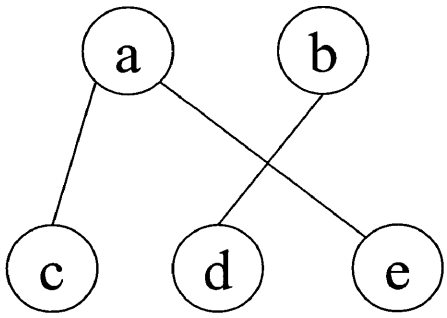


Figure 6.3 – Problematic Hasse Diagram

A simpler solution was found by combining the transitive orientation of the permutation graph with the transitive orientation of the graph's complement. The transitive orientation of the permutation graph is denoted as F_1. The transitive orientation of the complement of the permutation graph is denoted as F_2. By combining all of the

edges from both F_1 and F_2, another orientation of the graph is formed which will be denoted as L. The orientation L forms a complete order. The Hasse diagram of L has a width of one with each node falling on a separate level.

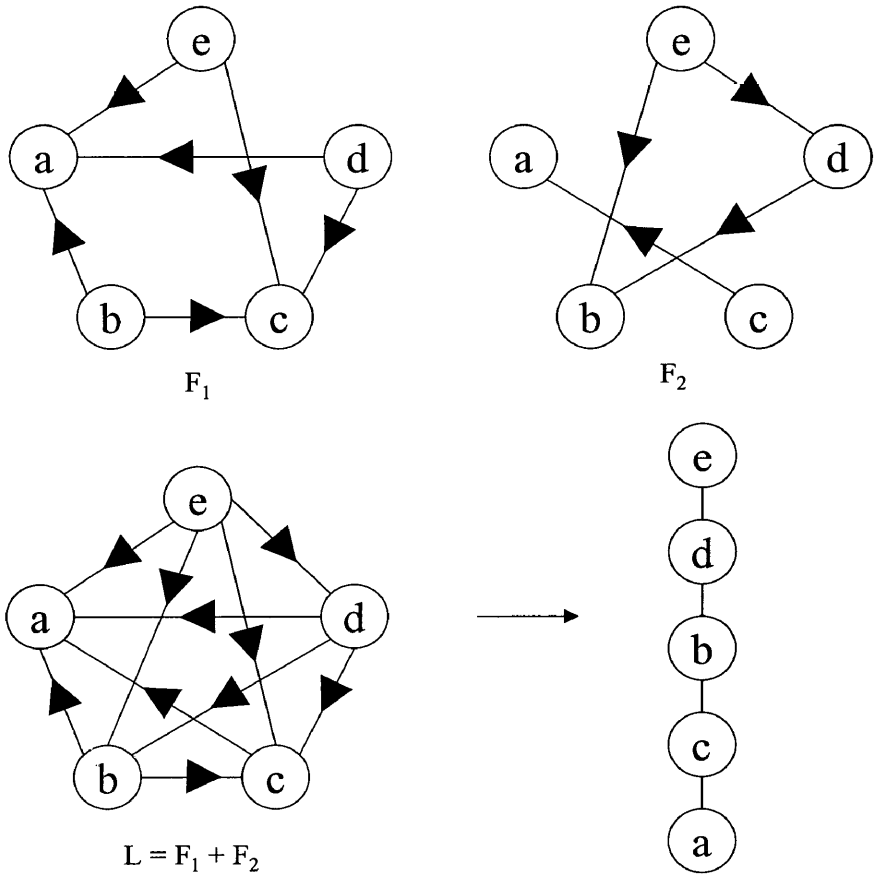


Figure 6.4 – Orientations of a Graph

Figure 6.4 illustrates the transitive orientation, F_1, of a permutation graph. The complement of the graph is transitively oriented in F_2. When the edges of both graphs are combined in L, a complete ordering is found.

The complete ordering of the graph is used to order the nodes on each level of the Hasse diagram. By ordering the Hasse diagram in this fashion, the problem encountered in figure 6.3 is prevented, and a tolerance representation can be found more easily.

After the ordered Hasse diagram is built, the intervals are found for the nodes in each level starting at the bottom level. For the bottom level, the first node is assigned the interval [1,2], the second node is assigned the interval [3,4], and so on.

The higher levels in the Hasse diagram require two steps. First, each node is assigned an interval such that the start of the interval is equal to the minimum of the starts of its children in the Hasse diagram. Similarly, the end of the interval is set equal to the maximum of its children in the Hasse diagram.

Once each node in a level has been assigned an interval, the second step is to make every interval on that level independent. In a Hasse diagram, all of the nodes in the same level form an independent set. If two of those nodes have the same children in the Hasse diagram then they will be assigned the same interval. In order to correct this, the start of one interval is decreased, and the end of the other interval is increased.

After all of the levels of the Hasse diagram have been processed, one more step must be accomplished to form the tolerance representation. For permutation graphs, the

tolerance of every interval is set to the size of the interval. This is the final step in creating a bounded tolerance representation for a permutation graph.

6.5 Approaches to the Solution

A couple different approaches were attempted in the search for an algorithmic method of finding a tolerance representation for a given bounded tolerance graph. The first approach separated the graph into components and utilized a template matching technique similar to the technique used in the PQ-Tree algorithm discussed in chapter 4. The second approach adds edges to the tolerance graph to make it an interval graph. The resultant interval graph is then converted to an interval representation. Finally the tolerance values of the intervals are adjusted to remove the edges that were added.

6.5.1 Bridges and Templates Approach

The first approach started with a piece of the graph and created a tolerance representation for that piece. Then the rest of the edges in the graph are added while adjusting the tolerance representation to incorporate those edges. This approach locates bridges (edges in the graph that if removed would break the graph into two pieces) and uses them to break the problem of finding a representation for the entire graph into finding representations for smaller pieces of the graph as show in the following figure.

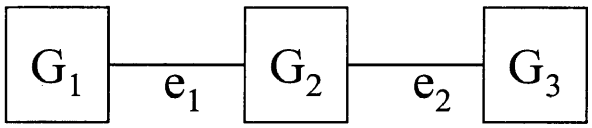


Figure 6.5 – Graph Separated By Bridges

In the above figure, the edges e_1 and e_2 are bridges that connect the subgraphs G_1, G_2, and G_3. If tolerance representations could be found for each piece of the graph in which the nodes incident to the bridges are on the outer edge of each tolerance representation, each piece could be reconnected to form the tolerance representation of the entire graph.

After breaking the graph down, each piece can be tested to see if it is either an interval graph or a permutation graph. Those pieces that are can be converted to tolerance representations using the algorithms in section 6.4. The remaining pieces could be solved by matching major components of the graph (such as chordless cycles of size 4) to templates and using some technique to combine those individual templates. This process of template matching appeared promising at first, but the number of templates required and the complexity of connecting templates to each other eventually lead to the eventual abandonment of this technique.

6.5.2 Temporary Edges Approach

The second approach focused on adding temporary edges so that the algorithm for interval graphs could be used. This approach is made up of three main steps. First,

temporary edges are added to the bounded tolerance graph to make it a triangulated graph and therefore an interval graph. In order to make the original graph triangulated, a chord must be added to each chordless C_4 cycle. Second, the algorithm defined in section 6.4.1 is used to find a tolerance representation of the interval graph. The final step is to increase the tolerance of some of the intervals in order to remove the temporary edges since an edge represents a conflict between two intervals.

There are several items to be considered before continuing this approach. They are as follows:

1. When edges are added to the bounded tolerance graph to make it triangulated, will the graph still be a co-comparability graph?
2. An edge is added to each chordless C_4 to make it triangulated. If the cycle is $[u_1,u_2,u_3,u_4]$, then either the edge (u_1,u_3) or the edge (u_2,u_4) needs to be added. Which edge should be added?
3. Will increasing the tolerance of an interval to remove a temporary edge cause it to not conflict with one of the other intervals and thus remove too many edges?

First, when edges are added to the bounded tolerance graph to make it triangulated, it must continue to be a co-comparability graph since the goal is to convert the graph to an interval graph. Adding an edge to a co-comparability graph is equivalent to removing an edge from a comparability graph. Removing an arbitrary edge from a

comparability graph could cause the graph to no longer remain comparable. However, in this approach edges are only added to the chordless C_4 cycles in the original graph and only enough edges to make the graph triangulated. Extensive testing was done in which edges were added to chordless C_4 cycles of bounded tolerance graphs, and none of the tests resulting in the graph losing its co-comparability property. While not proven mathematically, it appears after testing thousands of graphs that adding edges to a bounded tolerance graph to make it triangulated does not cause the graph to become a non-cocomparability graph.

The second issue is which of the two possible edges should be added to each chordless C_4. In the case of overlapping chordless C_4 cycles, adding one edge may take care of two or more cycles thus reducing the number of temporary edges to remove in the final step. However, this edge may be more difficult to remove since it affects more than one cycle.

The last issue is how to remove the temporary edges without affecting other relationships in the graph. When a temporary edge (u,v) must be removed, the tolerance values of u and v must be set such that $t_u > | I_u \cap I_v |$ and $t_v > | I_u \cap I_v |$. Setting the tolerances of the intervals u and v to be larger than the intersection of the two intervals will remove the conflict and thereby remove the edge from the graph. However, this may have the undesired effect of removing other edges from the graph. Because of this, the resultant tolerance representation must be tested to ensure that it accurately represents

the tolerance graph. This is the point where attempting to find a tolerance representation for a non-tolerance graph will fail. Since the original graph was not a tolerance graph it would be impossible to remove the temporary edges without also removing other edges. Unfortunately, some known tolerance graphs also fail at this point. This occurs when multiple chordless C_4 cycles overlap.

This approach was successful in finding the tolerance representation of a majority of the bounded tolerance graphs. Because of this, it can be used to recognize whether or not most graphs are bounded tolerance graphs. The code for this approach is presented in the following section.

6.6 Implementation of Temporary Edges Approach

The approach described in section 6.5.2 was implemented in Java and incorporated into the J-GAP application developed for this thesis. Pseudo-code is listed and described in this section.

```
TOLERANCE-REPRESENTATION(G)
1     if not IS-CO-COMPARABILITY(G) return NIL
```

The first step in this method is to ensure that the graph is a co-comparability graph. This algorithm is listed in chapter 4. If it is not a co-comparability graph then it is

definitely not a bounded tolerance graph. This method returns a value of nil indicating that a bounded tolerance representation could not be found.

```
2     if IS-TRIANGULATED(G)
3         then   T ← INTERVAL-REPRESENTATION(G)
4                return T
5     if IS-COMPARABILITY(G)
6         then   T ← PERMUTATION-REPRESENTATION(G)
7                return T
```

The next seven lines handle the special cases of interval graphs and permutation graphs. Tolerance representation can be found for both of these classes by using the more efficient algorithms listed in sections 6.4.1 and 6.4.2.

```
8     C ← ∅
9     for each vertex v ∈ V
10        do for each a,b ∈ adv(v) where a ∉ abj(b)
11            do S = adj(a) ∩ adb(b) - v - adj(v)
12               for each s ∈ S
13                   C ← C ∪ {[v,a,s,b]}
```

Next, lines 8 13 create the set C containing all of the chordless C_4 cycles in the graph *G*. Line 8 initializes the set C to the empty set. Line 9 loops through each vertex

in the graph. The loop starting at line 10 finds two vertices that are adjacent to vertex v, but are not adjacent to each other. At this point the path $[a,v,b]$ has been located where a and b are not adjacent. Then line 11 finds all vertices s that are adjacent to a and b but not adjacent to v. For each vertex s found, the cycle $[v,a,s,b]$ is added to the set of chordless C_4 cycles if it is not already in the set.

```
14    Temp ← ∅
15    for each cycle c ∈ C
16        do Temp ← Temp ∪ ADD-CHORD(G,c)
```

Lines 14-16 add the temporary edges to the graph. First the set of temporary edges, *Temp*, is initialized to the empty set. Then line 15 loops through each cycle in the set *C*. Line 16 adds the temporary edge to the graph and also adds that edge to the set of temporary edges.

```
17    if not IS-TRIANGULATED(G)
18        then return NIL
```

After a chord has been added to each chordless C_4 the graph should be triangulated. This is tested in line 17. If the graph is not triangulated then a chordless cycle of size 5 or larger must have existed in the original graph. Since chordless cycles

of size 5 or larger are not allowed in bounded tolerance graphs, this method returns the value of nil.

```
19    T ← INTERVAL-REPRESENTATION(G)
```

At line 19 the graph has been turned into an interval graph. This line then uses the *Interval-Representation* method to convert this interval graph to a tolerance representation, which is stored in *T*.

```
20    for each edge e ∈ Temp
21        do REMOVE-TEMP-EDGE(G,T,e)
22    if not IS-VALID-REPRESENTATION(G,T)
23        then return NIL
24    return T
```

The last five lines of this method attempt to remove the temporary edges. Line 20 loops through each temporary edge and removes it in line 21 using the method described below. After the edges are removed, the resultant tolerance representation is verified in line 22. If the representation is not a valid tolerance representation for the given graph, then the value of *nil* is returned otherwise the valid tolerance representation is returned.

The method *Remove-Temp-Edge* works as follows. There are four possible relationships between the two intervals associated with the temporary edge. Figure 6.6 illustrates those relationships.

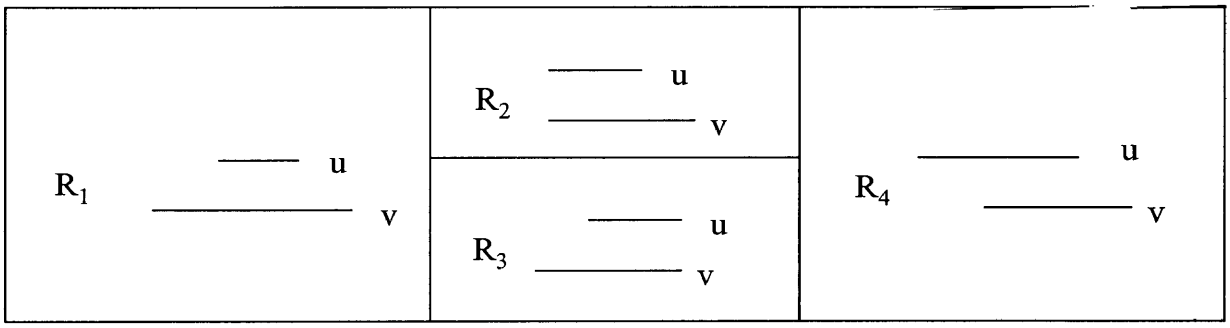


Figure 6.6 – Interval Relationships

For each relationship the temporary edge can be removed by changing the tolerances of the intervals and possibly adjusting the start or end of the interval to allow the graph to remain bounded. The following four bullets detail how each relationship is handled. In these bullets, I_u refers to the interval u. The points s_u and e_u refer to the starting and ending points of the interval u along a real number line. The tolerance of the interval u is represented as t_u.

- R1 – If I_u is contained in I_v ($s_u > s_v$ and $e_u < e_v$) then $t_u = \infty$, $t_v=\max\{t_v,|I_u|+0.1\}$; graph is unbounded.
- R2 – If I_u and I_v have a common starting point ($s_u = s_v$, $e_u \leq e_v$) then $s_u=s_u-0.5$, $t_u=\max\{t_u, | I_u \cap I_v|+0.1\}$, $t_v=\max\{t_v, | I_u \cap I_v|+0.1\}$.

- R3 – If I_u and I_v have a common ending point ($e_u = e_v$, $s_u \geq s_v$) then $e_u=e_u+0.5$, $t_u=\max\{t_u, | I_u \cap I_v|+0.1\}$, $t_v=\max\{t_v, | I_u \cap I_v|+0.1\}$.
- R4 – If I_u and I_v have staggered starting and ending points ($s_u<s_v$ and $e_u<e_v$) then $t_u=\max\{t_u, | I_u \cap I_v|+0.1\}$, $t_v=\max\{t_v, | I_u \cap I_v|+0.1\}$.

The first relationship, R_1, shows one interval contained within the other interval. The tolerance of interval u would have to be made unbounded in order to remove this edge. This would make the graph unbounded which contradicts the goal of finding a bounded tolerance representation. The next two relationships R_2 and R_3 illustrate two intervals that have either common starting points or common ending points. In R_2, the starting point of interval u could be shifted left to allow for a bounded tolerance of the two intervals. The same could be done for the ending point of interval u in R_3. The final relationship, R_4, illustrates two overlapping intervals in which the tolerance of each interval could be set to a fraction larger than their intersection.

6.7 Analysis

The temporary edges approach was tested by generating a series of random graphs and counting the number of graphs that were recognized as being bounded tolerance, not a bounded tolerance graph, or unknown.

The first test used 1000 random co-comparability graphs of each of the following sizes: 5-15 nodes, 10-20 nodes, 15-25 nodes, and 25-35 nodes. Figure 6.7 illustrates the findings of this test. As the graphs become larger and more complex, this approach is unable to characterize more graphs as being either bounded tolerance or not bounded tolerance. The approach is able to characterize from 87% to 93.1% of the co-comparability graphs.

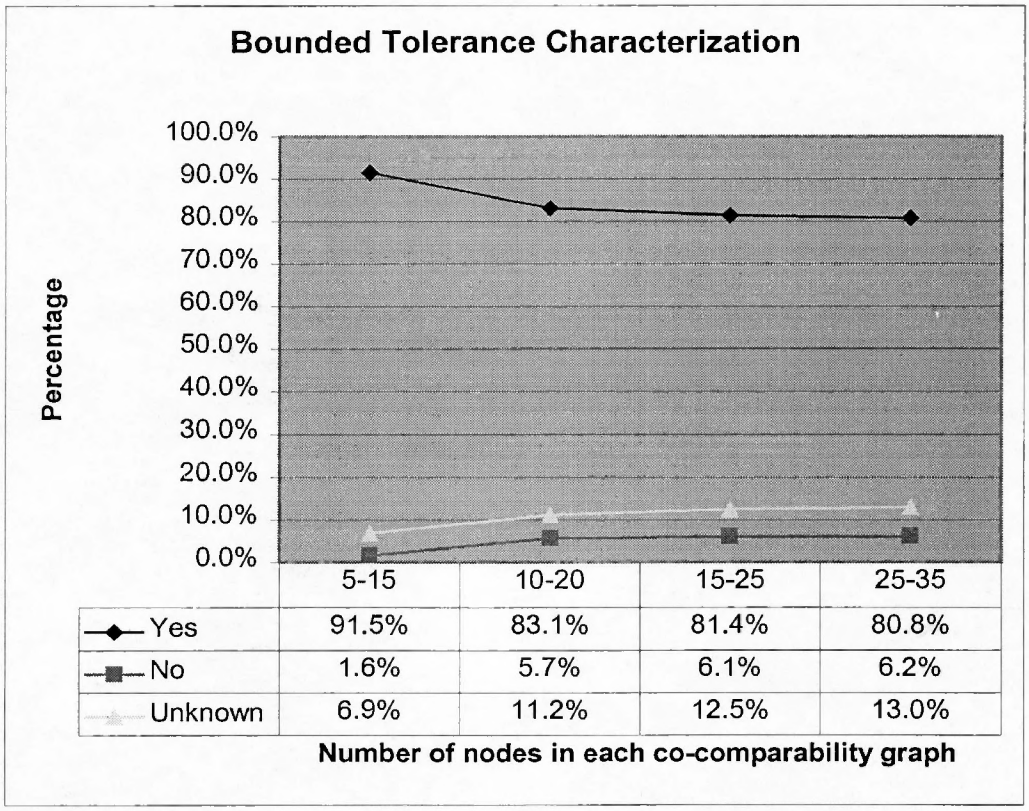
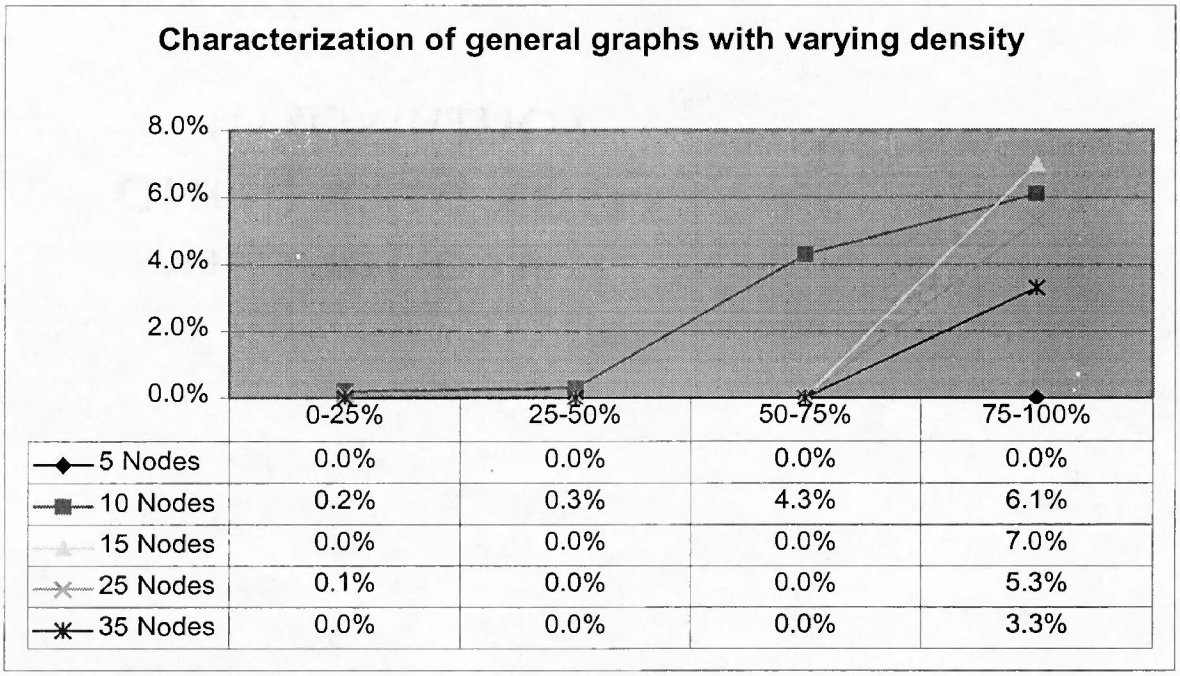


Figure 6.7 – Analysis with varying graph sizes

The second series of tests varied the density of the graphs. The density is the number of edges in the graph. The density of the graph can range from 0%, if no edges

are present, to 100%, if the graph is a complete graph. In this analysis, 1000 random graphs were generated in each of the density ranges of 0-25%, 25-50%, 50-75%, and 75-100% for each of five different graph sizes. The percentages of graphs that could not be characterized in this analysis are illustrated in figure 6.8. In these tests, the higher density graphs were more difficult to characterize. This analysis showed that the percentage of graphs that could be characterized ranged from 93% to 100%.



	0-25%	25-50%	50-75%	75-100%
5 Nodes	0.0%	0.0%	0.0%	0.0%
10 Nodes	0.2%	0.3%	4.3%	6.1%
15 Nodes	0.0%	0.0%	0.0%	7.0%
25 Nodes	0.1%	0.0%	0.0%	5.3%
35 Nodes	0.0%	0.0%	0.0%	3.3%

Figure 6.8 – Analysis with varying density

6.8 Results

The problem of converting a bounded tolerance graph to a bounded tolerance representation has been an open problem since tolerance graphs were first introduced. A

solution to this problem could also be used to characterize whether a given graph is a bounded tolerance graph.

Two different approaches were taken to solve this problem. The first approach successfully divided the graph into smaller portions that could be attacked separately. The use of template matching to solve the individual pieces was found to not be feasible.

The second approach taken was to add temporary edges to convert the bounded tolerance graph to an interval graph. The interval graph can easily be converted into a tolerance representation. The difficult part of this approach was in removing the temporary edges. While this approach did not find tolerance representations for all bounded tolerance graphs, it was successful for most graphs.

Both approaches took advantage of the special cases of interval graphs and permutation graphs. For interval graphs, a known algorithm was used to convert the graph to a bounded tolerance representation. For permutation graphs, a new algorithm was developed to find a bounded tolerance representation that took advantage of the Hasse diagram and the transitive orientations of both the graph and its complement.

Chapter 7 Summary and Conclusions

Graph theory can be used to provide an algorithmic approach to solving a wide variety of real world problems. Some of the applications of graph theory include transportation where the quickest route between two cities must be found, and for scheduling when an optimum assignment of a company's workers to a list of tasks is needed. These problems are solved by first modeling the problem as a graph. A graph is a diagram that represents the objects of the problem and the relationships between those objects. Once the problem has been modeled as a graph, one of the graph algorithms can be applied to find the solution to the problem.

Many graph problems require an exponential amount of time to solve. In these cases one of three approaches must be taken. One option is to perform an exhaustive search for the solution. This approach is only practical for small graphs since larger graphs can require months or even years to solve. The second approach is to use a heuristic to find a near optimum solution. Heuristics are designed to find a good solution in polynomial time rather than searching for the best solution in exponential time. The third approach is to attempt to classify the graph into one of the classes for which there exists a polynomial algorithm to find the optimum solution.

Perfect graphs are an important class of graphs in that several significant problems that take exponential time to solve in general graphs can be solved in

polynomial time for all perfect graphs. Such problems include the minimum coloring problem, the maximum clique problem, the minimum clique cover problem, and the maximum independent set problem. The class of perfect graphs is a large class that includes all triangulated graphs, comparability graphs, interval graphs, split graphs, and permutation graphs as well as their complements.

The interval graphs represent problems in which several objects occupy portions of a real number line. The objects conflict if their intervals overlap. Interval graphs can represent periods of time that classes are held at a university. If the meeting times of two classes overlap then they conflict since they cannot be assigned to the same classroom.

Depending on the application, a small amount of overlap may be acceptable. For example an individual may try to attend two meetings if they overlap by only 5 minutes. An interval graph would show a conflict between the interval since they overlap by a few minutes. A newer class of graphs known as tolerance graphs was created to allow for this flexibility. Tolerance graphs allow for a tolerance value to be assigned to each interval. If the overlap of two intervals is not larger than the tolerance of both intervals then the two intervals do not conflict. This generalization of interval graphs allows it to more closely model the variations that often occur in the real world.

7.1 Summary and Conclusions

There were three main goals of this thesis. The first was to make graph theory available to a larger audience by making them easier to use. Although graph theory is very powerful, it is not being fully utilized because of the level of knowledge required. The solution to this problem was accomplished through the creation of an extensive graphical application. The application allows a user to create a graph by either selecting the menu item to create a random graph or by adding nodes and edges through a graphical user interface. The user can then execute an algorithm by selecting it in the drop-down menu.

The second goal was to implement important graph algorithms that solved both basic and advanced graph algorithms. These algorithms were implemented in the application developed for this thesis.

The third goal was to do research in one of the open problems in the area of tolerance graphs. Since tolerance graphs are a relatively new type of graph, several important problems remain open. One of those problems is the characterization of bounded tolerance graphs. In other words, an algorithmic approach is needed to classify whether a given graph belongs to the class of bounded tolerance graphs. Another related open problem is how to convert a bounded tolerance graph into a corresponding bounded tolerance representation. The process of converting the tolerance representation to its

graph form is a straightforward process. The reverse process has been an open problem since it was identified by Golumbic and Monma in 1982.

A solution to the second problem of finding a bounded tolerance representation for bounded tolerance graphs could also be used to solve the first problem of characterizing bounded tolerance graphs. If a bounded tolerance representation could be found for a given graph, then the graph is a bounded tolerance graph.

A solution was provided that found a bounded tolerance representation for a majority of the bounded tolerance graphs. This solution characterizes a given graph as being a bounded tolerance graph, not a bounded tolerance graph, or unknown. If a bounded tolerance representation can be found for the graph, then the graph is a bounded tolerance graph. If the graph is not a co-comparability graph or if the graph contains an illegal subgraph, then the graph is not a bounded tolerance graph. Some more complicated graphs that contain overlapping chordless cycles of size four cannot be successfully characterized using this solution.

7.2 Future Research

Future research could be applied toward the implementation of other graph algorithms into the application developed for this thesis. The vast number of graph algorithms that have been developed have made it impractical to implement all of the

algorithms in this thesis. Some of the algorithms that could be implemented include the planar graph algorithms, which could be used to better arrange the random generated graphs. Currently, the nodes of a random generated graph are arranged in a circular fashion. This lay out results in a large number of edges that cross over each other. Planar graph algorithms could reduce or eliminated the number of edges that cross in the drawing area. Other more simple algorithms that could be added include an all-pairs shortest paths algorithm, and algorithms to find the cut-vertices and cut-edges in the graph.

The J-GAP application has been designed to allow additional algorithms to be easily incorporated. The abstract data types for the graphs, nodes, and edges include many useful methods to modify the graph including creating the complement of the graph and adding and removing certain nodes and edges. Extra attributes can be easily added to each node and edge. Adding another algorithm requires the creation of a new class for the algorithm, and minor modifications in two of the other classes in the application.

Future research could also be performed in the area of tolerance graphs. Research could be applied toward modifying the process of finding a bounded tolerance representation such that a solution can be found for all bounded tolerance graphs. The approach taken in this thesis was to add temporary edges and take advantage of the known algorithm for interval graphs. The difficulty in this approach is in removing those temporary edges in the graph and the corresponding tolerance representation. Other

methods of removing those temporary edges could be explored. Future research could also be made toward finding a completely different approach to solving this problem.

There are other open problems in the area of tolerance graphs that could be researched. Open problems identified by Golumbic and Monma include the characterization of tolerance graphs (both bounded and unbounded). Another open problem is whether all tolerance graphs that are co-comparability graphs are bounded tolerance graphs [GM84]. All bounded tolerance graphs are co-comparability graphs. It is unknown whether there are any unbounded tolerance graphs that are co-comparable.

Bibliography

[AH93] Andrea, T., Henning, U., and Parra, A. (1993), On a Problem Concerning Tolerance, *Discrete Applied Mathematics* 46, pages 73-78.

[BD93] Bibelnieks, E. and Dearing, P. (1993), Neighborhood Subtree Tolerance Graphs, *Discrete Applied Mathematics* 43, pages 13-26.

[BF95] Bogart, K., Fishburn, P., Isaak, G., and Langley, L. (1995), Proper and Unit Tolerance Graphs, *Discrete Applied Mathematics* 60, pages 99-117.

[BL76] Booth, K. and Lueker, G. (1976), Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms, *Journal of Computer and System Sciences* 13, pages 335-379.

[BT94] Bogart, K. and Trenk, A. (1994), Bipartite Tolerance Orders, *Discrete Mathematics* 132, pages11-22

[CL90] Corman, T., Leiserson, C., and Rivest, R. (1990), *Introduction to Algorithms*, The MIT Press

[GA85] Gibbons, A. (1985), *Algorithmic Graph Theory*, Cambridge University Press

[GM80] Golumbic, M. (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Inc.

[GM82] Golumbic, M. and Monma, C. (1982), A Generalization of Interval Graphs with Tolerances, *Congressus Numeratum* 35, pages 321-331.

[GM84] Golumbic, M., Monma, C., and Trotter, W. (1984), Tolerance Graphs, *Discrete Applied Mathematics* 9, pages 157-170.

[NM92] Narasimhan, G. and Manber, R. (1992), Stability Number and Chromatic Number of Tolerance Graphs, *Discrete Applied Mathematics* 36, pages 47-56.

[WW90] Wilson, R. and Watkins, J. (1990), *Graphs: An Introductory Approach*, John Wiley & Sons, Inc.