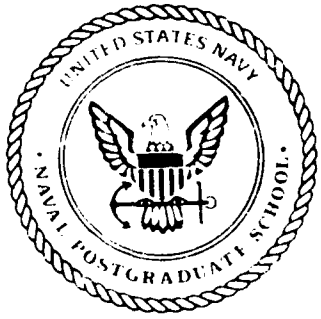
NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

IMPLEMENTATION OF AN EFFICIENT ALGORITHM TO DETECT MAXIMAL CLIQUES IN A CONFLICT GRAPH

by

Kristi Jo Bell

June 1990

Thesis Advisor: Kim A. S. Hefner

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
IMPLEMENTATION OF AN EFFICIENT ALGORITHM TO DETECT MAXIMAL CLIQUES IN A CONFLICT GRAPH

12. PERSONAL AUTHOR(S)
Bell, Kristi J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	June 1990	130

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Maximal Clique, Conflict Graph, Algorithm, Heuristic, NP-Complete

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In military operations, radio-frequency communications play an important role in command and control. Since the breadth of control may be limited by frequency and channel constraints, research continues to search for better ways to optimize the frequency allocation. In this thesis, graphs are used to model radio-communications networks. The problem considered is the detection of maximal cliques, representing subnets, from the graph model. However, detection of cliques is an NP-complete problem. Since NP-complete problems are not likely to be solvable in a reasonable time if the input is large, this paper limits the network input to six stations and fifteen transmissions. An algorithm is implemented in Pascal to detect *all* maximal cliques of a network and is known as the program CLIQUE. The program is designed to accept arbitrary connected graphs without being affected by isomorphisms and without generating duplicates. This thesis describes a limited solution to the clique problem and solves a subproblem of the communications frequency problem in real-time.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Kim A. S. Hefner	22b. TELEPHONE (Include Area Code): (408) 646-2198; 22c. OFFICE SYMBOL: MA

Approved for public release; distribution is unlimited.

IMPLEMENTATION OF AN EFFICIENT ALGORITHM TO DETECT MAXIMAL CLIQUES IN A CONFLICT GRAPH

by

Kristi Jo Bell
Captain, United States Army
B.S., United States Military Academy, 1982

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: _____
Kristi Jo Bell

Approved By: _____
Kim A. S. Hefner, Thesis Advisor

_____
Man-Tak Shing, Second Reader

_____
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

In military operations, radio-frequency communications play an important role in command and control. Since the breadth of control may be limited by frequency and channel constraints, research continues to search for better ways to optimize the frequency allocation. In this thesis, graphs are used to model radio-communications networks. The problem considered is the detection of maximal cliques, representing subnets, from the graph model. However, detection of cliques is an NP-complete problem. Since NP-complete problems are not likely to be solvable in a reasonable time if the input is large, this paper limits the network input to six stations and fifteen transmissions. An algorithm is implemented in Pascal to detect *all* maximal cliques of a network and is known as the program CLIQUE. The program is designed to accept arbitrary connected graphs without being affected by isomorphisms and without generating duplicates. This thesis describes a limited solution to the clique problem and solves a subproblem of the communications frequency problem in real-time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Radio-frequency communication plays a very important role in military operations. One of the major tasks in radio-communications network design is to establish the required point-to-point communications within channel capacity constraints. To tackle such a task, a radio-communications network can be modeled as a graph G. The network's stations (also known as nodes or sites) are modeled as the vertices (points) of a graph and the network's communications requirements and constraints are modeled as the edges (lines) between stations. This thesis does not model non-transmission requirements which also affect communications planning (e.g., weather, equipment availability, maintenance). We limit the criteria used to model the network, as directed by the network manager, to transmission requirements between stations.

In particular, this thesis will model a subproblem of assigning frequencies in a radio-communications network. Hintze previously looked at this subproblem in-depth and designed an algorithm to model it [Ref. 1:p. 59]. The input to Hintze's algorithm is a directed graph D, where the arcs (directed edges) represent transmissions between stations in a net. Two stations are in conflict with each other if they are both transmitting to a third station. To capture these conflicts, a conflict graph $C(D)$ is created in which there is an undirected edge joining two nodes if there is a conflict between them [Ref. 1:p. 45]. Given the conflict graph, Hintze's algorithm finds the largest directed graph D' such that $C(D') = C(D)$. Note that the digraph D' maximizes the network's transmission capability without increasing the number of frequencies used. By determining the net's maximum capacity, we confirm the

network's "limit" on the number of possible transmissions between stations. If we exceed the limit, we cause additional conflicts, requiring additional frequencies beyond our allocation [Ref. 1:p. 49].

With this in mind, an integral step in Hintze's algorithm requires finding the sets of stations in $C(D)$ known as *maximal cliques*. A station is a member of a *clique* if it is adjacent to every other station in the *clique*. We represent the clique as a graph K_n on n vertices and call the graph *complete* since every distinct pair of vertices is joined by a transmission edge [Ref. 2:p. 57]. The *clique* is *maximal* if it is not contained in a larger clique. If we then find the *minimum* number of maximal cliques which *cover* the edges (every edge must be a member of at least one clique) of $C(D)$, we have a *minimal edge clique covering* for the communications network. Hintze's algorithm uses the minimal edge clique covering of $C(D)$ to determine which vertices in the original network, D, can increase transmissions without creating more conflict or needing more frequencies [Ref. 1:p. 63].

We believe that Hintze's algorithm is too constraining for the frequency assignment problem. Therefore, the program CLIQUE in this thesis detects *every* maximal clique in a graph, known as the *edge clique covering* ($ECC(G)$). It is necessary to find *all* maximal cliques so that, if a station is eliminated from the network, the network manager may still know where conflicts exist in the resultant network. See Chapter II, Section D for an example.

Detecting maximal cliques in a conflict graph *appears* to be an easy problem. For a small (less than seven) set of vertices, we can usually "see" the maximal cliques. Unfortunately, a computer does not have our "vision" and most communications nets of consequence consist of more than seven stations. Additionally, most networks are built dynamically. The network configuration at any one time or place is dependent

upon a multitude of factors. Since the "links" between stations can grow or decline dynamically, a computer program which calculates the changes in network configuration *efficiently* would be very useful. However, using an algorithm to detect *cliques* is known to be NP-complete and is in general difficult to solve within reasonable time constraints [Ref. 3:p. 347].

Current research indicates two favored techniques/approaches are used to find the maximal cliques of an edge clique covering. The first, an algorithm, is "a step-by-step procedure for solving a problem" and is done "...in a finite number of steps that frequently involves a repetition of an operation." [Ref. 3:p. 1] However, using an algorithm to solve an NP-complete problem that has a large input size, such as the clique problem, is hard. So, we require a different strategy. Artificial intelligence "search" methods are representative of a second strategy. *Heuristics* are developed and, unlike an algorithm, do not necessarily give a "unique recommendation" in every situation. Instead, the heuristic acts as a "gardener," "pruning branches" until a sufficient, possibly optimal, solution is found [Ref. 4:p. 199]. Since the similar problems of finding a clique, vertex cover, dominating set and independent set are known to be NP-complete, most research in detecting cliques has centered on the development of heuristics.

This thesis solves a small part of the clique problem with an *algorithm* which detects cliques by partitioning the problem into number of stations, number of transmissions per net, and number of transmissions per station. In attempting to keep the problem manageable, the number of stations is limited to six and the number of transmissions per station is limited to five. Initially, prior to understanding the difficulty associated with problems which are NP-complete, we tried solving the clique problem with a single comprehensive algorithm. Realizing that a

comprehensive algorithm is not arrived at easily, this thesis implements the program CLIQUE using a reduced algorithmic approach.

The chapters which follow delineate problems and solutions associated with the detection of maximal cliques. Chapter II describes in greater detail the definitions, terms and research upon which this thesis is based. Chapter III describes the program CLIQUE in detail. The emphasis is on the interaction of the program's components and how this program contributes to the research for detecting cliques. Chapter IV analyzes the solution presented in this thesis and summarizes the performance of CLIQUE. Chapter V discusses the program's weaknesses and makes recommendations for further areas of research. At the very minimum, the reader will get an idea of some techniques that were tried to solve the clique problem and the success/failure rate associated with each.

II. BACKGROUND

The purpose of this chapter is to present background material associated with cliques and the communications subproblem, to state the specific objectives of CLIQUE, and to summarize research addressing the detection of cliques.

A. DEFINITIONS

In graph theory, a "picture" can be often more easily understood than the written definition. Therefore, the definitions in this section will reference Figure 1. A graph, G, is a collection of points and lines. The points are known as the vertices, $v \in V(G)$, which is the entire collection of vertices in G. The lines are known as the edges, $e \in E(G)$ where $E(G) \subseteq V(G) \times V(G)$. The graphs in this thesis will not have self-adjacent vertices (loops) or multiple edges because we assume that a station in a net would not be in conflict with itself and that a single edge is sufficient to illustrate a conflict between two stations. As mentioned in the introduction, a *complete subgraph* K_n is a maximal clique C_i. The maximal cliques found by the program CLIQUE are illustrated in Figure 2. In Figure 1, two complete subgraphs are found in the conflict graph $C(D)$, the **net**. The first is a K_2 subgraph on two vertices, forming the *maximal clique*, $C_1 = \{A, B\}$. The second is a K_3 subgraph on three vertices, forming the *maximal clique*, $C_2 = \{B, C, D\}$. If K has k vertices, it will be known as a *clique* of order k or a *k-clique*. In this example, C_2 is a three-clique or a clique of order three. Notice that edge $\{B, C\}$ is also a clique, however, it is not maximal since it is contained in the larger clique C_2. Since C_2 is not contained in other clique, it is maximal. For the remainder of this paper, *maximal clique* and *clique* will be used interchangeably. [Ref. 2:p. 57]

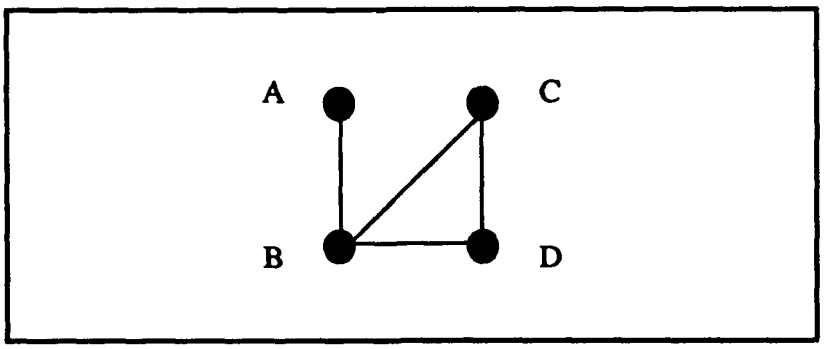


Figure 1 Conflict Graph *C(D)*: The Net

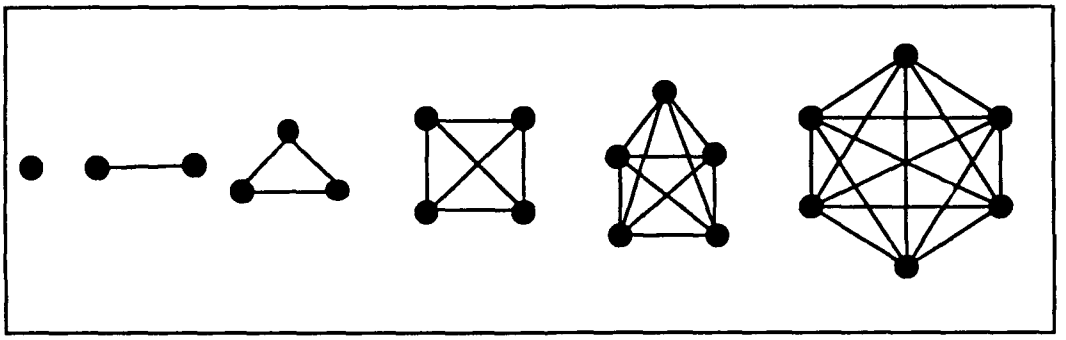


Figure 2 Maximal Cliques, K_n

If we combine cliques, C_1 and C_2 then we have a *family*, *C*, of cliques. The number of cliques in the family is represented as |C| and is called the *cardinality* of *C*. *C* is known as an *edge clique covering* if every edge of the net is in at least one member of *C*. The edge clique covering is said to be minimal if, for all clique coverings *C'* of the net, |C'| >= |C|. The minimal edge clique covering problem is to find a minimum edge clique covering *C*. [Ref. 2:p. 57]

As mentioned in the introduction to this thesis, finding a minimal edge clique covering is not enough to describe a communications network's conflicts. Instead, we

want to find the set of all maximal cliques in G. We call this problem the edge clique covering problem. This covering allows the network manager to supervise the net despite the elimination of one or more stations from the net. Remember that a minimal edge clique covering (which eliminates maximal cliques whose vertices are also members of other maximal cliques) is not sufficient for this thesis. It eliminates known "redundancy" which might otherwise be useful to the network manager.

With the basic terminology defined, it is beneficial to describe pioneering work in graph theory which influenced the development of the conflict graph. The conflict graph is a general case of the *competition graph*, first introduced by Cohen to model food webs in the ecosystem. A competition graph is defined mathematically as an undirected graph, $G(D, B, C)$ in which D is a digraph and B and C are, not necessarily disjoint, sets of vertices in D. G contains an edge between any two distinct vertices x and y of B if and only if, for some vertex z in C, there are arcs (x, z) and (y, z) in D. See Figure 3 for an example of a generalized competition graph. In this paper, the communications problem requires that the sets B and C be equal because it will be assumed that any receiving station (set C) is also capable of being a transmitting station (set B). [Ref. 5:p. 295]

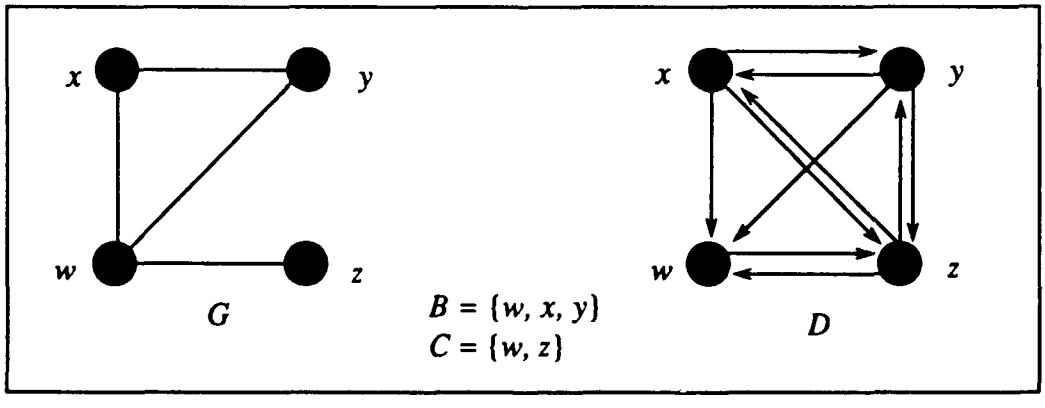


Figure 3 A Competition Graph and Its Digraph

Applications of competition graphs include modeling communications over a noisy channel (confusion graph), modeling complex systems (row graph), modeling food webs (niche overlap graph), and modeling radio or television transmitters (conflict graph) [Ref. 5:p. 296]. Since, as mentioned in the introduction, we are modeling transmitting stations in a communications net, the conflict graph application best meets our requirements. As mentioned before, two transmitting stations are in conflict if and only if a transmission by either could be received by the same third station [Ref. 5:p. 296]. Hintze gives an example of a conflict graph and its digraph, shown in Figure 4 [Ref. 1:p. 54]. In this thesis, the original digraph and the source of conflict is immaterial. Instead, we will concentrate on the resulting undirected graph and the discovery of a given *arbitrary* graph's edge clique covering. We emphasize arbitrary graphs because we realize that nets are configured dynamically. Any two nets having the same number of conflicts may have different adjacency matrices. This paper uses an adjacency matrix as input to represent the nets. The discrete input of ones and zeros is ideally suited for computer manipulation. An adjacency matrix represents the connectivity of the net with a one, "1", for a conflict and a zero, "0", if there is no conflict between two stations. In Figure 5, although the second matrix is configured with the vertices rotated once counter-clockwise, both matrices represent the conflict graph of Figure 4. The mathematical term for this phenomenon is *isomorphism.*

Although isomorphism contributes to the difficulty of finding cliques within reasonable time constraints, lack of an efficient algorithm is the primary constraint to finding an optimal solution to the clique problem. To understand the emphasis of current and past research, it is now necessary to define NP-complete. According to Manber, an *efficient* algorithm has a running time of $O(P(n))$, where $P(n)$ is a

polynomial in the size of the input n represented in bits. Polynomial-time algorithms are known as *tractable* and usually have practical solutions. However, there are quite a few problems not yet solvable by polynomial-time algorithms. Existing algorithms for these problems are too slow for even moderately large input instances. It makes sense to recognize these problems early, even before the quest for an algorithm begins, and avoid looking for a non-existent algorithm. These problems are classified as NP-complete problems. [Ref. 3:pp. 341-2]

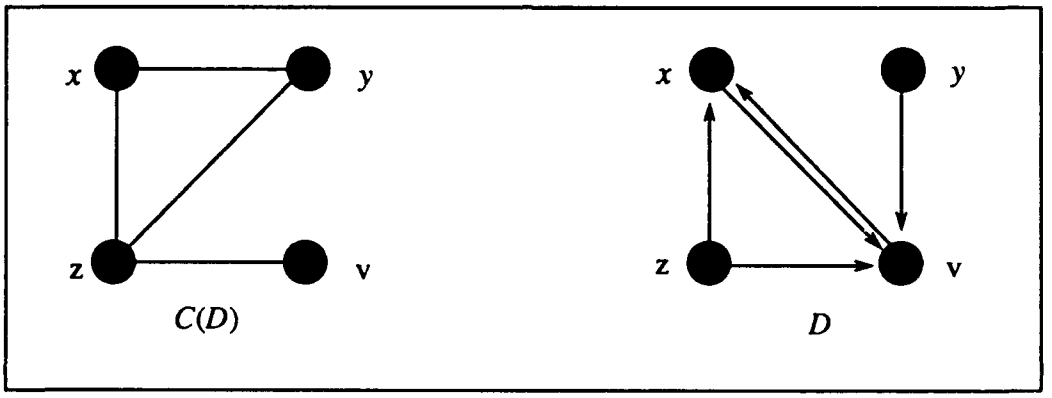


Figure 4 A Conflict Graph and Its Digraph

	v	x	y	z
v	0	0	0	1
x	0	0	1	1
y	0	1	0	1
z	1	1	1	0

	v	x	y	z
v	0	1	1	0
x	1	0	1	1
y	1	1	0	1
z	0	1	0	0

Figure 5 Adjacency Matrices of the Same Conflict Graph

According to Manber, Cook first proved the existence of NP-complete problems. Cook's proof made it easier to prove the existence of other NP-complete problems since, given any new problem, Z, it is *sufficient* to prove that Cook's SAT problem, or any other NP-complete problem, is *polynomially reducible* to Z. Thus, with the definition for NP-completeness and the fact that reducibility is a transitive relation, it may be possible to prove *any* two problems polynomially reducible. The importance of polynomially reducibility is clear since many additional problems were subsequently found to be NP-complete and saved many researchers from wasting time looking for non-existent efficient algorithms. [Ref. 3:p. 346]

The importance of the class of NP-complete problems is emphasized in Manber's definition since "...there exists an efficient algorithm for any one NP-complete problem if and only if there exist efficient algorithms for all NP-complete problems." [Ref. 3:p. 342] So, if we can polynomially reduce clique to a problem with an efficient algorithm, then an efficient algorithm exists for the clique problem also. However, since an efficient algorithm has not yet been found, the program CLIQUE in this thesis takes a small subset of the clique problem and automates it. The program is based on a "brute force" algorithm. The input graph is analyzed first by number of vertices, then by number of edges, and finally by the degrees of the vertices. An explanation follows in Chapter III.

B. PROBLEM DESCRIPTION

In a large communications network, say fifty or more stations, it is difficult to determine optimal frequency assignments. The network is often constrained by distance, equipment, weather and other factors. The need exists, therefore, for a method to keep track of conflicts and adjust the frequency assignments as required. Hintze designed an algorithm which handles a subproblem of the frequency

assignment problem [Ref. 1:pp. 59-60]. In this paper the program CLIQUE detects the maximal cliques in a arbitrary network. This will lead to identifying the stations which can add transmitters to their sites without adding conflict to the net.

The largest clique that the program CLIQUE detects is a K_6. This does not mean the largest network which can be input contains six stations. It does mean, however, that the largest *subnet* of a network which can be input is limited to six stations. Subsequently, multiple subnets may be combined to form the original net and be analyzed again. On the other hand, "stations" could represent whole networks. Thus, conflicts between six *networks* could also be analyzed.

Any communicator desiring to add additional links without increasing network conflict will want to use CLIQUE. Characteristics of its application are found in Chapter III.

C. SOLVING THE CLIQUE PROBLEM BY FINDING *A* MAXIMUM CLIQUE

1. Introduction

Tarjan declares the "obvious" algorithm examines every subset of the vertices of a graph, determines if the subset is a clique, and then chooses the largest clique found. Since there are 2^n subsets possible, where n is the number of vertices, the number of searches grows correspondingly. The time bound of the algorithm would be $O(n2^n)$, which is unsuitable for finding cliques in a reasonable time for large graphs. Intuitively, the problem must be approached differently and Tarjan does this by examining "a sufficiently large number" of vertices and limiting the quest to a single maximum clique. Although the single clique is not sufficient for the communications problem, Tarjan's paper illustrates the difficulty in designing an algorithm to solve an NP-complete problem. [Ref. 6:p. 1]

2. Definitions

G is a graph with vertex set *V*. G_S is the subgraph of *G* with vertex set *S* where $S \subseteq V$. G_S and G_{V-S} are the subgraphs of *G* induced by the subsets S and (V - S). Let *A*(C) be the set of vertices adjacent to one or more vertices in the clique C. In Figure 6, if C_1 = {x, y, z} and C_2 = {v, x, y, z} then C_2 dominates C_1. In other words, C_1, is a clique but not maximal since it is contained in C_2. If every clique in G_S is *dominated* by at least one clique in a set of cliques C, then the set of cliques C is dominant. Dominance is transitive. Finally, let ||*G*|| be the size of a maximal clique in *G*. [Ref. 6:pp. 1-2]

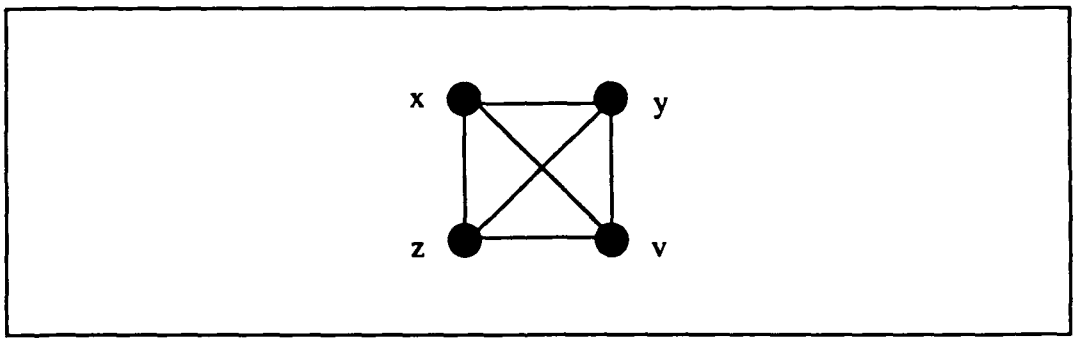


Figure 6 Example Graph

3. The Solution

In 1972, Tarjan described a *recursive* algorithm for finding *one* maximal clique of maximum size in *G*. The graphs's set of vertices is subdivided and examined for cliques. Once a clique C is found, the vertices in *A*(C) are examined. If possible, the two sets are combined to form a larger clique until the *maximum* clique in *G* is found. Crucial to the process is the relation between one subset and its adjacent subset of

vertices. One subset must be *Dominant* so that the algorithm will eventually succeed. The following lemmas are included for reference.

- *Lemma 1*: Let $G = (V, E)$ be a graph. Let $S \subseteq V$.
- Then $\|G\| = \max_{C \text{ a clique in } G_S} \{|C| + \|G_{A(C) - S}\|\}$ [Ref. 6:p. 2]

- *Lemma 2*: Let $S \subseteq V$. Let $\mathbf{C}$ be a dominant set of cliques in G_S.
- Then $\|G\| = \max_{C \in \mathbf{C}} \{|C| + \|G_{A(C) - S}\|\}$ [Ref. 6:p. 3]

Tarjan lists possible subproblems which might be encountered in determining the maximum clique. Ultimately, his approach gives a worst case time analysis of $t(n) \leq k(1.286)^n$, where n is number of vertices and k is some constant. This time analysis is better than 2^n and handles 2 3/4 as many vertices as the "obvious" algorithm can handle [Ref. 6:p. 13]. However, you will notice that the smaller number, 1.286, is possibly still not reasonable for large networks.

4. Observations and Conclusions

To summarize, Tarjan solves a smaller clique problem first in the subset S of vertices for each clique in a dominant set of cliques for G_S. He then applies the procedure recursively until a maximum clique is found [Ref. 6:pp. 2-3]. Tarjan's algorithm is not suitable for determining *all* of the conflicts in a net as is necessary for the communications problem. It will find the largest clique, however, and this may suffice in some situations.

D. SOLVING THE CLIQUE PROBLEM USING A MINIMAL EDGE COVERING

1. Introduction

Kou et al. design a heuristic algorithm to discover the *minimal* edge clique covering of a graph [Ref. 7:pp. 137-8]. An objective of their algorithm is to find an

optimal covering of edges. This optimal covering would not satisfy the requirements of finding cliques for a communications network because it eliminates a clique whose elements are found in other clique sets. In communications, every clique indicates a possible subnet which might influence frequency assignment and its existence needs to be known by the network manager.

2. Definitions

The heuristic algorithm uses an incidence matrix as input. An incidence matrix looks like an adjacency matrix except that the rows represent vertices and the columns represent edges. An entry in an incidence matrix is "1" if and only if the vertex is *incident* to the edge. [Ref. 7:p. 135]

3. The Solution

Kou et al. developed two heuristic algorithms to minimize the number of cliques found in a graph [Ref. 7:p. 135]. Since they believe that NP-completeness does not necessarily preclude finding polynomial-time approximation algorithms [Ref. 7: p. 137]. The heuristic uses an incidence matrix, P, and does not allow disconnected nodes. In Figure 7, their heuristic finds $C_1 = \{1, 3, 5\}$, $C_2 = \{2, 3, 4\}$, $C_3 = \{4, 5, 6\}$, and $C_4 = \{3, 6\}$ for graph A [Ref. 7:p. 138]. For graph B in Figure 7, the first algorithm finds all possible maximal cliques [Ref. 7:p. 138]. These are $C_1 = \{1, 2, 3\}$, $C_2 = \{1, 2, 4\}$, $C_3 = \{1, 3, 5\}$, and $C_4 = \{2, 3, 6\}$. However, the second "improved" algorithm produces an *optimal* (minimal) edge clique cover which eliminates the redundant triangle, $C_1 = \{1, 2, 3\}$. In CLIQUE, the redundant triangle is not eliminated.

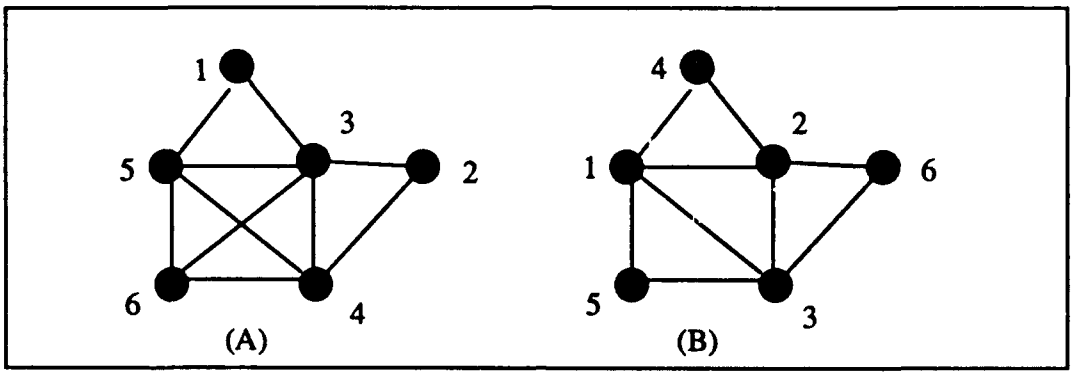


Figure 7 Finding a Minimal Edge Clique Covering

4. Observations and Conclusic `s

The program CLIQUE finds the ECC assignment C_1 = {1, 3, 5}, C_2 = {2, 3, 4}, and C_3 = {3, 4, 5, 6} for Figure 7, graph A and it finds the same ECC as the original heuristic for graph B. As mentioned in this paper's introduction, although a minimal covering is sufficient for Hintze's algorithm, *all* of these cliques may be necessary for communications frequency management [Ref. 1:p. 59]. For example, if we eliminated station six from the net in graph B, we might not realize that stations two and three are still in conflict. The "improvement" made by Kou et al. eliminates this subset of cliques (assuming he finds the cliques with the heuristic) without accounting for future planning requirements. Therefore, although it could be useful in certain situations, this heuristic is not useful for our purposes.

E. SOLVING THE CLIQUE PROBLEM WITH MAXIMAL DEGREE FOUR

1. Introduction

Pullman defines mathematical algorithms for calculating minimal edge clique coverings on graphs with vertex maximal degree less than five. He claims that the algorithms proposed can be accomplished in linear time $O(n)$, where n is the number

of vertices. This is quite an improvement over the exponential time algorithm which pursues all possibilities, such as Tarjan's in Section C. [Ref. 2:p. 57]

2. Definitions

Let G be a graph with vertices of degree less than five. Given G, if H is a subgraph containing some, but not all of the edges and vertices of G, then H is a proper subgraph. If a proper non-empty subgraph separates the cliques of G, then we say that G is *clique-separable*. Otherwise, G is *clique-inseparable*. B is a *clique-block* if B separates the cliques of G and no subclique of B does. B "separates the cliques" when every clique of G has either all or none of its edges in H. In other words, B's edges must not be contained in a triangle. If they are contained in a triangle, the triangle must not share edges with any other triangles of G. The maximum number of degrees of the vertices in G are denoted by $\Delta(G)$. The *neighborhood* of H consists of H and every vertex and edge of G adjacent to the vertices of H. *Deletable* subgraphs are defined as isolated vertices, neighborhoods of $\Delta(G)$-cliques, triangles not sharing edges with other triangles, and $\Delta(G)$-cliques. Examples of clique-inseparable graphs with $\Delta(G) \leq 4$ are shown in Figure 8. In the first algorithm presented by Pullman, the cardinality of the minimal edge clique covering, $cc(G)$ and the cardinality of the clique partition $cp(G)$ are computed. (We will not discuss clique-partitioning in this paper.) Let T(G) be the number of triangles in G. [Ref. 2:pp. 58-63]

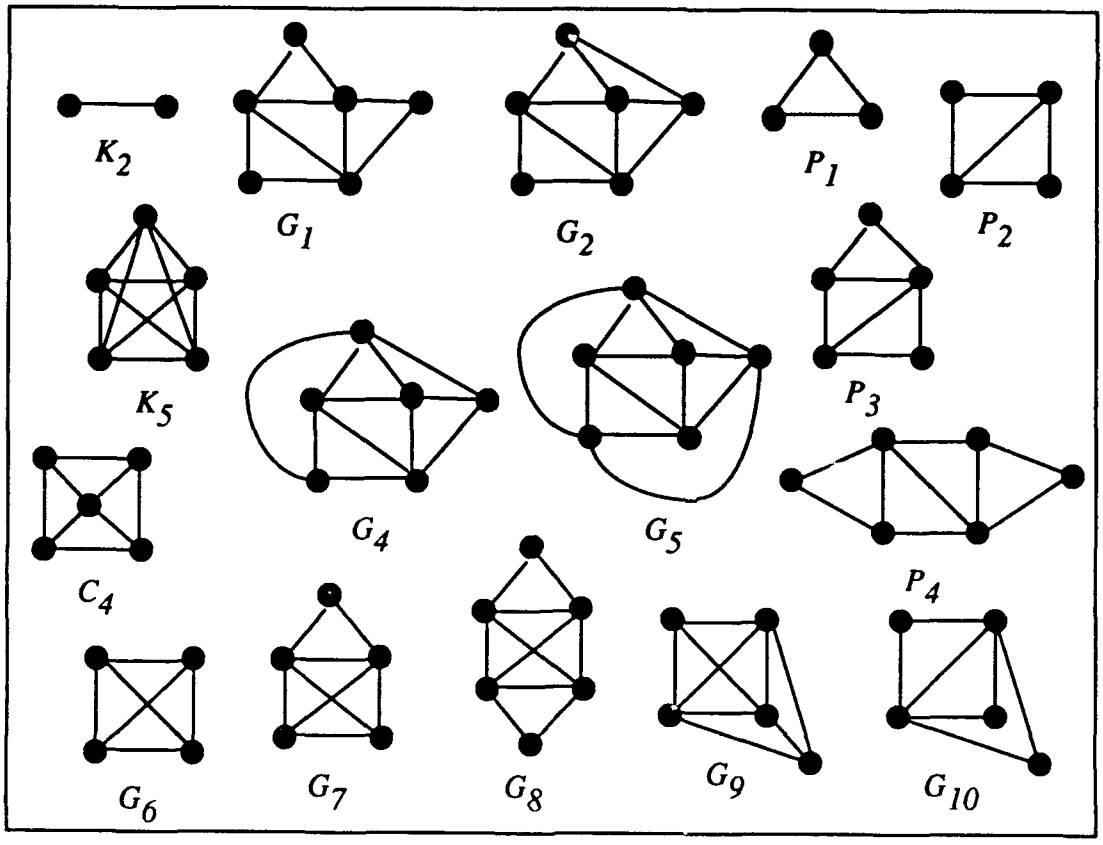


Figure 8 Clique-Inseparable Graphs

3. The Solution

Pullman's algorithm depends on the identification of clique blocks that contain triangles. He identifies 24 possible clique-inseparable graphs in his paper, and stores their characteristics in a table for quick reference [Ref. 2:p. 62]. The characteristics stored include the graph's edges and cc(G). In his algorithm, triangles are located recursively until all of the edges are used and the cliques identified. When no further triangles remain, the algorithm identifies any remaining edges as K_2 cliques. A second algorithm simply takes any edge, identifies the clique block, B, and

adds the cc(B) and cp(B) number to a running total. It then removes the edges from consideration and quits when all edges in G have been used. [Ref. 2:p. 58]

4. Observations and Conclusions

Of the research looked at thus far, Pullman's algorithms are the most closely related to the algorithm used in the program CLIQUE. However, CLIQUE detects *all* maximal cliques. Differences between the two algorithms are identified in Table 1. Because CLIQUE has an input limit of six stations, Table 1 does not contain Pullman's graphs consisting of more than six stations. The program CLIQUE removes clique blocks, usually starting with K_2 cliques, and proceeds to identify triangles which can be removed as clique blocks or those that are contained in K_4 and K_5 cliques.

F. SOLVING THE CLIQUE PROBLEM WITH BRANCH-AND-BOUND

1. Introduction

Bron and Kerbosch describe two backtracking algorithms designed to find all maximal complete subgraphs (cliques). They use a heuristic approach known as branch-and-bound. This technique eliminates "branches" which will not eventually lead to a clique. [Ref. 8:p. 575]

2. Definitions

Three sets are used in their approach. The first set known as *compsub* is the set which is expanded or shrunk by a vertex while traveling along the branch of the backtracking tree. The second set, *candidates*, are vertices which have not yet been used to extend the set in compsub. The last set consists of vertices which served earlier as extensions of the present compsub but are now explicitly excluded from further consideration and is known as *not*. To stop the heuristic from looking for more

cliques, a vertex must exist in the set *not* and be connected to all the *candidates*. This position in the heuristic is called the *bound condition*. [Ref. 8:p. 575]

TABLE 1
CARDINALITY ASSOCIATED WITH CLIQUE BLOCKS

G	P_1	P_2	P_3	G_{10}	P_4	C_4	G_1	G_6	G_2	G_7	G_4	G_8	G_9	G_5
$T(G)$	1	2	3	3	4	4	4	4	5	5	6	6	7	8
cc(G)	1	2	3	3	4	4	3	1	4	2	5	3	2	4
ECC(G)	1	2	3	3	4	4	4	†1	5	†1‡1	6	†1‡2	†2	8
E(G)	3	5	7	7	9	8	9	6	10	8	11	10	9	12

† indicates number of K_4 cliques
‡ indicates number of K_3 cliques

3. The Solution

Two versions of their heuristic are presented. In the basic version, the cliques are detected in lexicographic order. A key condition of this algorithm is a reliance on the set called *not*. Bron and Kerbosch discovered the necessity for the *candidates* set to be empty in order to create a clique, otherwise *compsub* could still be extended. The set *not* must also be empty to meet the condition that the present *clique* not be contained in another clique. (A clique is not maximal if contained within another.) They claim that the branch-and-bound method detects a clique earlier than the "brute force" algorithm. In the second version, their heuristic reduces the number of branches traversed. In doing so, it generates cliques arbitrarily. It is based, not on

the selection of *any* candidate, but on the selection of a *well-chosen* candidate. The emphasis is on reaching the "bound condition" as soon as possible. [Ref. 8:p. 575]

4. Observations and Conclusions

Since this approach uses a branch-and-bound heuristic, it may take exponential-time to find the ECC. For a communications network, this approach may or may not be sufficient depending upon the time constraints in which the cliques must be found.

III. THE CLIQUE PROGRAM

The program CLIQUE is an algorithm designed to detect all the maximal cliques of a conflict graph. Although constrained to communications networks consisting of six stations, we believe that the communications planner will find the program useful for maximizing the number of transmissions while simultaneously avoiding the addition of conflict between stations. This chapter is devoted to the program's characteristics and contributions to communications planning research.

A. INTERACTION OF THE PROGRAM'S COMPONENTS

1. The Components

The program CLIQUE is designed and implemented on an IBM-compatible XT using Intel's 8088 processor for memory management, 640K bytes of random access memory (RAM) for storing the program and related files, Microsoft's Disk Operating System (DOS) for executing the program and managing input/output, and Borland's Turbo Pascal version 5.0 for editing, compiling, and debugging the program. CLIQUE requires approximately 42K bytes of RAM to detect the cliques of one graph, 160 bytes to store one input graph, and 500-1000 bytes to print the results of one graph to an output file. Therefore, the program can be executed on any system which has at least 64K bytes of RAM. Since no special facilities or devices are required to run this program and our system is on the lower end of current technology, any user system advertised as IBM-compatible should be able to execute the program CLIQUE.

The program's data structures require a high-level language capable of representing arrays, both single-dimensional and multi-dimensional. These *simple*

data structures are easily implemented in Pascal. Single-dimensional arrays are used to store the stations' names and number of transmissions (degrees), the stations currently being considered in the search for a clique, and the maximal clique sets. A multi-dimensional array is used to store and manipulate the adjacency matrix of the conflict graph.

The input of the program is supplied by the user. The output of the program is printed to the screen or to a file as desired by the user. Details of these two operations may be found in the User's Guide in Appendix B.

A model of the program's interaction is found in Figure 9. As part of finding the clique, we first determine the vertex degrees of each station from the adjacency matrix input by summing the rows. Once the number of degrees has been determined, the total number of edges in the graph is known and, with the number of stations, we can determine the cliques. First, we fill an array with the stations being considered in the search for a clique. The appropriate subprocedures are then called recursively based on the number of stations and transmissions in the net.

The subprocedures in CLIQUE represent subproblems of detecting a network's cliques. A short introduction to these subprocedures follows. *Findpair* finds a pair of vertices, one of which has degree one, that form a K_2. *Oneedgelessthan* determines the K_2 cliques for a graph consisting of one less edge than its total number of vertices. *Circlekaytwo* determines the K_2 cliques for a graph in which the number of edges equals the number of vertices. *Findtriwithlegs* finds the K_2 cliques which are pendants from a triangle and the triangle itself. *Findatri* will find a triangle given two vertices or indicate its failure to find one. *Findadjnode* finds a node adjacent to a given node and is used as a preliminary to eliminating the edge as a K_2

or recognizing it as part of a three-order or greater clique. As maximal cliques are detected, *writeclique* outputs the sets to the screen or a file as directed by the user. Appendix A has a more detailed explanation of the program CLIQUE and its procedures.

2. Response Time

The network manager expects a response to input in a "reasonable" amount of time, especially if he is using an automated tool such as CLIQUE. Inherent in the clique problem, however, is that response is measured in exponential-time for even moderately sized graphs. It is important, therefore, to illustrate CLIQUE's response is measurable in seconds.

Because the possible combinations of vertices and edge is so numerous, exhaustive testing is not conducted. Verifying *all* possible cases is extremely difficult. For example, isomorphism contributes to the difficulty since each graph on six vertices has six possible adjacency matrices. So we tested CLIQUE's efficiency with a representative eighty graphs on six vertices, twenty-one graphs on five vertices, six graphs on four vertices, two graphs on three vertices and one graph on two vertices.

Using Borland's Turbo Profiler on an IBM-compatible AT, we tested seven files [Ref. 9:pp. 47-49]. This *profiler* measures execution time and "profiles" the time spent in each module of the program. A description of the seven separate test files follows. File number one contains three graphs on two and three vertices. File number two contains six graphs on four vertices. File number three contains twenty-one graphs on five vertices. File number four contains fifty graphs on six vertices. File number five contains thirty different graphs on six vertices. File number six contains a K_2 graph representing one of the smallest graphs and file number seven

contains a graph on six vertices representing one of the largest graphs. *Smallest* and *largest* are based on number of procedure calls during program execution. (A test was run separately on all of the graphs to determine how many times each graph called CLIQUE's different procedures.) These files contain valid input only.

Three tests were run on each file and the average time spent in executing each file is shown in Table 2. The files are read from and written to the same *drive*. Interestingly, the bulk of the execution time (80-90%) is spent reading and writing data for the nets. The time spent in procedure *findclique* (the main procedure which initiates clique detection) was less than one percent of the execution time. Therefore execution time in finding the graphs is minimal compared with processing the matrix. Manipulation of the input access would speed up the program execution but not significantly at this constrained level of input. These statistics are significant in that real-time results are achievable in solving the frequency problem using CLIQUE.

TABLE 2
CLIQUE EXECUTION TIMES

File	1	2	3	4	5	6	7
"A" Drive Execution Time	3.52	5.20	9.20	19.90	14.94	5.00	4.89
"C" Drive Execution Time	0.50	0.63	1.89	4.88	3.07	0.39	0.48

† time is in seconds

3. Error Checking and User Friendliness

Although the program performs error checking, it could use some fine tuning. Errors detected by the program are described in Appendix B. In particular there are

two errors CLIQUE does not detect. It does not check that the adjacency matrix is symmetrical with respect to its rows and columns, nor does it determine whether the graph is composed of more than one component. Currently, the user is responsible for inputting *correct* data, however, the computer's speed and parsing ability is ideally suited to these tasks. So, if the computer inserted the (j, i) entry (when the user entered the (i, j) entry) and/or detected a disconnected graph, the user would save time and avoid deceptive runs.

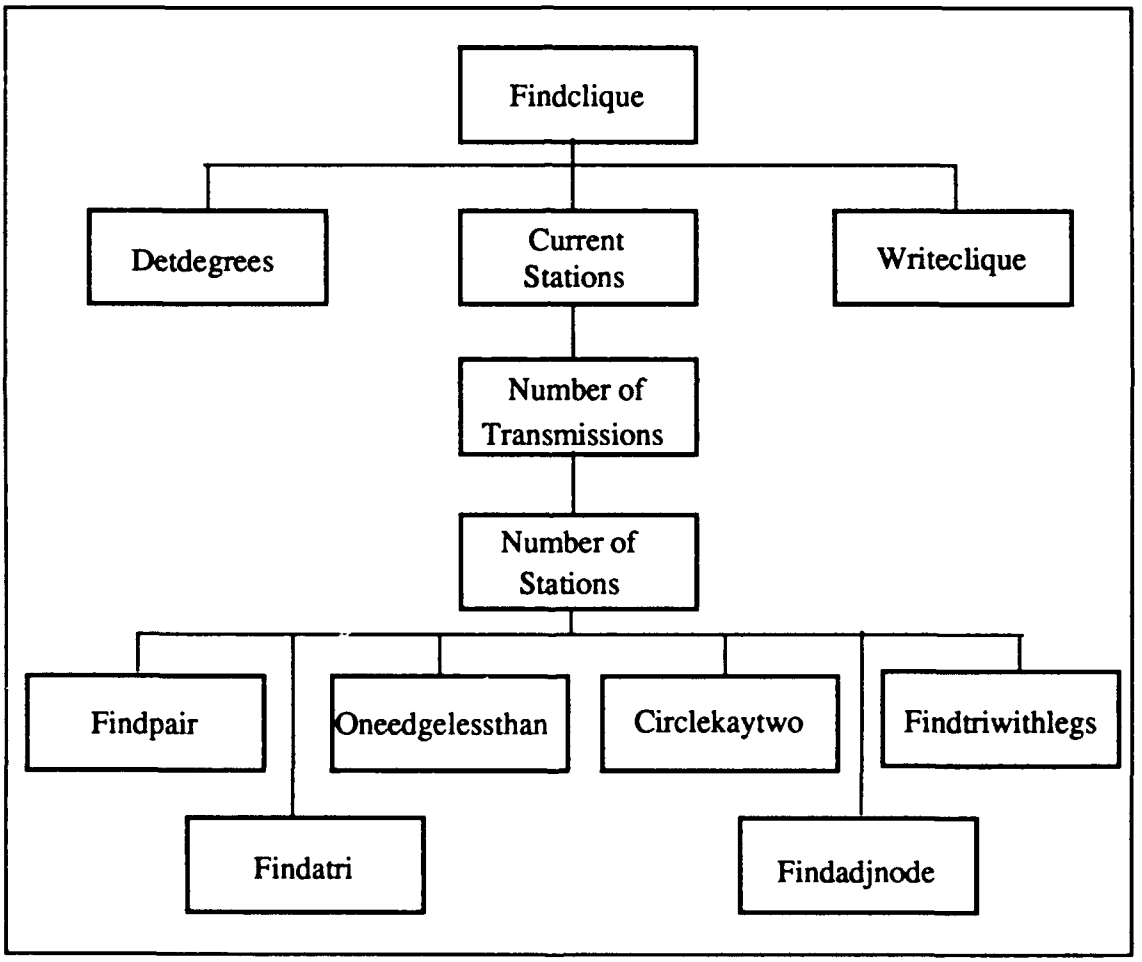


Figure 9 Components of the Program CLIQUE

The program has "friendly" characteristics as follows. The command to invoke CLIQUE is immediately followed by the user's input filename and output filename, saving the user from answering "prompts." If the input filename is not given or cannot be found, an error message reminds the user to include the input filename on the command line. CLIQUE also prints (to the screen or output file) the name of the input file, the number of stations in a net, the adjacency matrix of the net's conflict graph, the names of the stations, the maximal cliques of the net, and the appropriate error messages.

B. CONTRIBUTIONS TO RESEARCH

The program CLIQUE is a minor contribution to the NP-complete problem of detecting cliques. The plentiful amount of research published, some of which was described in Chapter II, is a testament to the difficulty involved in solving NP-complete problems. However, for researchers desiring an understanding of the *clique* problem on a smaller scale, CLIQUE is extremely valuable. Our deliberation, culminating in CLIQUE's procedures, reveals the existence of (or absence of) common characteristics in cliques with maximal degree less than or equal to five. The difficulty in finding the *ECC* of a net stresses the obstacles faced in designing a network with minimal conflict. For communications network managers, CLIQUE is a tool which can be used to maximize the limited frequency assets of a network or networks. However, we emphasize that CLIQUE is a "limited" solution in the NP-complete scenario of the clique problem.

C. SUMMARY

The "goal" is to find the complete subgraphs of a network and correspondingly improve frequency management. CLIQUE provides a sufficient number of error messages and feedback messages to keep the user informed while meeting the

"goal." It is significant as a *working* implementation of the solution to the clique problem, available to the user *now*. To improve upon CLIQUE's solution, however, we need to correct user interface problems and expand the program past the current constraint of six stations.

IV. ANALYSIS OF THE CLIQUE PROGRAM

As mentioned in the introduction to this paper, detecting the maximal cliques of a conflict graph *appears* to be an easy problem. In devising the program CLIQUE, therefore, we started with simple conflict graphs (i.e., those with four vertices) and progressed to graphs with more vertices and edges. We tried many different approaches to solving the problem. This chapter explains our methods of attack and why we decided to implement CLIQUE as we did. Realize that throughout the implementation process, the graph is represented as an adjacency matrix. We assume that the adjacency matrix represents a *valid* conflict graph of a communications network (e.g., is not disconnected, contain loops or multiple edges). Hopefully, the reader will get a feel for the difficulty involved in solving an NP-complete problem.

A. INITIAL APPROACH TO THE PROBLEM

1. The Approach

Initially, we started programming an algorithm for graphs of less than four vertices. By definition, a complete subgraph, K_n, has $(n - 1)$ degrees per vertex. As used by Pullman, we designate the maximum of the degree of G, $\Delta(G)$, equal to $(n - 1)$ [Ref. 2:p. 59]. With this fact in mind, we create a function to read in the number of edges per vertex from the adjacency matrix. This number is known as the degree of a vertex. Once the degree of each vertex is calculated, we search for the vertex with degree equal to $\Delta(G) = (n - 1)$. If such a vertex is found, we continue to search for more vertices of the same degree. We must have n vertices with degree equal to $\Delta(G) = (n - 1)$ before K_n can be the largest possible clique in the graph. If such a

vertex is not found, the search decrements n after each unsuccessful pass of all vertices until a vertex with degree equal to $\Delta(G')$ is found (where G' is a subgraph of G). Once again we search for n vertices of degree $(n - 1)$.

Once the largest possible size of a clique is determined in which G itself is not a complete graph, we must find the vertices which form the clique. Using one of the vertices with degree equal to $\Delta(G')$ and its adjacent vertices (as determined from the original adjacency matrix), we construct a *temporary* adjacency matrix, A'. A' is formed with the edges corresponding to the entries from the original matrix using the vertices above. We then use a *procedure* to determine whether A' forms a maximal clique. This procedure calculates the degrees of each vertex of A', compares the degrees of each vertex to $\Delta(G')$, and decides whether the clique is maximal. If the clique is maximal, it is output in set notation and $\Delta(G')$ is subtracted from the degrees of each vertex. (This subtraction eliminates the clique's edges from further consideration in the clique problem. Remember that a clique is not maximal if it is a complete subset of another clique.) If A' does not produce a maximal clique, then another vertex with degree equal to $\Delta(G')$ is picked from the remaining vertices and the preceding process is repeated. Figure 10 contains examples of graphs in which the cliques can and cannot be found using the initial approach. In the program CLIQUE, (a) is in the class known as one-edge-less-than-number-of-vertices and (b) is in the class known as a-triangle-with-legs. These classes can be found in graphs of all sizes (see Appendix A).

2. Observations and Conclusions

Our initial approach did not consider that an edge can be a component of more than one clique (see edge {x, y} in Figure 10(c)). Therefore, it was invalid to delete an edge from further consideration based *solely* upon membership in one clique.

Additionally, since a *linear* search is performed within the adjacency matrix for adjacent vertices, multiple *valid* and *invalid* temporary matrices were often calculated. A technique is needed to eliminate these "extra" cliques and decrement n at the end of a pass to form a new $\Delta(G')$. Thus, a clique which is a component of a larger clique may be erroneously output as maximal. These linear search problems indicate that an additional discriminator, in addition to vertex degrees, is needed to determine cliques.

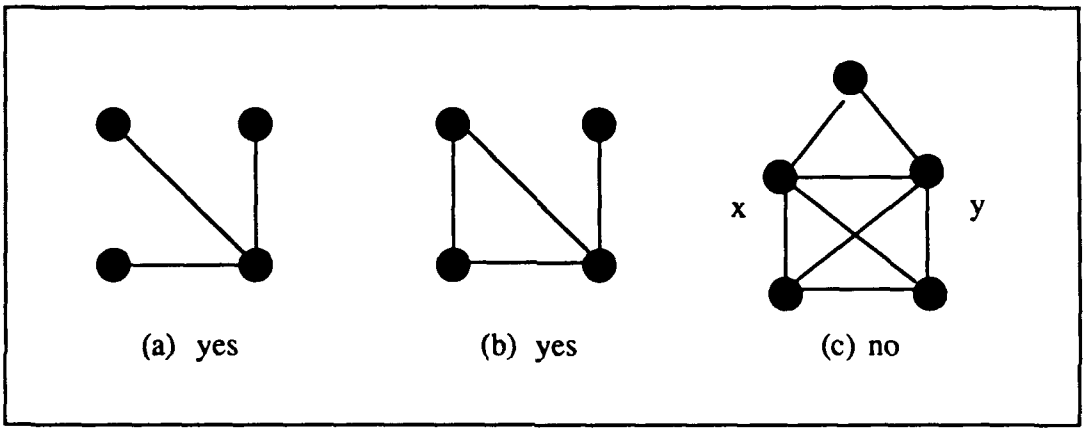


Figure 10 Graphs Used in Initial Programming

B. INTERMEDIATE APPROACH TO THE PROBLEM

1. The Approach

At this point, the most pressing problem is to determine which edges are components of multiple cliques. We proceed beyond arbitrary graphs less than or equal to four vertices, to graphs with up to nine vertices. As mentioned previously, we can *look* at a drawing of most graphs of less than ten vertices and separate the graph's maximal cliques fairly easily. In the program CLIQUE, however, any approach is affected by the computer's limitation of reading and interpreting the ones

and zeros of an adjacency matrix in order to glean the graph's configuration. Therefore, we create some data structures to act as "flags" when an edge is used in a clique. Flags are delineators when an edge is a member of more than one maximal clique. The flags distinguish between the two cliques of different type K_n (see Figure 10(b)) and the four cliques of the same K_n type (see Figure 11(a) and (b)) which are sharing edges. However, flags do not delineate the multiple cliques, of either the same or different type K_n, which are sharing edges in Figure 11(c). This failure is due to the fact that an edge, e = (x, y), may belong to one, two, three, or more cliques (see Figure 11(d)). An exhaustive search may be the only method of determining this, as flags cannot be manipulated to give the correct result every time.

2. Observations and Conclusions

It is apparent that flags have limited usefulness. In an arbitrary graph, every edge must be searched from every vertex for every possible clique. This "technique" is Tarjan's "obvious" algorithm and is not solvable in linear time [Ref. 6:p. 1]. Once the graph has more than five vertices, the patterns disappear since there is an increasing number of possible edge combinations among six or more vertices. Isomorphism also affects the detection of cliques. Since it is difficult to cover every possible case, the "algorithm" becomes more like an "heuristic." It finds the cliques of certain adjacency matrices but does not find the cliques of the same graph represented in a different adjacency matrix.

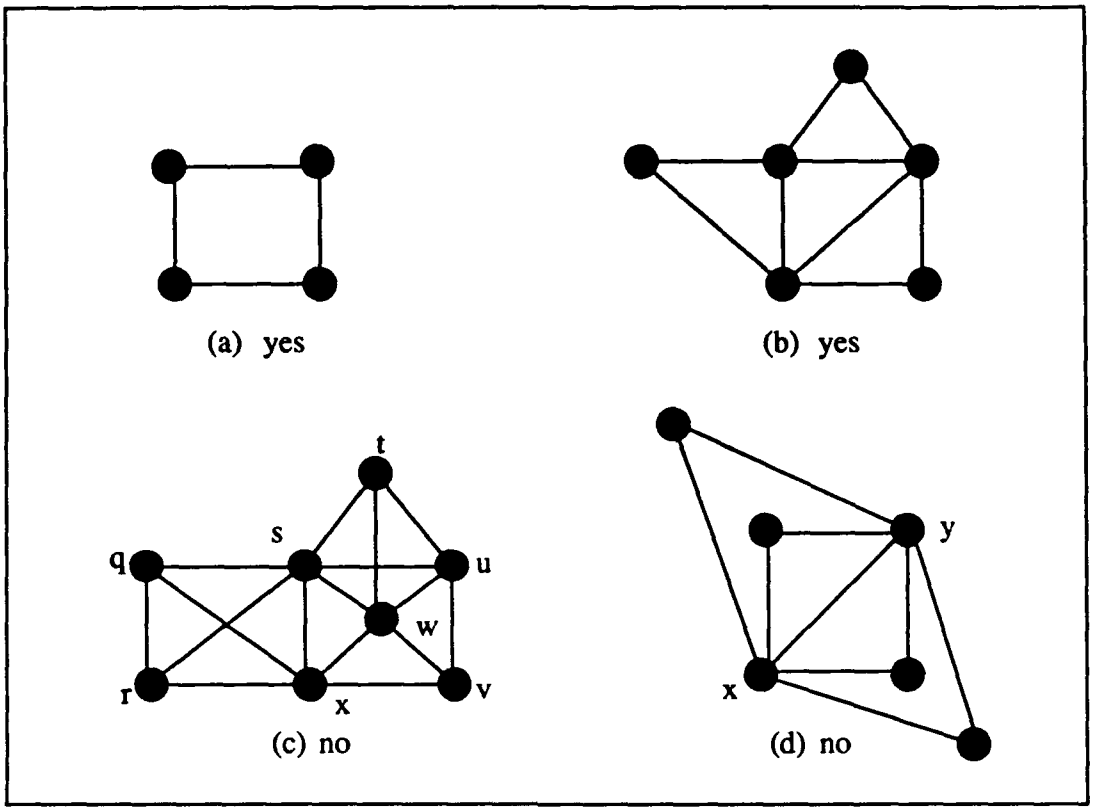


Figure 11 Graphs Used in Intermediate Programming

We need a method to *distinguish* those vertices which are on the outside of a graph and members of a single clique from those vertices which are components of more than one clique. Figure 11(c) illustrates this problem where {q, r, t} are members of a single clique, and the other vertices belong to multiple cliques. The K_4 cliques C_1 = { q, r, s, x } and C_2 = {s, t, u, w } are easily identified by the current algorithm. The K_3 cliques C_3 = { u, v, w } and C_4 = { v, w, x } are found with the flag modification and adjustment to the method of subtraction of degrees from the vertices' total. The K_3 clique C_5 = { s, x, w } cannot be found with this approach because its components have been found in the other cliques and there is no pattern which

distinguishes these edges and keeps the algorithm from eliminating them from further clique consideration. Therefore, we must limit the scope of the problem and minimize the number of vertices and possible graphs.

C. FINAL APPROACH TO THE PROBLEM

1. The Approach

We decide to constrain and limit the solution to the communications problem in order to represent most (if not all) graphs up to six vertices. We start with the graphs in Figure 12 and add one edge to each graph progressively until we have a collection of graphs. Many of the graphs we obtain are isomorphic to others, resulting in "redundant" graphs. Eliminating the redundant graphs is very tedious. To alleviate the tedium and possible error, we calculate the degrees of the vertices on each graph and use that, along with the number of edges, as an *indicator* of isomorphism. However, the number of degrees and edges are not the sole indicators of equality. As can be seen in Figure 13, two graphs can have the same degrees and edges but the first has a maximal clique K_2 and the second has a maximal clique K_3. Once the graphs are drawn, we categorize each by the maximum size clique in the graph. The idea behind the categories is to ensure that redundant graphs have been eliminated and to emphasize similarities between graphs. This organization of graphs makes it easier to program the clique problem.

In implementing the detection of cliques, we start with the vertices of degree one because this edge is easily eliminated from further consideration and can be printed as a set K_2. Eliminating the vertices and edges which are members of a single clique is beneficial for finding the graph's remaining cliques as it reduces the network. However, if the smallest degree of a graph's vertices is two and there are

no triangles or there is a mixture of K_2 and K_3 cliques, a check must first be performed to determine which K_i you have. Thus, to ensure that the user can input arbitrary graphs, we consider the graphs separately and program *individual* graphs. The program CLIQUE is composed as discussed in Chapter III and the complete program is found in Appendix A.

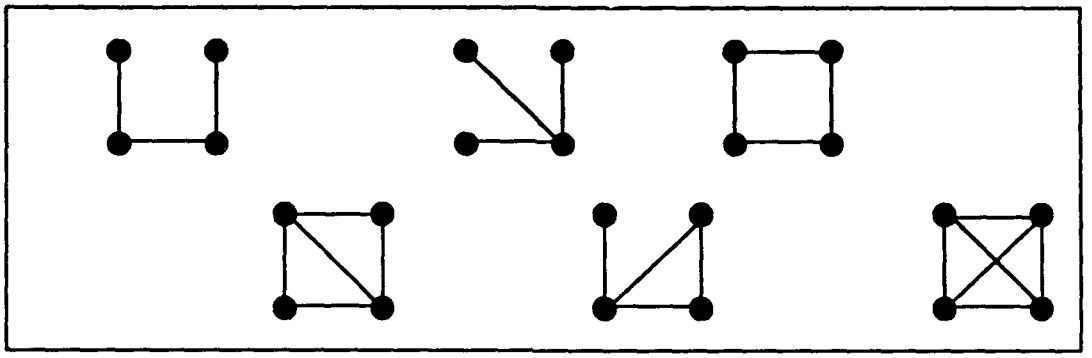


Figure 12 Graphs on Four Vertices

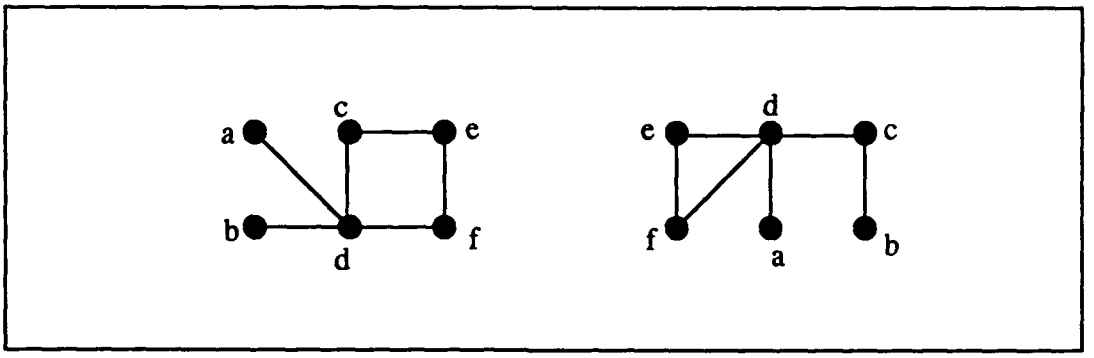


Figure 13 Non-Isomorphic Graphs with the Same Degrees Per Vertex

2. Observations and Conclusions

Finally, we understand why in this research, graphs have been limited to degree of four or less and why the heuristic approach is so popular. It is far easier to find a *representative* number of cliques which cover the edges. Although it is not

guaranteed that all possible graphs up to and including six vertices have been found, the process of drawing the graphs helps to delineate patterns, to understand isomorphism, and to comprehend the magnitude of NP-complete problems.

CLIQUE performs well for arbitrary graphs up to six vertices. Because each graph is coded for specific characteristics, cliques are located quickly. CLIQUE eliminates the problems of isomorphism and duplication of cliques, which saves on the calculating time expended in backtracking heuristics.

D. SUMMARY

The process of developing the program CLIQUE is representative of the complexity involved in solving an NP-complete problem. The exponential number of edge and vertex combinations, for even one additional vertex beyond the six programmed here, is potentially "mind boggling" for a single programmer. To get an idea of just how overwhelming the process is, take any graph on six vertices defined in CLIQUE, add a seventh vertex to the graph, and add an edge between it and a distinct vertex. Then add an edge between the seventh vertex and any *other* distinct vertex in the graph which results in a *different* graph from the one formed by adding the first edge. You will form from one to six different graphs. Now repeat this procedure for *every* graph on six vertices. Since there are 121 graphs identified on six vertices in this thesis, you will have at least that many new graphs on seven vertices and you haven't even added a second edge from the seventh vertex! Of course you will draw more than 121 graphs initially and must determine which are isomorphic and must be eliminated.

This chapter attempts to illustrate the difficulty which arises in finding the identical ECC for a graph, *despite* dissimilar graphical representation. Hopefully the steps used to arrive at the final version of CLIQUE clarify the difficulty in finding the maximal cliques for any *one* graph at any *one* time when that graph is *arbitrary*. We believe that the more patterns and similarities which are found between graphs of

different sizes, the closer we will be able to come to solving the clique problem. Computers are useful for implementing "patterns." Patterns are usually implemented in a "recursive" or " iterative" style, particularly suited to automation. Thus, part of the solution to solving the clique problem may lie in finding a pattern among arbitrary graphs which can then be programmed through an algorithm or heuristic.

V. CONCLUSIONS AND RECOMMENDATIONS

A. IMPROVEMENTS AND FUTURE RESEARCH

The program CLIQUE may be improved or expanded by anyone interested in the clique problem or the communications problem. Some suggestions follow.

As mentioned in Chapter III, there are some error detection measures which can be added to CLIQUE. Currently, CLIQUE does not find the graphs illustrated in Appendix C. The graphs can be coded and integrated into the existing program.

The number of non-isomorphic graphs on six or fewer vertices should be explored. Any graph which is not already included in CLIQUE should be added. A proof of completeness will then be necessary.

If the requirement that one program be able to handle *any* arbitrary graph is changed, then programs can be implemented to handle "classes" of graphs. Graphs will then be entered into a *program* for a particular class, which may have specific characteristics that reduce the overall computation time. However, if the user must classify the graphs manually, the purpose of automating the detection of cliques may be defeated. Therefore, it is desirable to automate the classification process itself.

Further research should analyze CLIQUE and modularize it more completely. Although CLIQUE does call some subsets of a graph recursively when detecting a clique, there is room for improvement. Particularly in the latter part of the program during the coding of graphs which have both K_2 and K_3 cliques (but not K_4 cliques). Because a vertex with degree two may be in either *two* K_2 cliques or *one* K_3 clique, we must test for triangularity. Testing for a triangle helps to "classify" the graph and determine which edges and degrees to eliminate from consideration as candidates for

future cliques. This specific test is not currently implemented in a module (procedure) which can be called as a case requires.

It is tedious work to dissect graphs. Automating this process would enable a researcher to analyze graphs greater than six vertices and further identify and categorize graphical patterns which can possibly contribute to the solution of the clique problem.

B. CONCLUSIONS AND OBSERVATIONS

The program CLIQUE finds a constrained set of maximal cliques efficiently. Although we have not analyzed CLIQUE's time complexity, its performance is measurable in real-time, as done on Borland's Turbo Profiler. Therefore, efficiency is claimed in the speed at which the program detects the maximal cliques for a net.

CLIQUE can be very useful in communications planning. With an input of six stations or less, it finds all possible maximum subnets of the network (known as maximal cliques in this thesis) and analyzes them for conflict. Conflict analysis contributes to the frequency management subproblem by identifying stations which can increase transmissions without new frequency allocation. This is important to the military communications planner, in particular, since frequency constraints are world-wide.

Since CLIQUE analyzes the nets using the number of stations, number of transmissions per station, and number of transmissions per net, the calculation of duplicate valid and invalid cliques within the net is avoided. Avoiding repetitive searches saves time in detecting cliques and gets the correct answer to the planner faster. Additionally, CLIQUE is not sensitive to the nuances of the adjacency matrix so the detection of cliques is not affected by isomorphism. Therefore, we will find the same clique sets of a graph despite a different matrix representation. Used in

conjunction with another researcher's algorithm or by itself, CLIQUE is an effective tool within its limitations.

In general, this paper emphasizes the extreme difficulty inherent in implementing an NP-complete problem such as the clique problem. Since CLIQUE's input is limited to six stations, this thesis has limited the scope of the problem to a finite set of possible graphs. CLIQUE is able to solve the edge clique covering problem because of the finiteness of the problem space. Other research, on the other hand, indicates that heuristics, which may or may not give the "best" answer, suffice in some applications. However, whether the researcher uses an algorithm or an heuristic, the fact remains that there are many possible "nets" for a large number of stations. There are currently no time-efficient algorithms to use which categorize or classify the characteristics of each.

APPENDIX A

CLIQUE : THE PROGRAM

{ **FILE NAME** : CLIQUE.PAS }

{**

AUTHOR : Kristi J. Bell
CREATED : 25 September 1989
REVISED : 18 May 1990
MACHINE/COMPILER : IBM compatible XT/ Borland's Turbo Pascal
PURPOSE :

This Program will determine the complete set of maximal cliques in any graph consisting of one to six vertices. The graph should represent a communications network with the vertices representing nodes. A complete set will include all cliques of maximum size to which the vertices belongs. Therefore, a vertex may belong to more than one clique. This program does not include those graphs consisting of six vertices and either twelve, thirteen , or fourteen edges. Programming of these three cases will be left for future projects.

INPUT :

A file representing the communications nodes as vertices which includes : total number of nodes, names of the nodes, and an adjacency matrix representation of the communications network.

OUTPUT :

The names of each communication's node in its associated maximum clique set. Also printed is the input filename and the graph's adjacency matrix representation.

PROCEDURES :

findclique: Finds a maximal clique using the following nested procedures.

NESTED IN : findclique
detdegrees : Determines the degrees of each vertex.
filltmpnum : Fills an array with vertices being considered in search for a clique.
findnode : Finds the vertex of a specific degree.
findpair : Finds a pair of vertices, one of which has degree one.
oneedgelessthan: Determines the K_2 cliques for a graph with one less edge than total number of vertices.

circlekaytwo : Determines the K_2 cliques for a graph in which the number of edges equals the number of vertices.
findtriwithlegs : Finds the K_2 cliques that dangle from a triangle.
findatri : Given two vertices, will find a triangle or return a failure boolean.
findadjnode : Given a node, finds an adjacent node.
twovert : Calculates the maximal cliques for a graph of two vertices.
threevert : Calculates the maximal cliques for a graph of three vertices.
fourvert : Calculates the maximal cliques for a graph of four vertices.
fivevert : Calculates the maximal cliques for a graph of five vertices.
sixvert : Calculates the maximal cliques for a graph of six vertices.

INCLUDE FILES :
FILE NAME : MATRINC.PAS
FUNCTIONS :
leftalign : Left-justifies a string.
spaces : Adds spaces to the end of a string.
PROCEDURE :
makematrix : Makes an adjacency matrix from an input string.
FILE NAME : WRITINC.PAS
PROCEDURE :
writeclique : Writes each maximal clique to an output file.

**********}

```
{*************************************************************************

 PROGRAM : clique

 CALLS   : makematrix
           findclique

 PURPOSE : This program will determine the complete set of maximal cliques
           from any input graph.

*************************************************************************}

program clique (input, output);

   { Compiler Directive for the inclusion of  files }

{$I matrinc}                    { Reads in file and makes adjacency matrix   }
{$I writinc}                    { Writes the node names in max clique sets   }
                                { To the given output file                   }

var
  matrixfile,                   { Input file                                 }
  cliquefile : text;            { Output file                                }
  nextparam : string;           { Output file parameter entered by user      }
  totnumvertices : string[1];   { Number of vertices in a graph              }
  i,                            { Incremental  counter                       }
  notvertex,                    { If zero, then vertice number is an integer }
  graphnumber,                  { Sequential number of graph in input file   }
  edgenum,                      { Counts number of edges in current matrix   }
  vertex : integer;             { Number of total vertices in adj matrix     }
  notvalid : boolean;           { Entered invalid edge number                }
```

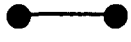
```
{*************************************************************************

 PROCEDURE : findclique

 CALLED BY  : clique

 CALLS      : detdegrees
              twovert
              threevert
              fourvert
              fivevert
              sixvert

 PURPOSE    : Analyzes the vertex input and determines the maximal cliques
              based on number of vertices, number of edges and number of
              degrees for each vertex.

*************************************************************************}

procedure findclique ( avertex, numedges: integer; matrix: matrixtype;
                       var cliquefile: text );
const
  degone = 1;                          { Number of degrees per vertex      }
  degtwo = 2;
  degthree = 3;
  degfour = 4;
  degfive = 5;
  degsix = 6;

var
  outclique,                           { Number of completed max cliques   }
  node,                                { Node of degree being searched for }
  i, j, k : integer;                   { Increment counters                }
  maxclique : nametype;                { Holds one maximum clique          }
  match : boolean;                     { Matches current node being looked for }
  tmpnum,                              { List of vertices being manipulated }
  outdeglist : column;                 { List of  vertex degrees           }
```

```
{*****************************************************************

 PROCEDURE : detdegrees

 CALLED BY  : findclique

 PURPOSE    : To determine the number of degrees of each vertex.

*****************************************************************}

procedure detdegrees ( somematrix: matrixtype; var deglist, nodenum:
                       column; numvertex: integer );
var
  deg,                          { Degrees of each vertex          }
  j, k : integer;               { Incremental counters            }

begin  { detdegrees }
 k := 1;
  while (k <= numvertex) do     { Searches every vertex and totals up the  }
   begin                        { Degrees by counting ones (equal to edges) }
    deg := 0;                   { In the graph's adjacency matrix  }
    for j := 1 to numvertex do
     deg := deg + somematrix[nodenum[k], nodenum[j]];
    deglist[k] := deg;
    k := k + 1;
   end;
end;  { detdegrees }
```

```
{*****************************************************************

  PROCEDURE : filltmpnum

  CALLED BY   : circlekaytwo
                findtriwithlegs
                fourvert
                fivevert
                sixvert

  PURPOSE     : To fill an array with the verticesof the next possible
                maximal clique. Done prior to writing out a complete
                graph.

*****************************************************************}

procedure filltmpnum ( deglist: column; var tmpnum: column );

var
 i, j : integer;                    { Incremental counters                 }

begin   { filltmpnum }
 j := 1;
 for i := 1 to avertex do           { Searches current array of vertex     }
  begin                             { Degrees and compacts those not       }
   if deglist[i] <> 0               { Equal to zero (still have edges)     }
    then begin
     tmpnum[j] := i;
     j := j + 1;
    end;
  end;
end;    { filltmpnum }
```

```
{*****************************************************************

  PROCEDURE : findnode

  CALLED BY  : oneedgelessthan
               findtriwithlegs
               fourvert
               fivevert
               sixvert

  PURPOSE    : Finds a node of a particular degree, else returns false.

*****************************************************************}

procedure findnode ( outdeglist: column; var node: integer;
                     subclique: integer; var match: boolean);

begin   { findnode }
 if (node < avertex)              { Searches the degree list of each      }
  then begin                      { Vertex                                }
   repeat
    node := node + 1;
    if outdeglist[node] = subclique
     then match := true;
   until ((match = true) or (node = avertex));
  end;
end;    { findnode }
```

```
{************************************************************

 PROCEDURE : twovert

 CALLED BY : findclique
             oneedgelessthan
             circlekaytwo
             findtriwithlegs
             fourvert
             fivevert
             sixvert

 CALLS     : writeclique

 PURPOSE   : Determines the vertex names, calls a procedure to
             write the K2 maximal clique, and increments the
             maximal clique counter.

************************************************************}

procedure twovert ( var outclique: integer; var maxclique: nametype;
                    tmpnum: column; var maxcliquefile: text);
const
 vertices = 2;                    { Number of vertices in K2 clique    }

var
 i : integer;                     { Incremental counter                }

begin   { twovert }
```



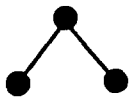
ECC(G) = 1

```
 for i := 1 to vertices do
    maxclique[i] := vertexname[tmpnum[i]];
 writeclique (maxclique, vertices, outclique, maxcliquefile);
 outclique := outclique + 1;
end;    { twovert }
```

```
{*************************************************************

  PROCEDURE : findpair

  CALLED BY  : oneedgelessthan
               circlekaytwo
               findtriwithlegs
               fourvert
               fivevert
               sixvert

  PURPOSE    : Finds a pair of vertices. One is a vertex of degree one
               and the other vertex is adjacent to it.

*************************************************************}
procedure findpair ( var outdeglist, tmpnum: column; matrix: matrixtype;
                     node: integer );
var
 j,k : integer;                    { Incremental counter                  }
 done : boolean;                   { True when found the adjacent vertex  }

begin   { findpair }
 j := 0;
 k := 2;
 done := false;
 repeat
  j := j + 1;
                                   { Finds an adjacent node that can still }
                                   { Be considered in the clique because   }
                                   { The vertex still has edges            }
  if (matrix[node,j] = 1) and (outdeglist[j] >= 1)
   then begin
    tmpnum[k] := j;
    done := true;
   end;
 until (done = true);
                                   { Subtract an edge from the overall     }
                                   { Degrees for the vertex just used,     }
                                   { So the pair won't be used again       }
 outdeglist[node] := outdeglist[node] - 1;
 outdeglist[j] := outdeglist[j] - 1;
end;   { findpair }
```

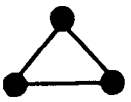
```
{************************************************************

 PROCEDURE  : oneedgelessthan

 CALLED BY  : threevert
              circlekaytwo
              fourvert
              fivevert
              sixvert

 CALLS      : findnode
              findpair
              twovert

 PURPOSE    : Finds and outputs the K2 cliques which result
              from a graph which has one less edge than number
              of vertices.

************************************************************}
procedure oneedgelessthan ( var outclique: integer; var maxclique:
                            nametype; tmpnum, outdeglist: column;
                            matrix: matrixtype; var maxcliquefile:
                            text );
var
 matched : boolean:          { Found a K2 clique                 }
 node : integer              { Any vertice in graph              };

begin  { oneedgelessthan }
 repeat
  matched := false;
  node := 0;
  findnode (outdeglist, node, degone, matched);
  tmpnum[1] := node;
  if matched = true          { Loops until it finds all the K2's }
   then begin
    findpair (outdeglist, tmpnum, matrix, node);
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
   end
  until (matched = false);
end;  { oneedgelessthan }
```

```
{*************************************************************

  PROCEDURE : threevert

  CALLED BY  : findclique
               findtriwithlegs
               fourvert
               fivevert
               sixvert

  CALLS      : oneedgelessthan
               writeclique

  PURPOSE    : Finds all max cliques consisting of three vertices.
               Uses number of edges and degrees as discriminators.

*************************************************************}
procedure threevert ( var outclique: integer; var maxclique: nametype;
                      tmpnum: column; numedges: integer; var maxcliquefile:
                      text );
var
 i, vertices : integer;

begin   { threevert }
```



ECC(G) = 2

```
 case numedges of
  2 : oneedgelessthan (outclique, maxclique, tmpnum, outdeglist,
                       matrix, maxcliquefile);
```



ECC(G) = 1

```
  3 : begin
       vertices := 3;
```

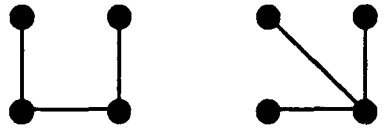
```
            for i := 1 to vertices do
              maxclique[i] := vertexname[tmpnum[i]];
            writeclique (maxclique, vertices, outclique, maxcliquefile);
            outclique := outclique + 1;
          end;
      end;
    end;    { threevert }
```

```
{*************************************************************

  PROCEDURE : circlekaytwo

  CALLED BY : findtriwithlegs
              fourvert
              fivevert
              sixvert

  CALLS     : findpair
              twovert
              filltmpnum
              oneedgelessthan

  PURPOSE   : Finds the K2 cliques indicative of graphs which have
              the same number of edges and vertices. The graphs
              appear circular.

*************************************************************}

procedure circlekaytwo ( outdeglist, tmpnum: column; var matrix:
                         matrixtype; outclique: integer;
                         var maxcliquefile: text );
var
 node : integer;

begin  { circlekaytwo }
 node := tmpnum[1];
                  { Finds a pair of vertices and removes the edges  }
                  { From the adj matrix after printing out the clique                }
 findpair (outdeglist, tmpnum, matrix, node);
 twovert (outclique, maxclique, tmpnum, maxcliquefile);
 matrix[tmpnum[1],tmpnum[2]] := matrix[tmpnum[1],tmpnum[2]] - 1;
 matrix[tmpnum[2],tmpnum[1]] := matrix[tmpnum[2],tmpnum[1]] - 1;

                  { Finds the remaining K2 cliques                                   }
 oneedgelessthan (outclique, maxclique, tmpnum, outdeglist, matrix,
                  maxcliquefile);
end;  { circlekaytwo }
```

```
{************************************************************

 PROCEDURE : findtriwithlegs

 CALLED BY  : fivevert
              sixvert

 CALLS      : findpair
              twovert
              findnode
              filltmpnum
              threevert
              circlekaytwo

 PURPOSE    : Finds the triangles with K2 pendants or
              determines that a triangle does not exist and that
              the rest of the graph is K2 cliques.

************************************************************}
procedure findtriwithlegs ( outdeglist, tmpnum: column; node,
                            currentvertices, newnumedges: integer;
                            maxclique: nametype; matrix: matrixtype;
                            var maxcliquefile: text );
const
 triangle = 3;

var
 matched : boolean;

begin  { findtriwithlegs }
 repeat                              { Loops until all vertices with degree   }
                                     { One are found and written out          }
  tmpnum[1] := node;
  findpair (outdeglist, tmpnum, matrix, node);
  twovert (outclique, maxclique, tmpnum, maxcliquefile);
  newnumedges := newnumedges - 1;
  node := 0;
  matched := false;
  findnode (outdeglist, node, degone, matched);
 until (matched = false);
                                     { Puts the remaining vertices in an      }
                                     { Array and either prints out the        }
```

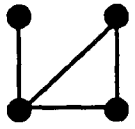
```
                                    { Triangle or finds the circular clique          }
 filltmpnum (outdeglist, tmpnum);
 if (newnumedges = triangle)
   then threevert (outclique, maxclique, tmpnum, newnumedges,
            maxcliquefile)
 else circlekaytwo (outdeglist, tmpnum, matrix, outclique, maxcliquefile);
end;    { findtriwithlegs }
```

```
{*************************************************************

 PROCEDURE : fourvert

 CALLED BY : findclique
             fivevert
             sixvert

 CALLS     : oneedgelessthan
             findnode
             findpair
             filltmpnum
             twovert
             threevert
             circlekaytwo
             writeclique

 PURPOSE   : Finds all maximal cliques in graphs of four vertices.
             Uses number of edges and degrees as discriminators.

*************************************************************}
procedure fourvert ( var outclique: integer; var maxclique: nametype;
                     tmpnum: column; numedges: integer;
                     var maxcliquefile: text );
var
 i , j,                          { Incremental counters                 }
 curvertices,                    { Number of vertices in current graph  }
 newnumedges : integer;          { Reduced number of edges after        }
                                 { Certain cliques removed              }
 matched : boolean;              { True if node is found of that degree }

begin  { fourvert }
 curvertices := 4;
 matched := false;
 node := 0;
 case numedges of
```



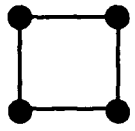
ECC(G)=3 *ECC(G)=3*

```
3 : oneedgelessthan (outclique, maxclique, tmpnum, outdeglist, matrix,
            maxcliquefile);
4 : begin
    findnode (outdeglist, node, degone, matched);
    if matched = true
     then begin
```



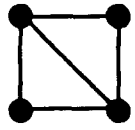
ECC(G)=2

```
      tmpnum[1] := node;
      findpair (outdeglist, tmpnum, matrix, node);
      twovert (outclique, maxclique, tmpnum, maxcliquefile);
      newnumedges := 3;
      filltmpnum (outdeglist, tmpnum);
      threevert (outclique, maxclique, tmpnum, newnumedges, maxcliquefile);
     end
```



ECC(G) = 4

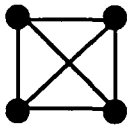
```
    else circlekaytwo (outdeglist, tmpnum, matrix, outclique,
                maxcliquefile);
  end;
5 : begin
```



ECC(G) = 2

```
    findnode (outdeglist, node, degtwo, matched);
    tmpnum[1] := node;
    node := 0;
    matched := false;
    findnode (outdeglist, node, degthree, matched);
    tmpnum[2] := node;
```

```
        matched := false;
        findnode (outdeglist, node, degthree, matched);
        tmpnum[3] := node;
        newnumedges := 3;
        threevert (outclique, maxclique, tmpnum, newnumedges, maxcliquefile);
        node := tmpnum[1];
        matched := false;
        findnode (outdeglist, node, degtwo, matched);
        tmpnum[1] := node;
        threevert (outclique, maxclique, tmpnum, newnumedges, maxcliquefile);
      end;

  6 : begin
```



ECC(G) = 1

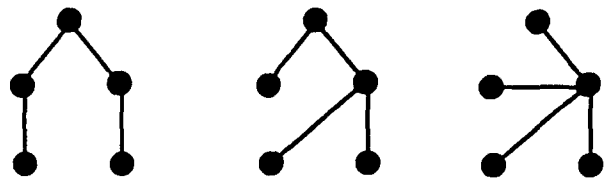
```
      for i := 1 to curvertices do
        maxclique[i] := vertexname[tmpnum[i]];
      writeclique (maxclique, curvertices, outclique, maxcliquefile);
      outclique := outclique + 1;
    end;
 end; {case}
end;    { fourvert }
```

```
{*************************************************************

 PROCEDURE : findatri

 CALLED BY  : fivevert
              sixvert

 CALLS      : threevert

 PURPOSE    : Takes two given nodes and finds a third
              adjacent node which forms a triangle.

*************************************************************}
procedure findatri ( var tmpnum: column; matrix:
                     matrixtype; var outclique: integer; var
                     maxclique: nametype; var done: boolean;
                     var maxcliquefile: text );
var
 i,                              { Incremental counter                 }
 numedges : integer;             { Number of edges in a triangle       }

begin   { findatri }
 i := 1;
 done := false;
 repeat                          { Loop until find a node in the matrix }
                                 { Which is adjacent to both given     }
                                 { Vertices, else return a false       }
  if (matrix[tmpnum[1],i] = 1) and
    (matrix[tmpnum[2],i] = 1)
   then begin
     tmpnum[3] := i;
     done := true;
   end;
  i := i + 1;
 until ((done = true) or (i > avertex));
                                 { If find a triangle, print it out    }
 if (done = true)
  then begin
    numedges := 3;
    threevert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
  end;
end;    { findatri }
```

```
{*************************************************************

 PROCEDURE : findadjnode

 CALLED BY  : fivevert
              sixvert

 PURPOSE    : Finds a separate node adjacent to the current node.

*************************************************************}
procedure findadjnode ( node: integer; matrix: matrixtype; var
                        tmpnum: column );
var
 done : boolean;
 i, j : integer;

begin   { findadjnode }
 tmpnum[1] := node;
 j := 1;
 done := false;
 repeat                          { Loops until an adjacency is located,   }
                                 { Then both nodes are passed back        }
                                 { And as part of a larger clique         }
  if matrix[node,j] = 1
  then begin
   tmpnum[2] := j;
   done := true;
  end;
  j := j + 1;
 until (done = true);
end;    { findadjnode }
```

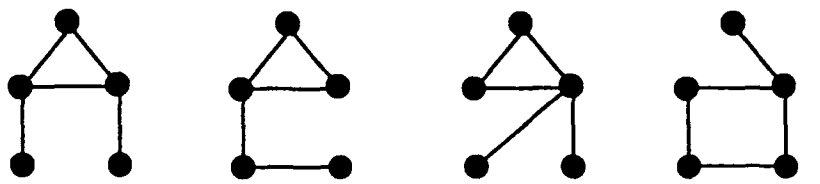
```
{*************************************************************

 PROCEDURE : fivevert

 CALLED BY : findclique
             sixvert

 CALLS     : oneedgelessthan
             findnode
             findtriwithlegs
             circlekaytwo
             findpair
             twovert
             filltmpnum
             fourvert
             findadjnode
             threevert
             findatri
             writeclique

 PURPOSE   : Finds all maximal cliques in graphs of five vertices.
             Uses number of edges and degrees as discriminators.

*************************************************************}
procedure fivevert ( var outclique: integer; var maxclique: nametype;
                     tmpnum: column; numedges: integer; matrix:
                     matrixtype; var maxcliquefile: text );
var
 matched, success : boolean;
 i, j, temp,
 counter,
 node, curvertices : integer;

begin   { fivevert }
 curvertices := 5;
 node := 0;
 matched := false;
```

```
case numedges of
```



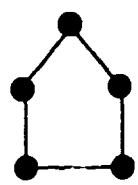
ECC(G) = 4

```
4 : oneedgelessthan (outclique, maxclique, tmpnum, outdeglist,
                     matrix, maxcliquefile);

5 : begin
```



ECC(G) = 3 *ECC(G) = 5*

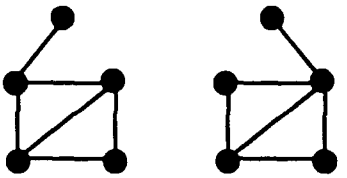
```
    findnode (outdeglist, node, degone, matched);
    if matched = true
      then findtriwithlegs (outdeglist, tmpnum, node, curvertices,
                            numedges, maxclique, matrix,
                            maxcliquefile)
```



ECC(G) = 5

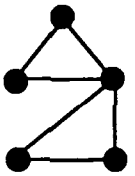
```
      else circlekaytwo (outdeglist, tmpnum, matrix, outclique,
                         maxcliquefile);
    end;
```

6 : begin



ECC(G) = 3

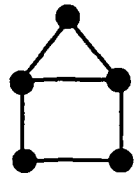
```
findnode (outdeglist, node, degone, matched);
if matched = true
 then begin
  tmpnum[1] := node;
  findpair (outdeglist, tmpnum, matrix, node);
  twovert (outclique, maxclique, tmpnum, maxcliquefile);
  filltmpnum (outdeglist, tmpnum);
  numedges := numedges - 1;
  fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
 end
else begin
```



ECC(G) = 2

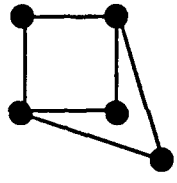
```
 node := 0;
 matched := false;
 findnode (outdeglist, node, degfour, matched);
 if matched = true
  then begin
   findadjnode (node, matrix, tmpnum);
   findatri (tmpnum, matrix, outclique, maxclique, success,
         maxcliquefile);
     for i := 1 to 3 do
      outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 2;
     filltmpnum(outdeglist, tmpnum);
     numedges := 3;
     threevert (outclique, maxclique, tmpnum, numedges,
           maxcliquefile);
   end {then}
  else begin
```

```
node := 0;
matched := false;
findnode (outdeglist, node, degthree, matched);
tmpnum[1] := node;
matched := false;
findnode (outdeglist, node, degthree, matched);
tmpnum[2] := node;
if (matrix[tmpnum[1],tmpnum[2]] = 1)
 then begin
```



ECC(G) = 4

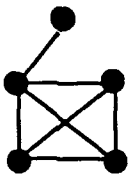
```
  findatri (tmpnum, matrix, outclique, maxclique,
        success, maxcliquefile);
  for i := 1 to 3 do
   outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 2;
  for i := 1 to 2 do
   for j := 1 to 3 do
    matrix[tmpnum[i], tmpnum[j]] := 0;
  filltmpnum (outdeglist, tmpnum);
  oneedgelessthan (outclique, maxclique, tmpnum,
            outdeglist, matrix, maxcliquefile);
 end
else begin
 temp := tmpnum[2];
 node := tmpnum[1];
```



ECC(G) = 6

```
 repeat
  repeat
   findpair (outdeglist, tmpnum, matrix, node);
   twovert (outclique, maxclique, tmpnum, maxcliquefile);
```

```
            matrix[tmpnum[1],tmpnum[2]] := 0;
            matrix[tmpnum[2],tmpnum[1]] := 0;
           until (outdeglist[node] = 0);
           node := temp;
           tmpnum[1] := node;
          until (outdeglist[node] = 0);
        end;
      end;
    end; {else}
  end; {6}

7 : begin
```



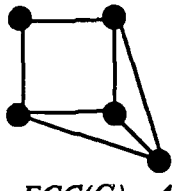
ECC(G) = 2

```
  findnode (outdeglist, node, degone, matched);
  if (matched = true)
   then begin
     tmpnum[1] := node;
     findpair (outdeglist, tmpnum, matrix, node);
     twovert (outclique, maxclique, tmpnum, maxcliquefile);
     filltmpnum (outdeglist, tmpnum);
     numedges := 6;
     fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
   end
  else begin
   node := 0;
   matched := false;
   findnode (outdeglist, node, degtwo, matched);
   tmpnum[1] := node;
   matched := false;
   findnode (outdeglist, node, degtwo, matched);
   if (matched = true)
```

```
then begin
```

ECC(G) = 3

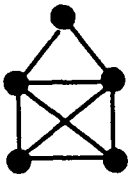
```
  temp := node;
  node := tmpnum[1];
  findadjnode (node, matrix, tmpnum);
  findatri (tmpnum, matrix, outclique, maxclique, success,
        maxcliquefile);
  for i := 2 to 3 do
   matrix[tmpnum[1],tmpnum[i]] := 0;
  outdeglist[tmpnum[1]] := 0;
  for i := 2 to 3 do
   outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
  filltmpnum (outdeglist, tmpnum);
  numedges := 5;
  fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
 end
else begin
```



ECC(G) = 4

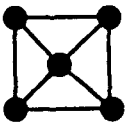
```
  node := tmpnum[1];
  repeat
   findpair (outdeglist, tmpnum, matrix, node);
   twovert (outclique, maxclique, tmpnum, maxcliquefile);
   matrix[tmpnum[1],tmpnum[2]] := 0;
  until (outdeglist[tmpnum[1]] = 0);
  filltmpnum (outdeglist, tmpnum);
  numedges := 5;
  fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
 end;
end; {else}
```

```
  end; {7}

8 : begin
```



ECC(G) = 2

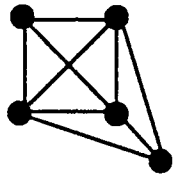
```
    findnode (outdeglist, node, degfour, matched);
    temp := node;
    matched := false;
    findnode (outdeglist, node, degfour, matched);
    if (matched = true)
     then begin
      tmpnum[1] := temp;
      tmpnum[2] := node;
      node := 0;
      matched := false;
      findnode (outdeglist, node, degtwo, matched);
      tmpnum[3] := node;
      numedges := 3;
      threevert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
      outdeglist[tmpnum[3]] := 0;
      for i := 1 to 2 do
       outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
      filltmpnum (outdeglist, tmpnum);
      numedges := 6;
      fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
     end
    else begin
```



ECC(G) = 4

```
     counter := degfour;
     node := temp;
     findadjnode (node, matrix, tmpnum);
     repeat
```

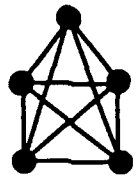
```
        findatri (tmpnum, matrix, outclique, maxclique, success,
              maxcliquefile);
        if (success = true)
         then begin
          matrix[tmpnum[2], tmpnum[3]] := 0;
          matrix[tmpnum[3], tmpnum[2]] := 0;
          if (counter < degfour)
           then begin
            matrix[tmpnum[1], tmpnum[2]] := 0;
            tmpnum[2] := tmpnum[3];
           end;
          counter := counter - 1;
         end;
       until (counter = 0);
      end;
     end; {8}

9 : begin
```



ECC(G) = 2

```
    findnode (outdeglist, node, degfour, matched);
    tmpnum[1] := node;
    matched := false;
    findnode (outdeglist, node, degfour, matched);
    tmpnum[2] := node;
    matched := false;
    findnode (outdeglist, node, degfour, matched);
    tmpnum[3] := node;
    node := 0;
    matched := false;
    findnode (outdeglist, node, degthree, matched);
    tmpnum[4] := node;
    numedges := 6;
    fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
    outdeglist[tmpnum[4]] := 0;
    for i := 1 to 3 do
     outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
```

```
      filltmpnum (outdeglist, tmpnum);
      fourvert (outclique, maxclique, tmpnum, numedges, maxcliquefile);
     end; {9}

  10 : begin
```

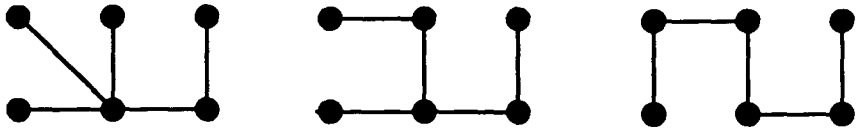


ECC(G) = 1

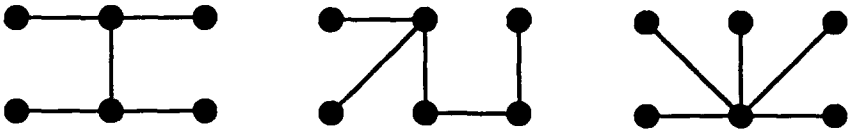
```
      for i := 1 to curvertices do
       maxclique[i] := vertexname[tmpnum[i]];
      writeclique (maxclique, curvertices, outclique, maxcliquefile);
      outclique := outclique + 1;
     end; {10}
  end; {case}
end;    { fivevert }
```

```
{*************************************************************

 PROCEDURE : sixvert

 CALLED BY : findclique

 CALLS     : oneedgelessthan
             findnode
             findtriwithlegs
             circlekaytwo
             findpair
             twovert
             filltmpnum
             fourvert
             findadjnode
             threevert
             findatri
             fivevert
             writeclique

 PURPOSE   : Finds all maximal cliques in graphs of six vertices.
             Uses number of edges and degrees as discriminators.

*************************************************************}

procedure sixvert ( var outclique: integer; var maxclique: nametype;
                    tmpnum: column; numedges: integer; matrix:
                    matrixtype; var maxcliquefile: text );
var
 matched,                    { Returns true if node is found          }
 success,                    { Returns true if triangle found         }
 adjacent : boolean;         { Returns true if two vertices are adj   }
 i, j, counter,              { Incremental counters                   }
 node,                       { Node found in findnode search          }
 temp, temp1, temp2,         { Temporary storage for nodes            }
 curvertices : integer;      { Current vertice number of graph        }

begin  { sixvert }
 curvertices := 6;
 node := 0;
 matched := false;
```

case numedges of



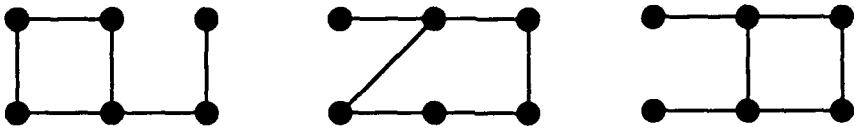
ECC(G) = 5



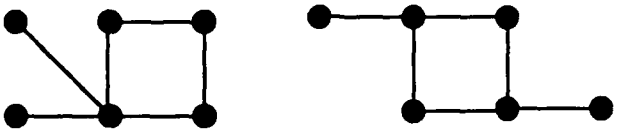
ECC(G) = 5

5 : oneedgelessthan (outclique, maxclique, tmpnum, outdeglist, matrix, maxcliquefile);

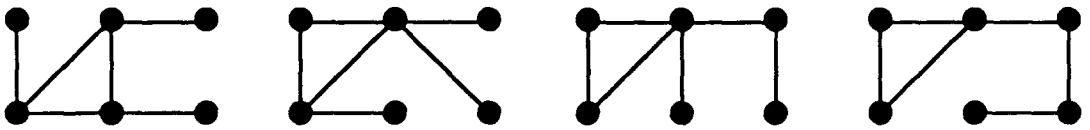
6 : begin



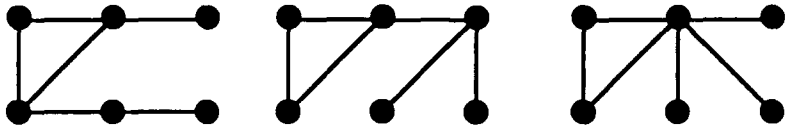
ECC(G) = 6



ECC(G) = 6

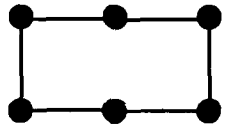


ECC(G) = 4



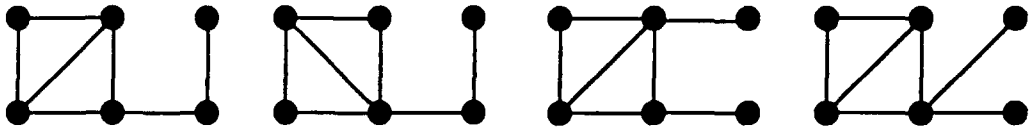
ECC(G) = 4 *ECC(G) = 4* *ECC(G) = 4*

```
findnode (outdeglist, node, degone, matched);
 if (matched = true)
  then begin
    tmpnum[1] := node;
    findpair (outdeglist, tmpnum, matrix, node);
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    matrix[tmpnum[2],tmpnum[1]] := 0;
    numedges := 5;
    filltmpnum (outdeglist, tmpnum);
    fivevert (outclique, maxclique, tmpnum, numedges, matrix,
          maxcliquefile);
  end
```

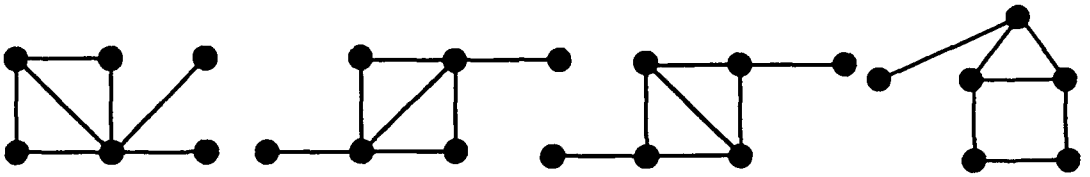


ECC(G) = 6

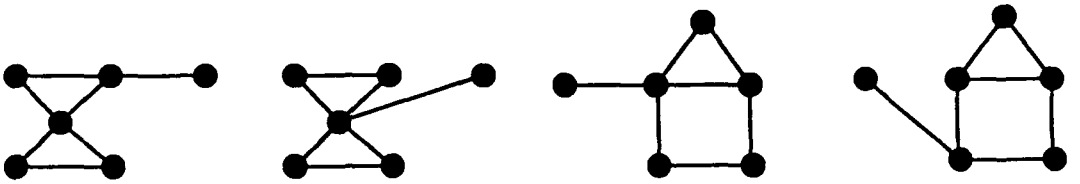
```
  else circlekaytwo (outdeglist, tmpnum, matrix, outclique,
            maxcliquefile);
end; {6}

7 : begin
```



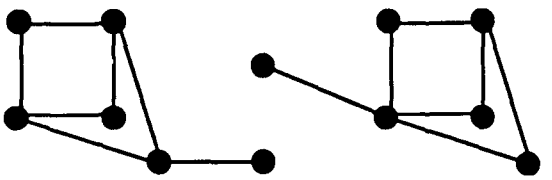
ECC(G) = 4



ECC(G) = 4 *ECC(G) = 4* *ECC(G) = 4* *ECC(G) = 5*



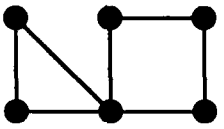
ECC(G) = 3 *ECC(G) = 5*



ECC(G) = 7

```
findnode (outdeglist, node, degone, matched);
  if (matched = true)
    then begin
      tmpnum[1] := node;
      findpair (outdeglist, tmpnum, matrix, node);
      twovert (outclique, maxclique, tmpnum, maxcliquefile);
      matrix[tmpnum[2], tmpnum[1]] := 0;
      matched := false;
      findnode (outdeglist, node, degone, matched);
      if (matched = true)
        then begin
          tmpnum[1] := node;
          findpair (outdeglist, tmpnum, matrix, node);
          twovert (outclique, maxclique, tmpnum, maxcliquefile);
          matrix[tmpnum[2], tmpnum[1]] := 0;
          filltmpnum (outdeglist, tmpnum);
```

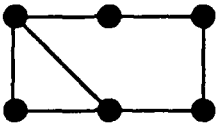
```
      numedges := 5;
      fourvert (outclique, maxclique, tmpnum, numedges,
            maxcliquefile);
     end
    else begin
     numedges := 6;
     fivevert (outclique, maxclique, tmpnum, numedges, matrix,
           maxcliquefile);
    end;
   end
  else begin
```



ECC(G) = 5

```
  node := 0;
  matched := false;
  findnode (outdeglist, node, degfour, matched);
  if (matched = true)
   then begin
    findadjnode (node, matrix, tmpnum);
    findatri (tmpnum, matrix, outclique, maxclique,
          success, maxcliquefile);
    if (success = true)
     then begin
      for i := 1 to 3 do
       outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 2;
      filltmpnum (outdeglist, tmpnum);
      numedges := 4;
      fourvert (outclique, maxclique, tmpnum, numedges,
            maxcliquefile);
     end
    else begin
     twovert (outclique, maxclique, tmpnum, maxcliquefile);
     for i := 1 to 2 do
      outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
     filltmpnum (outdeglist, tmpnum);
     findnode (outdeglist, node, degone, matched);
     numedges := 6;
     findtriwithlegs ( outdeglist, tmpnum, node,
               curvertices, numedges, maxclique,
```

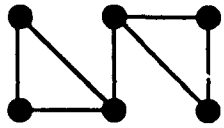
```
                matrix, maxcliquefile);
    end;
  end
else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  tmpnum[1] := node;
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  tmpnum[2] := node;
  if (matrix[tmpnum[1],tmpnum[2]] = 1)
    then begin
      findatri (tmpnum, matrix, outclique, maxclique,
            success, maxcliquefile);
      if (success = true)
        then begin
```



ECC(G) = 5

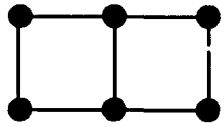
```
          for i := 1 to 3 do
            outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 2;
          for i := 1 to 2 do
            for j := 1 to 3 do
              matrix[tmpnum[i], tmpnum[j]] := 0;
          filltmpnum (outdeglist, tmpnum);
          oneedgelessthan (outclique, maxclique, tmpnum,
                    outdeglist, matrix, maxcliquefile);
        end
      else begin
        twovert (outclique, maxclique, tmpnum, maxcliquefile);
        matrix[tmpnum[1],tmpnum[2]] := 0;
        matrix[tmpnum[2],tmpnum[1]] := 0;
        for i := 1 to 2 do
          outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
        temp := tmpnum[2];
        node := tmpnum[1];
        findadjnode (node, matrix, tmpnum);
        findatri (tmpnum, matrix, outclique, maxclique,
              success, maxcliquefile);
```

```
if (success = true)
 then begin
```



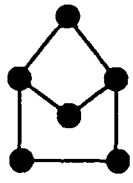
ECC(G) = 3

```
   node := temp;
   findadjnode (node, matrix, tmpnum);
   findatri (tmpnum, matrix, outclique, maxclique,
         success, maxcliquefile);
 end
```



ECC(G) = 7

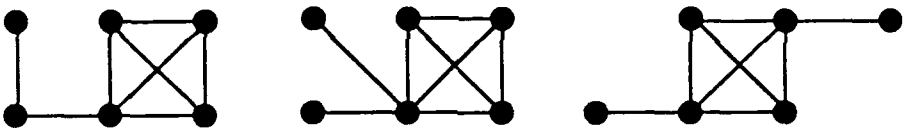
```
  else circlekaytwo (outdeglist, tmpnum, matrix,
              outclique, maxcliquefile);
 end;
end
else begin
```



ECC(G) = 7

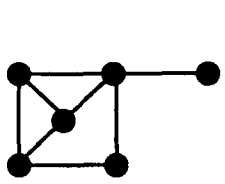
```
 temp := tmpnum[2];
 node := tmpnum[1];
 repeat
  repeat
   findpair (outdeglist, tmpnum, matrix, node);
   twovert (outclique, maxclique, tmpnum, maxcliquefile);
   matrix[tmpnum[1],tmpnum[2]] := 0;
   matrix[tmpnum[2],tmpnum[1]] := 0;
  until (outdeglist[node] = 0);
```

```
          node := temp;
          tmpnum[1] := node;
        until (outdeglist[node] = 0);
        filltmpnum (outdeglist, tmpnum);
        twovert (outclique, maxclique, tmpnum, maxcliquefile);
       end;
     end;
    end; {else}
   end; {7}

8 : begin
    findnode (outdeglist, node, degone, matched);
    if (matched = true)
     then begin
      tmpnum[1] := node;
      findpair (outdeglist, tmpnum, matrix, node);
      twovert (outclique, maxclique, tmpnum, maxcliquefile);
      matrix[tmpnum[2],tmpnum[1]] := 0;
      matched := false;
      findnode (outdeglist, node, degone, matched);
      if (matched = true)
       then begin
```

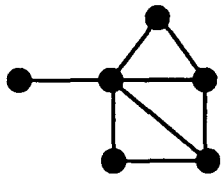


ECC(G) = 3

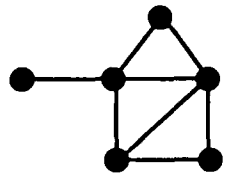
```
        tmpnum[1] := node;
        findpair (outdeglist, tmpnum, matrix, node);
        twovert (outclique, maxclique, tmpnum, maxcliquefile);
        filltmpnum (outdeglist, tmpnum);
        numedges := 6;
        fourvert (outclique, maxclique, tmpnum, numedges,
              maxcliquefile);
       end
      else begin
```



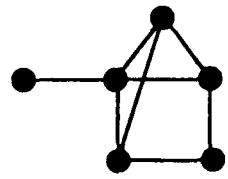
ECC(G) = 5



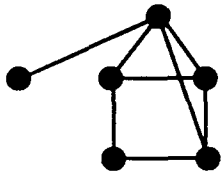
ECC(G) = 4



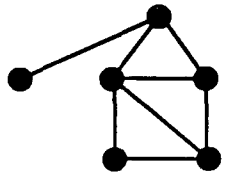
ECC(G) = 4



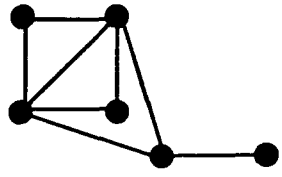
ECC(G) = 5

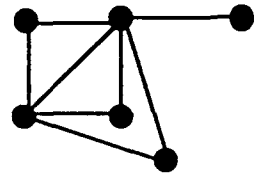


ECC(G) = 5

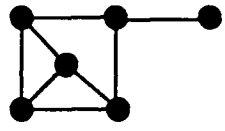


ECC(G) = 4





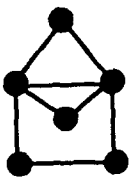
ECC(G) = 4



ECC(G) = 5

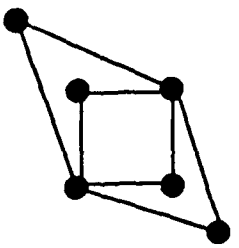
```
    numedges := 7;
    fivevert (outclique, maxclique, tmpnum, numedges, matrix,
          maxcliquefile);
   end;
  end
 else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degfour, matched);
  if (matched = true)
   then begin
    tmpnum[1] := node;
    matched := false;
    findnode (outdeglist, node, degfour, matched);
    if (matched = true)
```

```
then begin
```



ECC(G) = 5

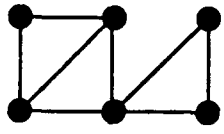
```
 tmpnum[2] := node;
 if (matrix[tmpnum[1], tmpnum[2]] = 1)
  then begin
   findatri (tmpnum, matrix, outclique, maxclique,
          success, maxcliquefile);

   matrix[tmpnum[1], tmpnum[3]] := 0;
   matrix[tmpnum[2], tmpnum[3]] := 0;
   for i := 1 to 2 do
    outdeglist[tmpnum[i]] :=
           outdeglist[tmpnum[i]] - 1;
   outdeglist[tmpnum[3]] := 0;
   numedges := 6;
   fivevert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
  end
 else begin
```



ECC(G) = 8

```
   node := tmpnum[1];
   findpair (outdeglist, tmpnum, matrix, node);
   twovert (outclique, maxclique, tmpnum,
         maxcliquefile);
   matrix[tmpnum[1], tmpnum[2]] := 0;
   matrix[tmpnum[2], tmpnum[1]] := 0;
   numedges := 7;
```

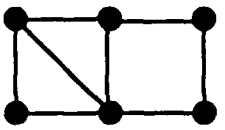
```
      sixvert (outclique, maxclique, tmpnum, numedges,
            matrix, maxcliquefile);
    end;
  end
else begin
  temp := tmpnum[1];
  node := 0;
  matched := false;
  findnode (outdeglist, node, degtwo, matched);
  tmpnum[1] := node;
  matched := false;
  findnode (outdeglist, node, degtwo, matched);
  tmpnum[2] := node;
  matched := false;
  findnode (outdeglist, node, degtwo, matched);
  tmpnum[3] := node;
  adjacent := false;
  for i := 1 to 2 do
   for j := 2 to 3 do
     begin
      if (matrix[tmpnum[i], tmpnum[j]] = 1)
        then begin
          adjacent := true;
          temp1 := tmpnum[i];
          temp2 := tmpnum[j];
        end;
     end;
 if (adjacent = true)
   then begin
```



ECC(G) = 3

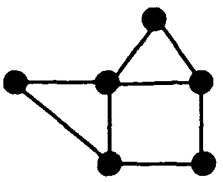
```
    tmpnum[1] := temp1;
    tmpnum[2] := temp2;
    findatri (tmpnum, matrix, outclique, maxclique,
          success, maxcliquefile);
    if (success = true)
     then begin
       for i := 1 to 3 do
```

```
      begin
       outdeglist[tmpnum[i]] :=
             outdeglist[tmpnum[i]] - 2;
       matrix[tmpnum[3] , tmpnum[i]] := 0;
      end;
     numedges := 5;
     fourvert (outclique, maxclique, tmpnum,
          numedges, maxcliquefile);
   end
  else begin
```



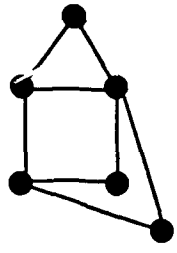
ECC(G) = 5

```
    twovert (outclique, maxclique, tmpnum,
         maxcliquefile);
    for i := 1 to 2 do
     outdeglist[tmpnum[i]] :=
           outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    numedges := 7;
    sixvert (outclique, maxclique, tmpnum, numedges,
         matrix, maxcliquefile);
   end;
  end
else begin
  node := temp;
  findadjnode (node, matrix, tmpnum);
  findatri (tmpnum, matrix, outclique, maxclique,
       success, maxcliquefile);
  if (success = true)
   then begin
```



ECC(G) = 4

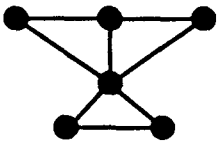
```
      for i := 1 to 3 do
       begin
        outdeglist[tmpnum[i]] :=
              outdeglist[tmpnum[i]] - 2;
        for j := 1 to 3 do
         matrix[tmpnum[i], tmpnum[j]] := 0;
       end;
      filltmpnum (outdeglist, tmpnum);
      numedges := 5;
      fivevert (outclique, maxclique, tmpnum, numedges,
            matrix, maxcliquefile);
     end
    else begin
```



ECC(G) = 6

```
      twovert (outclique, maxclique, tmpnum, maxcliquefile);
      for i := 1 to 2 do
       outdeglist[tmpnum[i]] :=
             outdeglist[tmp    n[i]] - 1;
      matrix[tmpnum[1], tmpnum[2]] := 0;
      matrix[tmpnum[2], tmpnum[1]] := 0;
      numedges := 7;
      sixvert (outclique, maxclique, tmpnum, numedges,
            matrix, maxcliquefile);
     end;
    end;
   end;
  end
 else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degfive, matched);
  if (matched = true)
```

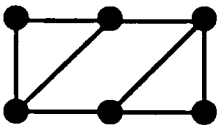
```
then begin
```



ECC(G) = 3

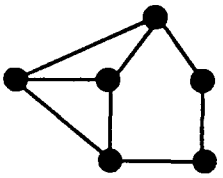
```
  tmpnum[1] := node;
  node := 0;
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  tmpnum[2] := node;
  findatri (tmpnum, matrix, outclique, maxclique, success,
        maxcliquefile);
  outdeglist[tmpnum[3]] := 0;
  for i := 1 to 2 do
   outdeglist[tmpnum[i]] :=
          outdeglist[tmpnum[i]] - 1;
  matrix[tmpnum[1],tmpnum[3]] := 0;
  matrix[tmpnum[2],tmpnum[3]] := 0;
  findatri (tmpnum, matrix, outclique, maxclique, success,
        maxcliquefile);
  for i := 2 to 3 do
   outdeglist[tmpnum[i]] := 0;
  filltmpnum (outdeglist, tmpnum);
  numedges := 3;
  threevert (outclique, maxclique, tmpnum, numedges,
         maxcliquefile);
 end
else begin
 node := 0;
 matched := false;
 findnode (outdeglist, node, degtwo, matched);
 findadjnode (node, matrix, tmpnum);
 findatri (tmpnum, matrix, outclique, maxclique,
       success, maxcliquefile);
 if (success = true)
```

```
then begin
```

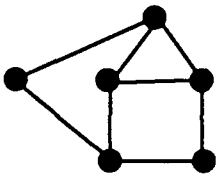


ECC(G) = 4

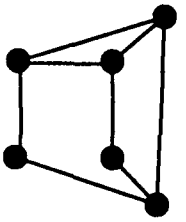
```
 for i := 1 to 3 do
  begin
   outdeglist[tmpnum[i]] :=
          outdeglist[tmpnum[i]] - 2;
   for j := 1 to 3 do
    matrix[tmpnum[i], tmpnum[j]] := 0;
  end;
 numedges := 5;
 fivevert (outclique, maxclique, tmpnum, numedges,
       matrix, maxcliquefile);
end
else begin
```



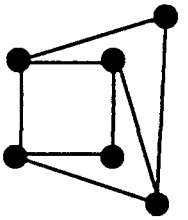
ECC(G) = 5



ECC(G) = 6



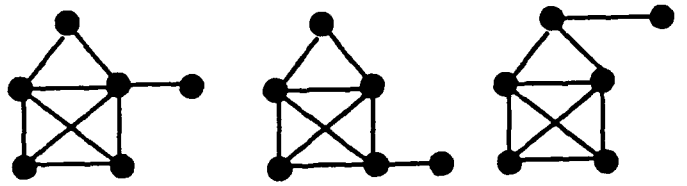
ECC(G) = 6



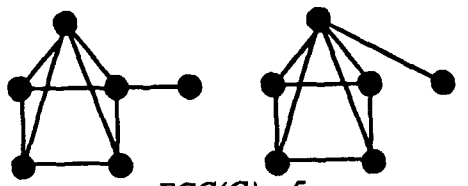
ECC(G) = 8

```
    findpair (outdeglist, tmpnum, matrix, node);
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    numedges := 7;
    sixvert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end;
  end;
 end;
end; {else}
end; {8}

9 : begin
```

```
findnode (outdeglist, node, degone, matched);
```

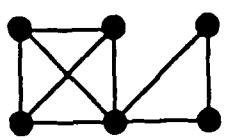


ECC(G) = 3



ECC(G) = 5

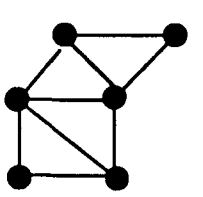
```
if (matched = true)
 then begin
  tmpnum[1] := node;
  findpair (outdeglist, tmpnum, matrix, node);
  twovert (outclique, maxclique, tmpnum, maxcliquefile);
  matrix[tmpnum[2], tmpnum[1]] := 0;
  numedges := 8;
  fivevert (outclique, maxclique, tmpnum, numedges,
        matrix, maxcliquefile);
 end
else begin
 node := 0;
 matched := false;
 findnode (outdeglist, node, degtwo, matched);
 if (matched = true)
  then begin
   findadjnode (node, matrix, tmpnum);
   findatri ( tmpnum, matrix, outclique, maxclique,
         success, maxcliquefile);
   if (success = true)
    then begin
```



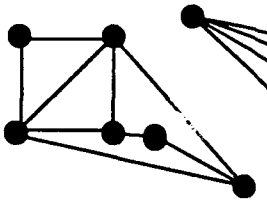
ECC(G) = 2

```
     if (outdeglist[tmpnum[2]] = degtwo) or
```

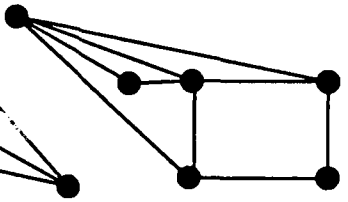
```
        (outdeglist[tmpnum[3]] = degtwo)
 then begin
  outdeglist[tmpnum[1]] := 0;
  if (outdeglist[tmpnum[2]] = degtwo)
    then begin
      outdeglist[tmpnum[2]] := 0;
      matrix[tmpnum[3], tmpnum[2]] := 0;
      matrix[tmpnum[3], tmpnum[1]] := 0;
      outdeglist[tmpnum[3]] :=
            outdeglist[tmpnum[3]] - 2;
    end
    else begin
      outdeglist[tmpnum[3]] := 0;
      matrix[tmpnum[2], tmpnum[3]] := 0;
      matrix[tmpnum[2], tmpnum[1]] := 0;
      outdeglist[tmpnum[2]] :=
            outdeglist[tmpnum[2]] - 2;
    end;
    filltmpnum (outdeglist, tmpnum);
    numedges := 6;
    fourvert (outclique, maxclique, tmpnum,
          numedges, maxcliquefile);
  end
 else begin
```



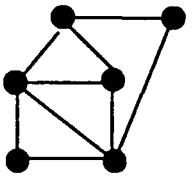
ECC(G) = 4



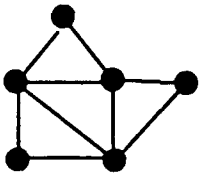
ECC(G) = 5



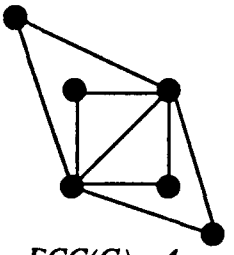
ECC(G) = 5



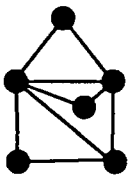
ECC(G) = 5



ECC(G) = 4



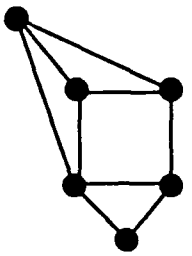
ECC(G) = 4



ECC(G) = 4

```
 matched := false;
 findnode (outdeglist, node, degtwo, matched);
```

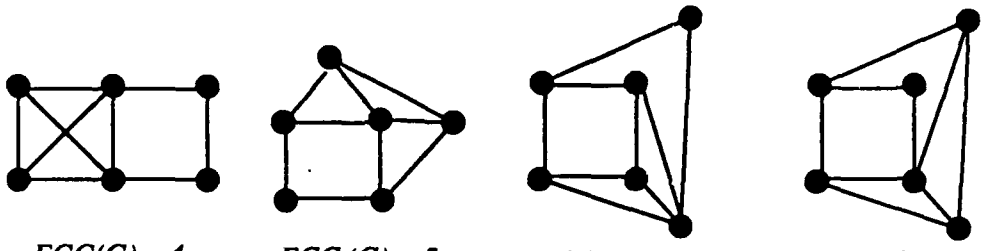
```
  if (matched = true)
   then begin
    outdeglist[tmpnum[1]] := 0;
    for i := 2 to 3 do
     outdeglist[tmpnum[i]] :=
           outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    matrix[tmpnum[3], tmpnum[1]] := 0;
    numedges := 7;
    fivevert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end
  else begin
```



ECC(G) = 4

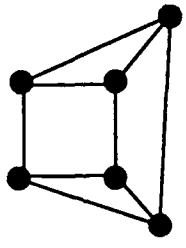
```
    for i := 1 to 3 do
     begin
      outdeglist[tmpnum[i]] :=
            outdeglist[tmpnum[i]] - 2;
      for j := 1 to 3 do
       matrix[tmpnum[i],tmpnum[j]] := 0;
     end;
    numedges := 6;
    fivevert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end;
  end;
end
```

```
else begin
```



ECC(G) = 4 *ECC (G) = 5* *ECC(G) = 6* *ECC(G) = 6*

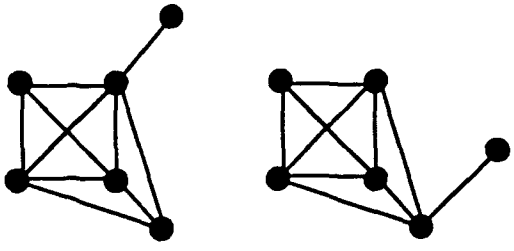
```
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    for i := 1 to 2 do
     outdeglist[tmpnum[i]] :=
            outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    numedges := 8;
    sixvert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
  end;
 end
else begin
```



ECC(G) = 5

```
 node := 1;
 findadjnode (node, matrix, tmpnum);
 findatri (tmpnum, matrix, outclique, maxclique, success,
       maxcliquefile);
 if (success = true)
  then begin
   for i := 1 to 3 do
    begin
     outdeglist[tmpnum[i]] :=
            outdeglist[tmpnum[i]] - 2;
     for j := 1 to 3 do
```

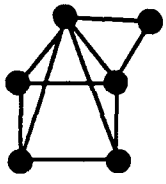
```
            matrix[tmpnum[i], tmpnum[j]] := 0;
          end;
         numedges := 6;
         sixvert (outclique, maxclique, tmpnum, numedges,
              matrix, maxcliquefile);
       end
      else begin
       twovert (outclique, maxclique, tmpnum, maxcliquefile);
       for i := 1 to 2 do
        outdeglist[tmpnum[i]] :=
              outdeglist[tmpnum[i]] - 1;
       matrix[tmpnum[1], tmpnum[2]] := 0;
       matrix[tmpnum[2], tmpnum[1]] := 0;
       numedges := 8;
       sixvert (outclique, maxclique, tmpnum, numedges,
            matrix, maxcliquefile);
      end;
     end;
    end;
  end; {9}

10 : begin
```



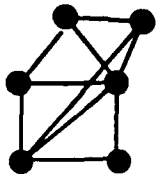
ECC(G) = 3

```
    findnode (outdeglist, node, degone, matched);
    if (matched = true)
     then begin
      tmpnum[1] := node;
      findpair ( outdeglist, tmpnum, matrix, node);
      twovert (outclique, maxclique, tmpnum, maxcliquefile);
      matrix[tmpnum[2], tmpnum[1]] := 0;
      numedges := 9;
      fivevert (outclique, maxclique, tmpnum, numedges,
           matrix, maxcliquefile);
     end
    else begin
```

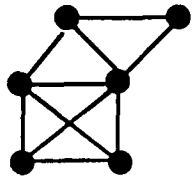
```
node := 0;
matched := false;
findnode (outdeglist, node, degtwo, matched);
if (matched = true)
 then begin
  findadjnode (node, matrix, tmpnum);
  findatri (tmpnum, matrix, outclique, maxclique,
        success, maxcliquefile);
  if (success = true)
   then begin
```



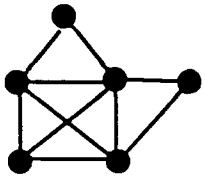
ECC(G) = 5



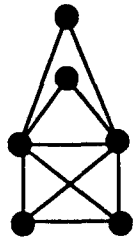
ECC(G) = 5



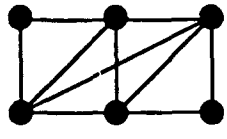
ECC(G) = 3



ECC(G) = 3



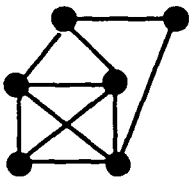
ECC(G) = 3



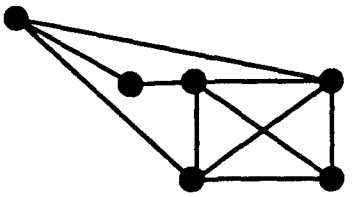
ECC(G) = 3

```
    outdeglist[tmpnum[1]] := 0;
    for i := 2 to 3 do
     outdeglist[tmpnum[i]] :=
           outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    matrix[tmpnum[3], tmpnum[1]] := 0;
    numedges := 8;
    fivevert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end
```

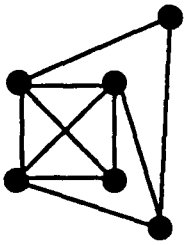
else begin



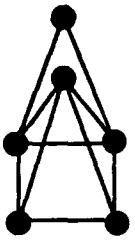
ECC(G) = 4



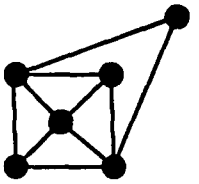
ECC(G) = 4



ECC(G) = 4



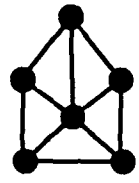
ECC(G) = 5



ECC(G) = 6

```
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    for i := 1 to 2 do
      outdeglist[tmpnum[i]] :=
            outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    numedges := 9;
    sixvert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end; {success}
  end
 else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degfive, matched);
  if (matched = true)
```

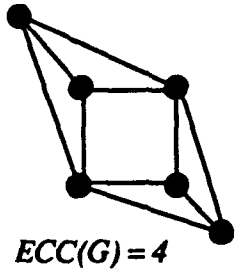
```
then begin
```



ECC(G) = 5

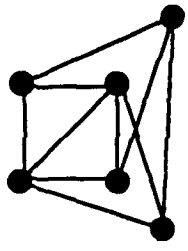
```
  counter := degfive;
  findadjnode (node, matrix, tmpnum);
  repeat
   findatri (tmpnum, matrix, outclique, maxclique,
         success, maxcliquefile);
   if (success = true)
    then begin
     matrix[tmpnum[2], tmpnum[3]] := 0;
     matrix[tmpnum[3], tmpnum[2]] := 0;
     if (counter < degfive)
      then begin
       matrix[tmpnum[1], tmpnum[2]] := 0;
       tmpnum[2] := tmpnum[3];
      end;
     counter := counter - 1;
    end;
  until (counter = 0);
 end
else begin
 node := 0;
 matched := false;
 findnode (outdeglist, node, degthree, matched);
 tmpnum[1] := node;
 adjacent := false;
 repeat
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  if (matched = true)
   then begin
    if (matrix[tmpnum[1], node] = 1)
     then begin
      adjacent := true;
      tmpnum[2] := node;
     end;
   end;
```

```
until (adjacent = true);
findatri (tmpnum, matrix, outclique, maxclique,
      success, maxcliquefile);
if (success = true)
 then begin
```



ECC(G) = 4

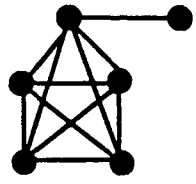
```
  outdeglist[tmpnum[3]] := outdeglist[tmpnum[3]] - 2;
  for i := 1 to 2 do
   outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
  matrix[tmpnum[3], tmpnum[1]] := 0;
  matrix[tmpnum[3], tmpnum[2]] := 0;
  matrix[tmpnum[1], tmpnum[3]] := 0;
  matrix[tmpnum[2], tmpnum[3]] := 0;
  numedges := 8;
  sixvert (outclique, maxclique, tmpnum, numedges,
        matrix, maxcliquefile);
 end
else begin
```



ECC(G) = 6

```
 twovert (outclique, maxclique, tmpnum, maxcliquefile);
 matrix[tmpnum[1], tmpnum[2]] := 0;
 matrix[tmpnum[2], tmpnum[1]] := 0;
 for i := 1 to 2 do
  outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
 numedges := 9;
 sixvert (outclique, maxclique, tmpnum, numedges,
       matrix, maxcliquefile);
```

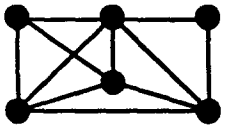
```
        end;
       end;
      end;
     end;
    end; {10}

11 : begin
```



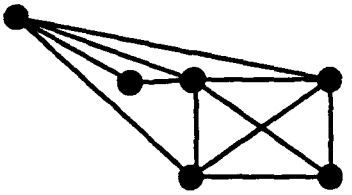
ECC(G) = 2

```
  findnode (outdeglist, node, degone, matched);
  if (matched = true)
   then begin
    tmpnum[1] := node;
    findpair (outdeglist, tmpnum, matrix, node);
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    matrix[tmpnum[2], tmpnum[1]] := 0;
    filltmpnum (outdeglist, tmpnum);
    numedges := 10;
    fivevert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end
  else begin
   node := 0;
   matched := false;
   findnode (outdeglist, node, degtwo, matched);
   if (matched = true)
    then begin
     findadjnode (node, matrix, tmpnum);
     findatri (tmpnum, matrix, outclique, maxclique,
           success, maxcliquefile);
     if (success = true)
```

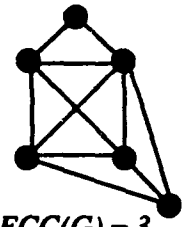
then begin



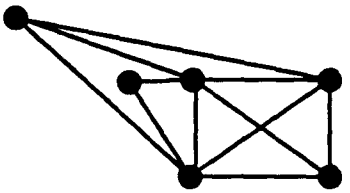
ECC(G) = 3



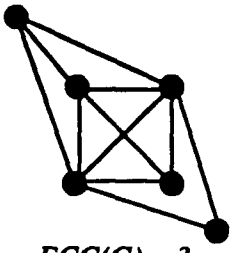
ECC(G) = 3



ECC(G) = 3

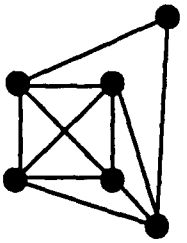


ECC(G) = 3



ECC(G) = 3

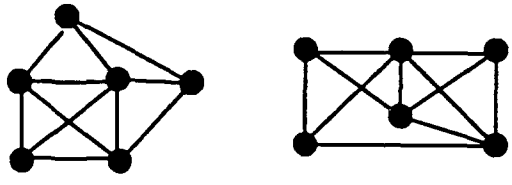
```
 outdeglist[tmpnum[1]] := 0;
 for i := 2 to 3 do
  outdeglist[tmpnum[i]] :=
          outdeglist[tmpnum[i]] - 1;
 matrix[tmpnum[2], tmpnum[1]] := 0;
 matrix[tmpnum[3], tmpnum[1]] := 0;
 numedges := 9;
 fivevert (outclique, maxclique, tmpnum, numedges,
        matrix, maxcliquefile);
end
else begin
```



ECC(G) = 4

```
twovert (outclique, maxclique, tmpnum, maxcliquefile);
for i := 1 to 2 do
 outdeglist[tmpnum[i]] :=
```

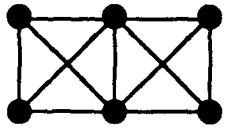
```
            outdeglist[tmpnum[i]] - 1;
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    numedges := 10;
    sixvert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end; {success}
  end
else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degfive, matched);
  if (matched = true)
   then begin
    tmpnum[1] := node;
    node := 0;
    matched := false;
    findnode (outdeglist, node, degfour, matched);
    if (matched = true)
     then begin
```



ECC(G) = 4

```
      tmpnum[2] := node;
      matched := false;
      findnode (outdeglist, node, degfour, matched);
      tmpnum[3] := node;
      counter := 1;
      adjacent := false;
      repeat
       if (matrix[tmpnum[1], counter] = 1) and
         (matrix[tmpnum[2], counter] = 1) and
         (matrix[tmpnum[3], counter] = 1)
        then begin
         adjacent := true;
         tmpnum[4] := counter;
         numedges := 6;
         fourvert (outclique, maxclique, tmpnum,
```

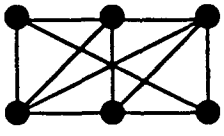
```
            numedges, maxcliquefile);
        outdeglist[tmpnum[4]] := 0;
        outdeglist[tmpnum[1]] :=
              outdeglist[tmpnum[1]] - 1;
        for j := 2 to 3 do
         outdeglist[tmpnum[j]] :=
               outdeglist[tmpnum[j]] - 2;
        for i := 2 to 3 do
         for j := 2 to 4 do
          matrix[tmpnum[i],tmpnum[j]] := 0;
        matrix[tmpnum[1],tmpnum[4]] := 0;
        numedges := 7;
        fivevert (outclique, maxclique, tmpnum,
              numedges, matrix, maxcliquefile);
      end;
    counter := counter + 1;
  until (adjacent = true);
 end
else begin
```



ECC(G) = 2

```
 node := tmpnum[1];
 matched := false;
 findnode (outdeglist, node, degfive, matched);
 tmpnum[2] := node;
 node := 0;
 matched := false;
 findnode (outdeglist, node, degthree, matched);
 tmpnum[3] := node;
 counter := 1;
 adjacent := false;
 repeat
  if (matrix[tmpnum[1], counter] = 1) and
    (matrix[tmpnum[2], counter] = 1) and
    (matrix[tmpnum[3], counter] = 1)
   then begin
    adjacent := true;
    tmpnum[4] := counter;
```

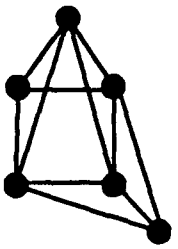
```
        numedges := 6;
        fourvert (outclique, maxclique, tmpnum,
              numedges, maxcliquefile);
        for i := 1 to 2 do
          outdeglist[tmpnum[i]] :=
                outdeglist[tmpnum[i]] - 2;
        for i := 3 to 4 do
          outdeglist[tmpnum[i]] := 0;
       end;
      counter := counter + 1;
    until (adjacent = true);
    filltmpnum (outdeglist, tmpnum);
    numedges := 6;
    fourvert (outclique, maxclique, tmpnum, numedges,
          maxcliquefile);
   end;
  end
else begin
  node := 0;
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  tmpnum[1] := node;
  matched := false;
  findnode (outdeglist, node, degthree, matched);
  tmpnum[2] := node;
  if (matrix[tmpnum[1], tmpnum[2]] = 1)
   then begin
```



ECC(G) = 4

```
    twovert (outclique, maxclique, tmpnum, maxcliquefile);
    matrix[tmpnum[1], tmpnum[2]] := 0;
    matrix[tmpnum[2], tmpnum[1]] := 0;
    for i := 1 to 2 do
     outdeglist[tmpnum[i]] := outdeglist[tmpnum[i]] - 1;
    numedges := 10;
    sixvert (outclique, maxclique, tmpnum, numedges,
          matrix, maxcliquefile);
   end
```

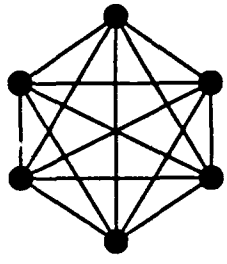
```
else begin
```



ECC(G) = 6

```
temp := tmpnum[1];
findadjnode (node, matrix, tmpnum);
findatri (tmpnum, matrix, outclique, maxclique,
      success, maxcliquefile);
if (matrix[tmpnum[2],tmpnum[temp]] = 1)
 then begin
  for i := 1 to 2 do
   outdeglist[tmpnum[i]] :=
         outdeglist[tmpnum[i]] - 1;
  matrix[tmpnum[1], tmpnum[2]] := 0;
  matrix[tmpnum[2], tmpnum[1]] := 0;
 end
else begin
 outdeglist[tmpnum[1]] :=
       outdeglist[tmpnum[1]] - 1;
 outdeglist[tmpnum[3]] :=
       outdeglist[tmpnum[3]] - 1;
  matrix[tmpnum[1], tmpnum[3]] := 0;
  matrix[tmpnum[3], tmpnum[1]] := 0;
end;
numedges := 10;
sixvert (outclique, maxclique, tmpnum, numedges,
     matrix, maxcliquefile);
 end;
 end;
 end;
 end;
end; {11}

12 : begin
   writeln (cliquefile);
   write (cliquefile, '  This program cannot detect the maximal ');
   writeln (cliquefile, 'clique covering for a net ');
```

```
        writeln (cliquefile, ' of six stations with twelve edges.');
      end; {12}

  13 : begin
        writeln (cliquefile);
        write (cliquefile, '  This program cannot detect the maximal ');
        writeln (cliquefile, 'clique covering for a net ');
        writeln (cliquefile, ' of six stations with thirteen edges.');
      end; {13}

  14: begin
        writeln (cliquefile);
        write (cliquefile, '  This program cannot detect the maximal ');
        writeln (cliquefile, 'clique covering for a net ');
        writeln (cliquefile, ' of six stations with fourteen edges.');
      end; {14

  15 : begin
```



ECC(G) = 1

```
        for i := 1 to curvertices do
          maxclique[i] := vertexname[tmpnum[i]];
        writeclique (maxclique, curvertices, outclique, maxcliquefile);
        outclique := outclique + 1;
      end; {15}
    end; {case}
  end;    { sixvert }

{*************************************************************************}

begin  { findclique }
  for i := 1 to avertex do                {puts node numbers in an array          }
    tmpnum[i] := i;
  detdegrees (matrix, outdeglist, tmpnum, avertex);
  outclique := 1;                         {row number to hold found cliques       }
```

```
{ The following will initialize the list that will hold the maxcliques                }
{ found in the program by first filling the array with blanks.                        }
  for i := 1 to avertex do
    maxclique[i] := blank;
  case avertex of
    1 : begin
          maxclique[1] := vertexname[tmpnum[1]];
          writeclique (maxclique, avertex, outclique, cliquefile);
        end;

    2 : twovert (outclique, maxclique, tmpnum, cliquefile);

    3 : threevert (outclique, maxclique, tmpnum, numedges, cliquefile);

    4 : fourvert (outclique, maxclique, tmpnum, numedges, cliquefile);

    5 : fivevert (outclique, maxclique, tmpnum, numedges, matrix,
                  cliquefile);

    6 : sixvert (outclique, maxclique, tmpnum, numedges, matrix, cliquefile);
  end;
end;   { findclique }

{*************************************************************************}

begin   { main body clique }
  i := 0;                                 { Initialize the parameter counter          }
                                          { The following for loop reads in the user's }
                                          { Input and output file names               }
  for i := 1 to paramcount do
    nextparam := paramstr(i);
 if (i = 0)
  then writeln ('Must enter an input filename to execute CLIQUE.')
 else begin
                                          { Assigns the output file to a variable name }
                                          { And opens the file for output             }
    assign (cliquefile, paramstr(2));
    rewrite (cliquefile);
                                          { Assigns the input file to a variable name  }
    assign (matrixfile, paramstr(1));
    {$I-}
    reset (matrixfile);                   { Opens the file for reading from           }
```

```
{$I+}
                                    { Checks to make sure the input file exists      }
if (ioresult = 0)
 then begin
  notvalid := false;
  graphnumber := 1;                 { Numbers graphs from input file sequentially   }
  writeln (cliquefile);
  writeln (cliquefile, blank:20, 'The input filename is : ', paramstr(1), '.');
  writeln (cliquefile);
                                    { Loop until all the matrices in the input      }
                                    { File have been calculated or an invalid       }
                                    { Input character exists in the file            }
  while not eof (matrixfile) and (notvalid = false) do
   begin
    vertex := 0;
    readln (matrixfile, totnumvertices);
    readln (matrixfile);
    val (totnumvertices, vertex, notvertex);        { Must be an integer            }
    if (vertex <= matrixsize) and (vertex > 0)      { Vertices must be 1-6          }
     then begin
      if (notvertex = 0) and (vertex <= matrixsize)
       then begin
        for i := 1 to vertex do
         begin
          readln (matrixfile, namestring);          { Read in vertex names          }
          vertexname[i] := namestring;
         end;
       end;
      readln (matrixfile);
      makematrix (vertex, matrix, notvalid, edgenum, cliquefile, matrixfile);
      graphnumber := graphnumber + 1;
      if (notvalid = false)
       then begin
        readln (matrixfile);        { So can start at next matrix                   }
        writeln (cliquefile, 'The maximal cliq es, K, are: ');
        findclique (vertex, edgenum, matrix, cliquefile);
       end
     end
    else begin
     notvalid := true;
     writeln (cliquefile);
     writeln (cliquefile);
     write (cliquefile, 'Must enter a valid integer ');
```

```
          writeln (cliquefile, 'less than or equal to ', matrixsize, '.');
        end;
      end; {while}
     close (matrixfile);
     close (cliquefile);
     end
    else begin
     assign (cliquefile, paramstr(2));
     rewrite (cliquefile);
     writeln (cliquefile, 'Cannot find the input file ', paramstr(1), '.');
     close (cliquefile);
    end;
   end; {else}
end.   { main body clique }
```

```
{************************************************************

  INCLUDE FILE NAME : matrinc.pas

  REFERENCED BY     : clique

  PURPOSE           : To read in the adjacency matrix of a particular
                      graph and output the matrix to a file.

************************************************************}

                { The following are the global structures referenced    }
                { In PROGRAM CLIQUE                                     }

const
  blank = ' ';
  matrixsize = 6;                { Maximum vertices graph may have     }
  namelength = 10;               { Maximum chars for nodal names       }

type
  subscript = 1 .. matrixsize;
  column = array [subscript] of integer;
  matrixtype = array [subscript] of column;   { Data structure holds matrix  }
  nametype = array [subscript] of string;     { Data structure holds names   }

var
  vertexname : nametype;                      { All vertices names for a graph }
  matrix : matrixtype;                        { The adjacency matrix           }
  namestring : string[namelength];            { References one node's name     }
```

```
{*****************************************************************

 FUNCTION   : spaces

 CALLED BY  : leftalign

 PURPOSE    : Adds spaces to the string name given by the user in order
              to square off the output matrix.

 REFERENCE : [Ref. 10:p. 223]

*****************************************************************}
function spaces ( spacestoadd: integer ): string;

var
 i : integer;
 tempspace : string;

begin  { spaces }
 tempspace := blank;
 for i := 1 to spacestoadd do
  tempspace := tempspace + blank;
 spaces := tempspace;
end;   { spaces }
```

```
{**************************************************************

  FUNCTION   : leftalign

  CALLED BY  : makematrix

  PURPOSE    : Left justifies a string. In this program the strings are
               node names from the input file.

  REFERENCE  : [Ref. 10:p. 223]

**************************************************************}
function leftalign ( namestring: string ): string;

var
 spacestoadd : integer;

begin   { leftalign }
 spacestoadd := namelength - length (namestring);
 leftalign := namestring + spaces (spacestoadd);
end;    { leftalign }
```

```
{**************************************************************

  PROCEDURE  : makematrix

  CALLED BY  : clique main program

  CALLS      : leftalign

  PURPOSE    : Reads an input file matrix and outputs the results to a
               file. Done prior to maximal clique calculations, with
               valid input. Checks for isolated vertices and vertex loops.

**************************************************************}
procedure makematrix ( var vertex: integer; var matrix: matrixtype;
                       var notvalid: boolean; var edgectr: integer;
                       var cliquefile: text; var matrixfile: text;
                       var matrixnumber: integer );

var
 isdigit : boolean;                { Ensures entry of "1" or "0" in matrix     }
 inedge : char;                    { Edge read in from input matrix             }
 row,                              { Row of the matrix from input file          }
 column,                           { Column of the matrix from input file       }
 edge,                             { Edge - either a zero or a one              }
 zeroctr,                          { Checks for a row of all zeros - a          }
                                   { A disconnected vertex                      }
 i, j : integer;                   { Incremental counters                       }

begin  { makematrix }
  row := 1;                        { First vertex...                            }
  edgectr := 0;                    { Initialize edge counter for # of vertices  }
  inedge := ' ';                   { Initialize input edge character to blank   }
  edge := 0;                       { Initalize edge input                       }
  while (row <= vertex) and (notvalid = false) do
                   { Loops until every vertice and its corresponding edges     }
                   { Are read in . Checks for illegal characters (not a 0 or1) }
    begin
     zeroctr := 0;                 { Initialize disconnected vertex counter     }
     for column := 1 to vertex do
       begin
         isdigit := false;         { Initialize character check boolean         }
         repeat
```

```
    read (matrixfile, edge);          { Loops while reading blanks spaces      }
  until (inedge <> ' ');
  if (inedge in ['0' .. '1'])         { Valid matrix entry                     }
    then isdigit := true;
  if (isdigit = false)                { NOT a valid entry  - in input file     }
    then begin
      notvalid := true;
      writeln (cliquefile);
      writeln (cliquefile);
      write (cliquefile, 'In net ', matrixnumber);
      writeln (cliquefile, ', to represent an edge : ');
      write (cliquefile, '  Enter a "1" ');
      writeln (cliquefile, 'or a "0" followed by a blank space.');
      writeln (cliquefile);
      write (cliquefile, 'There is an illegal character at ');
      write (cliquefile, 'matrix position row ', row, ', ');
      writeln (cliquefile, 'column ', column, '.');
      writeln (cliquefile);
    end
    else begin
      edge := ord (inedge) - ord ('0');
      matrix[row, column] := edge;          { Reads in an input edge         }
      if (edge = 1)                         { Counts node connections        }
        then zeroctr := zeroctr + 1;
    end;
  end;
for i := row to vertex do
  edgectr := edgectr + matrix[row,i];       { Counts edges in graph          }
if (zeroctr = 0)                            { Node is not connected          }
  then begin
    notvalid := true;
    writeln (cliquefile);
    writeln (cliquefile);
    write (cliquefile, 'In net ', matrixnumber);
    write (cliquefile, ', station ', vertexname[row], ' must be ');
    writeln (cliquefile, 'connected to at least one other station.');
    write (cliquefile, 'Such an edge must be indicated by a ');
    writeln (cliquefile, '"1" in the input matrix.');
  end;
row := row + 1;                             { Goes to next vertice           }
readln (matrixfile);
end;
```

```
  if (notvalid = false)                          { If valid graph input          }
   then begin                                    { Check the diagonal for a      }
    j := 0;                                      { Vertice with an edge to       }
    for i := 1 to vertex do                      { To itself                     }
     begin
      j := j + 1;
      if (matrix[i,j] <> 0)
       then begin
        notvalid := true;
        writeln (cliquefile);
        writeln (cliquefile);
        write (cliquefile, 'In net ', matrixnumber);
        write (cliquefile, ', position ', i, ' for ');
        writeln (cliquefile, vertexname[i], ' is not valid.');
        writeln (cliquefile, 'The entry on a diagonal must be a "0".');
        writeln (cliquefile);
       end;
     end;
   end;

  if (notvalid = false)                  { If input is still valid then write matrix to }
   then begin                            { A file and return to MAIN to calculate the}
    writeln (cliquefile);                { Maximal cliques for the input graph       }
    writeln (cliquefile);
    write (cliquefile, 'The maximal cliques for net ', matrixnumber);
    writeln (cliquefile, ' will be determined for ', vertex:1,' stations.');
    writeln (cliquefile);
    write (cliquefile, blank:11);
    for i := 1 to vertex do
     write (cliquefile, leftalign(vertexname[i]));
    writeln (cliquefile);
    for i := 1 to vertex do
     begin
      write (cliquefile, leftalign(vertexname[i]));
      for j := 1 to vertex do
       write (cliquefile, blank:2, matrix[i, j], blank:8);
      writeln (cliquefile);
     end;
    writeln (cliquefile);
    writeln (cliquefile);
   end;
end;   { makematrix }
```

```
{*************************************************************

  INCLUDE FILE NAME : writinc.pas

  REFERENCED BY     : clique

  PROCEDURE         : writeclique

  CALLED BY         : findclique
                      twovert
                      threevert
                      fourvert
                      fivevert
                      sixvert

  PURPOSE           : To write the found maximum clique to the user's
                      output file. The cliques are determined in the
                      main program.

*************************************************************}
procedure writeclique ( var maxclique: nametype; totsize, numcliques:
                        integer; var maxcliquefile: text );

var
  j : integer;                  { Incremental counter                 }

begin  { writeclique }
   writeln (maxcliquefile);
   write (maxcliquefile, blank:5, numcliques, ' = { ');
   j := 1;
                                 { While there are vertices in the clique }
                                 { Number of vertices in clique = totsize}
   while (maxclique[j] <> blank) and (j <= totsize) do
      begin
        write (maxcliquefile, maxclique[j], blank:2);
        j := j + 1;
      end;
  write (maxcliquefile, '}');
   writeln (maxcliquefile);
   for j := 1 to totsize do
      maxclique[j] := blank;
end;   { writeclique }
```

APPENDIX B

CLIQUE USER'S GUIDE

A. OVERVIEW

CLIQUE is a stand-alone, IBM-compatible, executable program. Given the adjacency matrix representation of possible transmission conflicts (conflicts are characterized by the user) in a communications net, it will produce a listing of maximal (k-order) cliques, representing communications subnets with a common conflict (k represents the number of stations in a complete subnet K of the original net). Any text editor may be used to create an input file of the conflict representation. Using the input and a user's named output file, CLIQUE will then detect and record as cliques the maximum number of stations, per net, in radio-transmission conflict.

B. FEATURES OF CLIQUE

CLIQUE detects all maximal cliques (subnets) or determines that the primary net itself is maximal. Each primary net or subnet may consist of no more than six stations. Currently, a net may not consist of twelve, thirteen, or fourteen transmission "conflicts" between six stations. A transmission is defined as the ability of one station to transmit to a second station's receiver and for the second station to have the ability to transmit back to the first station's receiver. If two stations may also transmit to the same third station, a possible conflict exists and the conflict is modeled as an edge between the first two stations in the net conflict graph.

Transmission conflicts within a net are limited to one if the net has two stations and fifteen if the net has six stations.

C. CREATING A CLIQUE INPUT FILE

1. What is an Input File?

An input file can be written using any IBM-compatible text editor. Simply create a file as you would for any high-level-language program. Characteristics of the filename and filename extension you select are dependent upon your particular operating system. (See your operating system's reference manual for details.) The input each graph consists of three sections which will be explained in detail in the following paragraphs. Figure 14 represents one communication net's conflict graph and the beginning of a second net. A file containing a single net will use a maximum of 160 bytes. The actual file may contain as many nets as your text editor will allow in one file.

2. Number of Stations

The first line of the file should contain an integer from one to six which represents the number of stations in the net. Any other input will be considered an error. Enter a carriage return, leaving a blank line following the number of stations, before entering the station name as described next.

3. Station Names

Enter the name of each station in the net on separate lines. The number of names should correspond to the number of stations you entered above or an error will occur. The station names in the input file are limited only by the number of characters allowed on a line in your text editor. However, just the first ten characters of each station's name will be printed in the output file. The names may consist of any keyboard letter, number or special character. It is left up to the user to choose a

unique name for each station. The clique output may be confusing if the user does not somehow distinguish a station's name. Enter a carriage return, leaving a blank line, before entering the adjacency matrix as described next.

```
6

Denver
Toronto
Seattle
Orlando
Salinas
New  Orleans

0 1 1 1 1 1
1 0 1 1 1 1
1 1 0 1 1 1
1 1 1 0 1 1
1 1 1 1 0 1
1 1 1 1 1 0

3

Honolulu...
```

Figure 14 Input File to CLIQUE

4. Station Conflict Representation

In the graph modeling of a communication's net, vertices represent the stations and edges represent the conflict between stations. CLIQUE requires that an edge be represented as a "1" (one) and the absence of an edge be represented as a "0" (zero). To better visualize the net, we will put the stations into a matrix format. Thus each station will be both a row and a column header. Placing a "1" in the column under station two on station one's row indicates that station one and station two have a conflict. The matrix must be symmetrical. In other words, a "1" must also be in the column under station one on station two's row. An inaccurate detection of

cliques may occur if the matrix is not symmetrical. A quick check to ensure symmetry involves adding the one's across a station's row and then adding the one's down a station's column. They should be equal!

A station is not allowed to be in conflict with itself so you must enter a "0" in the position intersecting a station's own row and column. This is easily checked by verifying that the center diagonal in the matrix consists only of zeros.

You may leave as many blanks between entries as you desire but you should leave at least one for ease of reading the matrix. Each station is limited to one line (row) in the text file. You must enter a carriage return, leaving a blank line, at the end of the completed matrix before starting a new graph or an error will occur.

D. EXECUTING CLIQUE

Ensure that the program, CLIQUE, and your input file have been loaded onto your hard drive or are located in the proper floppy-disk drive. If located on your hard drive, make sure you can access the program/file. Access is obtained within a directory or via a path your system recognizes. The output file does not have to exist prior to running CLIQUE. However, you must have enough memory to store the resulting output file. The amount of memory required to store an output file should be cumulative, estimating up to 900bytes per graph.

On the command line on your IBM-compatible computer, type the following : *clique(space)input filename(space)output filename.*

Typing "clique" executes the program. Do not type in the word "space". It is merely a stroke of the space bar to separate the program and its parameters (parameters are the input filename and output filename). The input filename and output filename are per the instructions in Section C.1. If you do not type the command line as specified, CLIQUE may not operate correctly.

E. CLIQUE OUTPUT FILE FORMAT

An example of CLIQUE's output is in Figure 15. The output may be printed to the screen or to a file.

```
                The input filename is : matrix.txt

The maximal cliques for net 1 will be determined for 6
stations.

           Denver Toronto Seattle Orlando Salinas Minneapoli
Denver        0      1       1       1       1         1
Toronto       1      0       1       1       1         0
Seattle       1      1       0       1       1         0
Orlando       1      1       1       0       1         0
Salinas       1      1       1       1       0         0
Minneapoli    1      0       0       0       0         0


The maximal cliques, K, are:

     1 = { Minneapoli  Denver }

     2 = { Denver  Toronto  Seattle  Orlando  Salinas }
```

Figure 15 CLIQUE Output

1. Header

The header at the top of the file indicates the user's named input file. If an input file is not found, CLIQUE will abort. Only one input file may be used at a time.

2. Identifying the Graph

Each graph will be identified by a sequential number corresponding to its place in the input file. The number of stations, the station names, and the matrix representation will correspond to input file information. Any entries not meeting the criteria of Section C above will create an error and cause CLIQUE to abort.

3. Maximal Cliques

The maximal cliques found in the current graph will be output sequentially using mathematical set notation. For graphs consisting of six stations and twelve, thirteen or fourteen edges, an appropriate message will be generated.

F. CLIQUE MESSAGES

1. General

If CLIQUE encounters errors, then error messages will be generated. Messages will be output to the screen if an output filename is not supplied. CLIQUE does not currently check for groups of stations which might not be connected to each other. Nor does it check for matrix symmetry. In other words, the user must insure that the entry for [row_i, $column_j$] is the same as the [row_j, $column_i$] entry.

2. Error Messages and Their Explanations

- Must enter an input filename to execute CLIQUE.
- EXPLANATION : Must give an input filename parameter after invoking the program CLIQUE, per Section D.

- Must enter a valid integer less than or equal to six.
- EXPLANATION : The total number of stations per net must be between one and six. This is entered in the input file per Section C.2.

- Cannot find the input file ____.
- EXPLANATION : You entered an input file on the command line which does not exist or cannot be accessed by CLIQUE.

- In net ____, to represent an edge :
- Enter a "1" or a "0" followed by a blank space.
- There is an illegal character at matrix position row ____, column ____.
- EXPLANATION : An illegal character is entered in the input file in the position CLIQUE expects to be reading the net's adjacency matrix. Check Section C for input file format.

- In net ____, station ____ must be connected to at least one other station.
- Such an edge must be indicated by a "1" in the input matrix.
- EXPLANATION : The entry for the indicated station contains only zeros which indicates an isolated station. Either remove the station from the net or connect it to at least one other station.

- In net ____, position ____ for (station name) is not valid.
- The entry on a diagonal must be a "0".
- EXPLANATION : A station may not be in conflict with itself. The input file must have a center diagonal containing only zeros in order to avoid a station "looping" to itself.

- This program cannot detect the maximal cliques for a net of six stations with ____ edges.
- EXPLANATION : CLIQUE cannot currently detect cliques for a net of six stations and twelve, thirteen or fourteen conflict edges.

3. Other Messages and Their Explanations

- The maximal cliques for net ____ will be determined for ____ stations.
- EXPLANATION : Header which inserts a sequential number for the net in the order it occurs in the input file and indicates the number of stations in the current net.

- The maximal cliques, K, are :
- EXPLANATION : Lists the sets of cliques determined for the input net.

- The input filename is : ____.
- EXPLANATION : Header for the output which indicates the input file used.

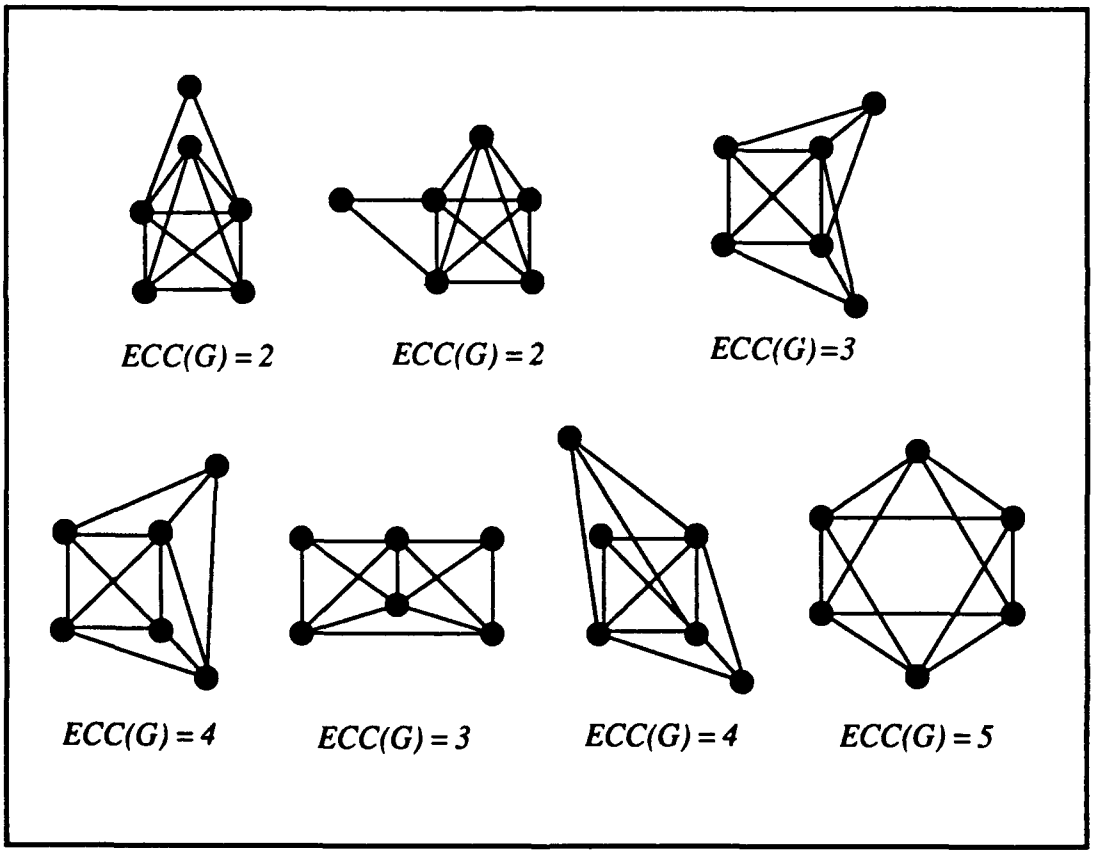
APPENDIX C

CLIQUES NOT IN THE PROGRAM

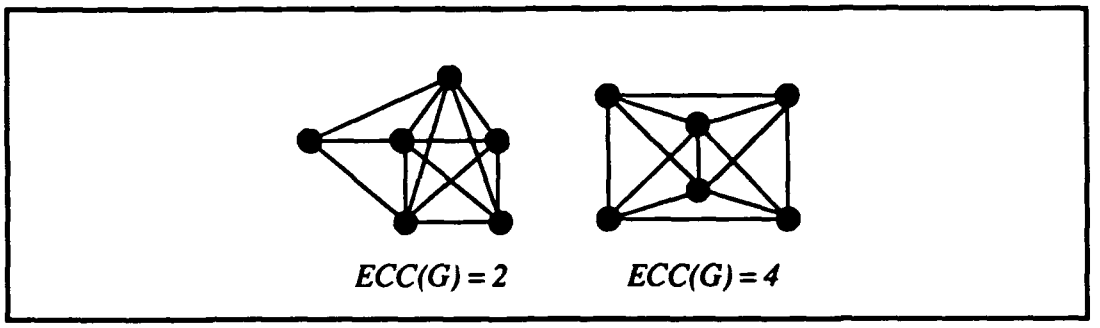
A. OVERVIEW

The following graphs are those which are not yet programmed in CLIQUE and each has six vertices.

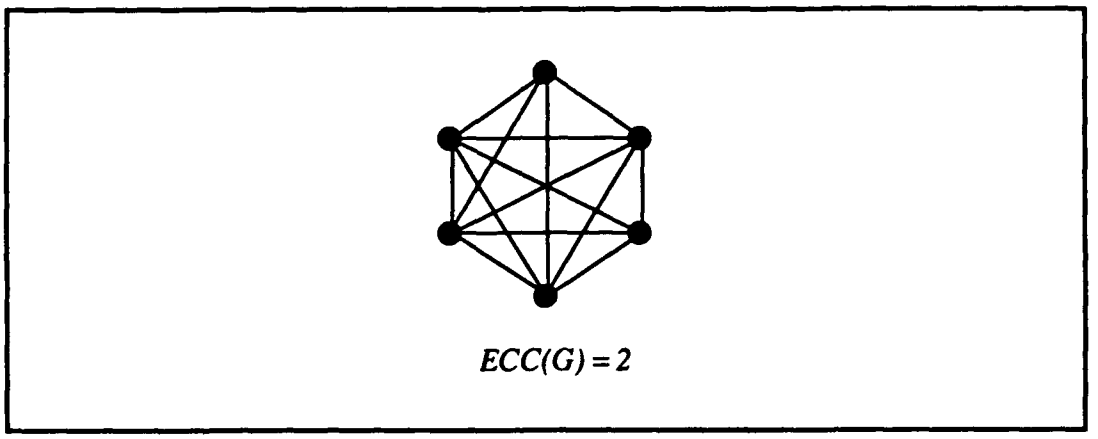
1. Conflict Graphs with Twelve Edges and Six Stations



2. Conflict Graphs with Thirteen Edges and Six Stations



3. Conflict Graph with Fourteen Edges and Six Stations



B. PROGRAMMING

The conflict graphs in Section A above should be programmed in Turbo Pascal and inserted into procedure sixvert in order to be compatible with CLIQUE. The programming of these ten graphs would complete the "brute-force" method of finding the maximum number of maximal cliques in a graph of less than six vertices. There is no guarantee that all possible graphs from one to six vertices have been found. However, the program can be easisly modified to include such a graph if one is found.

LIST OF REFERENCES

1. Hintze, D. W., *Examining a Subproblem of the Frequency Assignment Problem Using a Conflict Graph*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1990.

2. Pullman, N. J., "Clique Covering of Graphs IV. Algorithms," *SIAM Journal on Computing*, v. 13, February 1984.

3. Manber, U., *Introduction to Algorithms A Creative Approach*, Addison-Wesley Publishing Company Inc., 1989.

4. Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall Inc., 1988.

5. Raychaudhuri, A. and Roberts, F. S., "Generalized Competition Graphs and Their Applications," *Methods of Operations Research*, Anton Hain, Konigstein, West Germany, 1985.

6. Office of Naval Research Report 72-123, *Finding a Maximum Clique*, by R. Tarjan, March 1972.

7. Kou, L. T., Stockmeyer, L. J., and Wong, C. K., "Covering Edges by Cliques with Regard to Keyword Conflicts and Intersection Graphs," *Communications of the ACM*, v. 21, February 1978.

8. Bron, C. and Kerbosch J., "Finding All Cliques of an Undirected Graph[H]," *Communications of the ACM*, v. 16, September 1973.

9. *Turbo Profiler Version 1.0 User's Guide*, Borland International, 1990.

10. Hergert, D., *Mastering Turbo Pascal 5*, Sybex, 1988.

BIBLIOGRAPHY

1. Balas, E., and Yu, C.S., "Find a Maximum Clique in an Arbitrary Graph," *SIAM Journal of Computing*, v. 15, pp. 1054-1068, November 1986.

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

3. Gavril, F., "Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph," *SIAM Journal of Computing*, v. 1, pp. 180-187, June 1972.

4. Misra, J., "Necessary Subproblems in Maximal Clique Construction," *Proceedings of the 7th Hawaii International Conference on System Sciences*, Western Periodicals Co., pp. 124-128, 1974.

5. Roberts, F.S., "Applications of Edge Coverings By Cliques," Discrete Applied Mathematics, v. 10, pp. 93-109, 1985.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code MA Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5100	1
4.	Department Chairman, Code CS Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	1
5.	Professor Kim A. S. Hefner, Code MA/Hk Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5100	2
6.	Professor Man-Tak Shing, Code CS/Sh Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5100	2
7.	Capt. Kristi J. Bell 5241 Beachway Drive NE Olympia, WA 98506	2