Department of Applied Analysis
and Computer Science

University of Waterloo

Technical Report CS-73-29

November, 1973

# METHODS FOR PRODUCING VISUAL DISPLAYS OF LINEAR GRAPHS

by

R. Brien Maguire

**Faculty of Mathematics**

**University of Waterloo**

**Waterloo, Ontario**

**Canada**

CONCORDIA CUM VERITATE

**Department of Applied Analysis**

**&**

**Computer Science**

Department of Applied Analysis
and Computer Science

University of Waterloo

Technical Report CS-73-29

November, 1973

# METHODS FOR PRODUCING VISUAL DISPLAYS OF LINEAR GRAPHS

by

R. Brien Maguire

TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

Figure Page

Figure Page

## ABSTRACT

This thesis investigates display methods for representing linear graphs. The basic aim of the research involved is to discover whether it is possible to create an automated computer process for generating visual reppesentations of linear graphs.

A visual representation of a graph is a picture or display of the graph. For the purpose of the thesis this picture is assumed to be one drawn on the screen of an IBM 2250 Graphic Display Unit.

The first two chapters introduce the problem and describe earlier research in the area, including the concept of the automorphism group of a graph.

Chapter III discusses the suitability of the automorphism group of a graph as a basis for generating visual representations. It is shown that the symmetries in the automorphism mappings can be portrayed visually.

Chapter IV considers the tradeoffs between algorithmic and heuristic methods for calculating the automorphism group and discusses the reasons for choosing the heuristic method. It is shown that, by using interactive heuristics to calculate the automorphism group, it becomes practical to use the automorphism group to generate visual representations.

In Chapter V several visual representation procedures are examined, two of which were actually implemented on an interactive graph processing system. The degree of success of these routines is evaluated and the tradeoffs between general and specific types of representation routines are considered.

Finally, possible improvements and applications of the thesis research are described and the results are reviewed in light of the original goals for the thesis.

# CHAPTER I : INTRODUCTION

This thesis discusses some problems which arise when visual representations of graphs[1] are presented on a computer terminal which contains a two-dimensional display screen. The process of generating a visual representation of a graph requires two elements; a non-visual representation of a graph such as its incidence matrix and a computer program which, given this matrix or similar representation, can produce a visual display of the graph. The major goal of the research presented in this thesis has been to develop an automated computer facility for generating these visual representations. The generation process is not completely automatic in that it requires user interaction to direct its operation.

It is not really necessary to define a visual representation of a graph as a picture on a graphic display. For example, a diagram of a graph in a text on graph theory is equally well a visual representation of a graph. However, in order to describe algorithms and data structures for the generation of representations which are part of an interactive graphics system, a visual representation of a graph will mean a screen display representation of the graph.

While the visual representation of graphs does not have to be an intrinsic or even a necessary part of graph theory, the ability to model even complex relationships pictorially has led to the use of graph theory

1. The definition of a graph is found in Appendix A. The thesis contains many other graph-theoretic terms, definitions of which are given in Appendix A.

in many different areas and applications. Computers normally work with a numeric representation of a graph such as its adjacency matrix. Humans, however, usually find a visual representation more meaningful.

Drawing a picture of a graph can be a very tedious process. A rough picture is drawn and then redrawn repeatedly, moving vertices and edges until the result appears satisfactory or usually until one runs out of time, paper or patience. Moreover, the final picture may only appeal to the person who drew it. For example, if a graph is a tree then it is also bipartite but a diagram of its tree structure is of little interest to someone concerned with its bipartite nature. In fact, one cannot even be sure that given the same graph again the result of this trial and error method will produce the same or even a similar visual representation. One graph can have an infinite number of visual representations since any picture of the graph is a visual representation. Moreover, as indicated above, the choice of visual representation may be made so as to illustrate various properties of the graph such as its tree structure or polygonal nature.

The system and, in particular, the representation routines described in the thesis constitute the first automated process for manipulating graphs. Previous graph processing systems allowed the user to work with data structures especially designed for manipulating graph-theoretic entities. Even the systems incorporating a graphic display required the user to generate his own visual representations in essentially the same trial and error approach described above. The major improvement was that the system now provided a light-pen and display screen in place of pencil and paper.

Thus, this is the first graph processing system which through a combination of heuristic techniques and semi-automatic procedures generates visual representations which are meaningful to the user. The user does not generate the representation but uses the computer as a visual aid, a new tool for working with graphs.

There are two important criteria which must be satisfied by any automated process for generating visual representations of graphs:

1) The process must use a systematic approach which generates the same visual representations for the same graph every time it is used. The same approach also should be taken with similar graphs so that a particular class of graphs can be treated and the results used to compare the different graphs in a systematic fashion.

2) The process or display system must allow the user to create and test his own representation schemes. The wide range of possible applications for such an automated visual representation process demands a flexible system where old representation schemes may be easily replaced or altered to suit a particular application.

A decision was made to use the symmetries of a graph as given by its automorphism group as a basis for generating the visual representations. The detailed reasoning behind this decision is discussed in Chapter III. However, it is possible to indicate briefly how this approach satisfies the above criteria. Firstly, using the automorphism group of the graph implies a systematic approach by the very nature of the automorphism group. It is invariant in the sense that it will be the same every time it is calculated: It is also possible to use the different mappings of the automorphism group to generate several distinct visual representations of the graph. Moreover, the symmetries of the graph as given by its automorphism group reveal the basic structure of the graph.

A cycle of length one in an automorphism is a vertex fixed by the mapping. Cycles of length two indicate vertices which are interchanged. Cycles of length three or greater represent vertices which are rotated. These three types of vertices are used to generate the visual representations of the graph. The representation in Figure 1.1 shows a simple graph which can be interpreted as a mirror-image reflection of vertices across an axis of symmetry. Alternatively, it could be interpreted as a rotation of the vertices about a central point.

The second criterion for the display process says that it must be easy to change the representation schemes. Most of the representation schemes described in the thesis, however, generate visual representations based on the symmetries of the automorphism group of the graph. This is not owing to a lack of flexibility in the display system as it has been designed to facilitate easy implementation of alternative representation schemes. Several such alternative schemes are described in Chapter IV.

It is not a trivial problem to create a useful or general visual representation scheme. However, the task is made much simpler by using

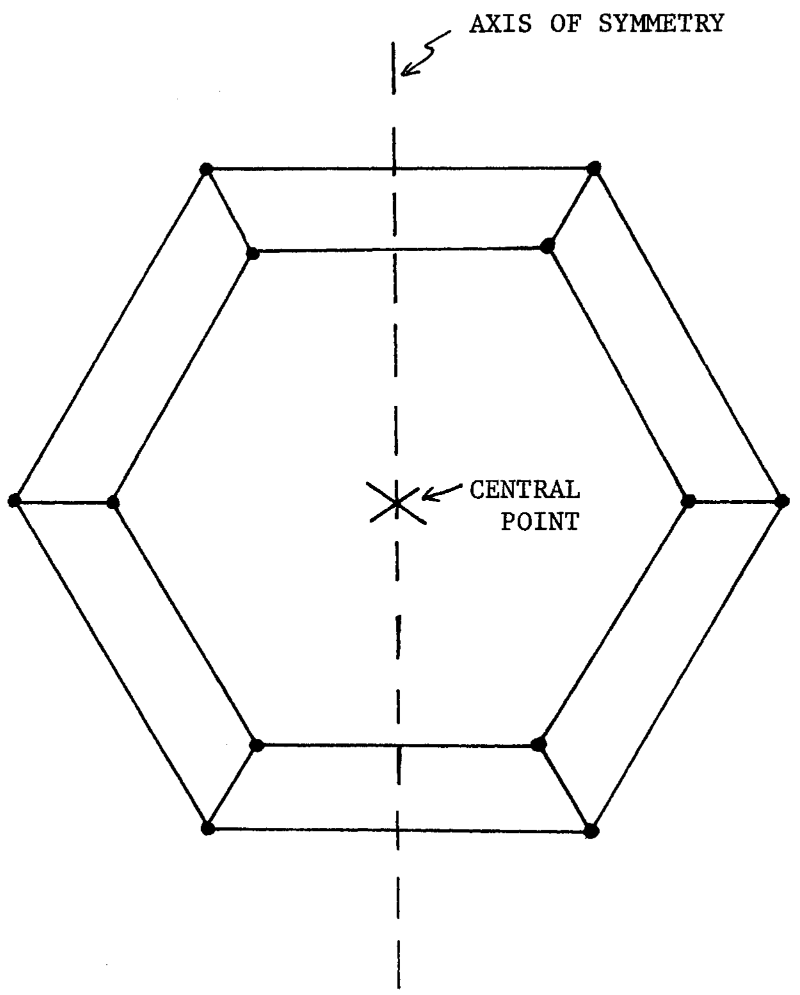


FIGURE 1.1 MIRROR-IMAGE OR ROTATIONAL SYMMETRY

the symmetries of a graph as given by its automorphism group as a common starting point. The symmetries can then be interpreted in a manner most suitable to the application at hand. The symmetries of graphs can be used to create representation schemes for all classes of graphs, thus providing a link between the wide range of applications in which such an automated representation system might be useful.

The symmetries of a graph are also directly related to the internal structure of the graph. In a sense, these symmetries indicate what parts of the graph "look alike". That is, the symmetries in the automorphism group can be interpreted by a visual representation of the graph.

The above discussion is not meant to imply that the choice of the automorphism group as a basis for generating visual representations of graphs is obvious or even necessary. The discussion in Chapter III describes an initial attempt to create representation schemes without using the automorphism group and how the decision to use the automorphism group was reached.

Once the decision was made to base these visual representations on the symmetries of a graph, the graphics programs with which to actually implement the tool were needed. Since none were readily available at the time, it was necessary to create the GSYM graphics system. This interactive system for the creation and manipulation of graphs on a display screen was especially designed as an appropriate tool for working on the visual representation of graphs.

The GSYM system is described in detail in a forthcoming University of Waterloo research report. However, the thesis will occasionally refer to features of the GSYM system which were especially designed to facilitate the creation of an automated process for generating visual representations of graphs.

At this point a basis for the representation process has been decided and a tool is available with which to implement the system. There remain the two problems of obtaining the automorphism group of a graph and creating a visual representation for each mapping in the automorphism group. The resolution of these problems forms the most important part of the thesis. Chapter IV contains a discussion of the automorphism group problem. In general, computing the automorphism group of a graph is very expensive in terms of the execution time required for its calculation. The differences between algorithmic and heuristic approaches to automorphism group calculation and the advantages and disadvantages of each approach are described. The reasoning behind the choice of the heuristic approach is found here. A heuristic used by Unger (xi) for computing graph isomorphisms has been adapted to the automorphism group problem. The main point of this discussion is the question of whether a heuristic approach (or any approach) for automorphism group calculation is practical for use in an interactive system for generating pictures of graphs.

Following this, the heuristics which have been implemented in a program for automorphism group calculation are discussed. The discussion centres around the tradeoff between the execution time required by the final automorphism group calculation routine and that required by the heuristics.

This routine considers all the possible mappings of the vertex set with the exclusion of those mappings which have been eliminated by the earlier use of the heuristic tests. It is an algorithm in that it will find the automorphism group. However, it is relatively expensive when there is a large number of possible mappings. The rationale behind the heuristic approach is to eliminate as many mappings as possible using the various tests available without using more execution time than would have been required by calling the final routine sooner. With this in mind the discussion indicates how the various heuristics were chosen and implemented in order to minimize the execution time necessary to run these tests.

Chapter IV closes with a discussion of the advantages in implementing an interactive version of the automorphism group calculation program. That is, the user is able to choose and repeat tests at will, specifying new parameters if he so wishes. For some of the heuristics it is even possible to view the results as the routine proceeds and to stop the test if it does not appear to be making any headway in reducing the number of mappings which must be checked later by the automorphism group algorithm. The real advantage of this interactive implementation is that only through this approach did it become practical to compute and use the symmetries of the automorphism group of a graph to generate visual representations of it.

Chapter V investigates the most important aspect of the whole process of visual representation of graphs -- the actual creation of the displays themselves. The problem to be solved is to generate from the mappings of the automorphism group, which are actually just permutations of the vertex set of the graph, three dimensional visual representations of the graph. The first representation procedure that is treated is a very general routine which produces a visual representation stressing the two basic types of symmetries in graphs, mirror-image and rotational symmetry. The success and failure of this attempt to handle all classes and types of graphs with just one representation routine is discussed in detail. One of the major conclusions reached from this investigation was the need for a special postediting feature which would allow the user to make alterations in a generated visual representation before going on to the next.

The next representation routine examined was implemented in order to provide special treatment for the large class of graphs which are trees. If a graph is known to be a tree then this routine can be used to generate a visual representation of the tree structure of the graph. Moreover, the symmetry given by the mapping is used to shape the tree so as to reveal the mirror-image and rotational symmetries. The remainder of the chapter discusses the advantages of designing visual representation schemes for special classes of graphs over the use of more general representation schemes. Several possible techniques are suggested and evaluated in some detail.

Chapter VI evaluates the degree of success of this attempt to automate the visual representation process in terms of the original goals that were set. Suggestions are made for improvements in the existing system and several possible applications are considered.

# CHAPTER II : HISTORICAL BACKGROUND

We believe that the system we have developed for generating visual representations of graphs is the next logical step in the evolution of special purpose programming languages for graph processing. In order to fully understand this evolution several existing graph processing languages and the various approaches used in their implementation are reviewed briefly. Following this, some of the recent research on the automorphism group problem and those results which have been utilized in this thesis are examined. Finally, we discuss the research which has led to this investigation of the problem of visual representation of graphs.

The ability to manipulate graph-theoretic constructs on a computer is becoming quite commonplace. In fact, many programming languages such as PL/1 now contain all the list processing facilities necessary to define and manipulate graphs without too much difficulty (ii). However, only those languages and language extensions designed solely or primarily for handling graphs are of interest here. Thus, the many graphics languages and systems such as GRAF (iii), B-LINE (iv) and GPAK (v) which have appeared in recent years are not considered. These languages are designed to allow a user to communicate with a graphics display unit but, since the basic elements of the language such as the data structures and commands are not defined in graph-theoretic terms, they are not graph processing languages. Of course this does not mean they cannot be used for working with graphs.

There are three approaches which have been used to implement graph processing languages. The simplest approach is to create a subroutine package which is called upon to perform the graph related operations. This method only requires the existence of the subroutine library as the subroutine calls are made within the context of the existing language, eliminating the need to modify the original language compiler.

A second more complex method is to define extensions to a language in the form of new data types and commands which operate on the new data types. The graphs are represented using these special data types. This requires a similar type of subroutine library to that of the above method. Moreover, it is necessary to translate the special graph processing statements into the format of the host language. Hence, the extended language program is passed through a preprocessor program which produces an equivalent program written in the base language. This program is then passed to the host compiler. The most difficult approach is to implement the graph processing language separately from any base language. This method, while requiring the most effort to implement, is usually the most efficient in terms of the execution time used by programs written in the language. It again requires a program library of run-time routines, as well as a new compiler for the language. Depending on the complexity of the language this can be very costly in terms of the manpower and machine time required to write the compiler.

Now that the usual methods of implementing special purpose languages such as those used for graph processing have been reviewed, several examples of such languages are described briefly. One example of an extension to an existing language is GASP, which is a graph processing extension of PL/1.(vi) A new data type called an 'object' has been added to PL/1. An object is a conglomeration consisting of a character string, a name and a set. The elements of a set are themselves objects. The name is used by the system to reference various objects. The character string part of an object may be assigned character string values by the programmer and is generally used to identify objects when doing output. A graph is represented as an object whose set contains the vertices and edges of the graph. The vertices and edges are again objects, the set associated with an edge being the pair of incident vertices and the set associated with a vertex being the set of all incident edges and adjacent vertices. GASP additions to PL/1 consist of:

1) PL/1 statements which declare or assign values to GASP data types,
2) GASP procedure calls, and
3) Type functions which return a name, integer or truth value.

The declaration statements define the GASP variables as names, character strings, integers, bits or truth values. The basic component of a set is an element which has been declared as a name. Before being used the name must be allocated storage using the $ALLOC type function. This function actually creates space for an object which becomes the definition of the name passed to it as an argument. The character string value of the object is the same as the name used.

The GASP function $CHANGES is used to add objects to or delete objects from the set of another object. In this manner one adds (deletes) vertices and edges to (from) a graph. The programmer works at the name level; that is, he references an object through its name. The procedure PELEMSK is a special I/O routine which prints the contents of a set in the form of a list. For example, 'CALL PELEMSK(A)' will print the character string value of A followed by the character string values of all the objects in the set corresponding to the name A. Graph input is handled by calling READGR, which reads a sequence of digits defining a graph as a set of paths. A new path is indicated by a minus sign preceding the starting vertex of the path.

For example, the input sequence 3,4,5,-1,2,3,-4,1,0 would cause READGR(B) to create the graph shown in Figure 2.1. The graph would be the object definition of the name B. READGR assigns node i the character string value 'Ni'. The ith edge created in building the graph is assigned the character string value 'Bi'. These character string values may subsequently be changed by the user. The command PELEMSK(B) would produce the following output:

```
B  = (B1, B2, B3, B4, B5, N1, N2, N3, N4, N5)
B1 = (N3, N4)
B2 = (N4, N5)
B3 = (N1, N2)
B4 = (N2, N3)
B5 = (N1, N4)
N1 = (B3, B5, N2, N4)
N2 = (B3, B4, N1, N3)
N3 = (B1, B4, N2, N3)
```

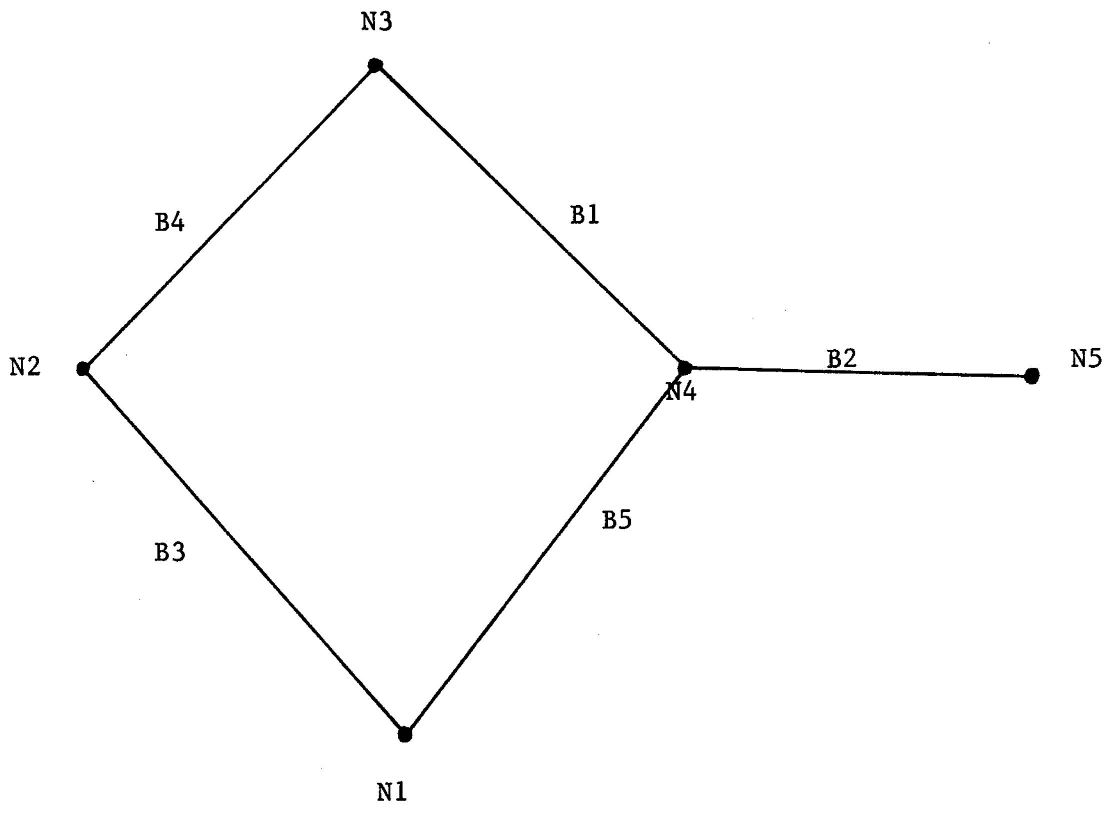


FIGURE 2.1   A GRAPH IN GASP

N4 = (B1, B2, B5, N1, N3, N5)

N5 = (B2, N4)

GASP contains several graph-theoretic functions such as INCIDENT, a routine which determines whether a vertex is incident to an edge. In theory GASP is quite powerful because it utilizes a very general set theoretic data structure. However, in practice it is definitely limited due to its relatively poor I/O facilities and the comparatively large amount of trivial programming which must be done to set up and manipulate a graph. For example, each object of a set must be assigned a character string value in a separate assignment statement in order to be able to distinguish the different elements of the set should it be printed by PELEMSK. Thus, GASP is a potentially powerful but somewhat tedious tool for graph processing.

HINT is a graph processing language which is a stand-alone language.(vii) HINT extends the semantics of list processing to graph processing. The language is based on the concept of a hierarchical graph, a graph with vertices that may themselves contain or represent graph structures. The basic elements in HINT are numbers, character strings and names. The names are given numeric or character string values or they may be 'lists' composed of a collection of names and numbers. A 'set' is a list with no elements repeated. A 'list structure' is a list where the elements may themselves be similar to those of LISP.(viii) Directed graphs are also defined in a similar list type notation, a graph being a set of pairs of the form (pair 1,...,pair n) where each pair consists of a vertex, followed by $\rightarrow$ ,followed by the link set of the vertex. The link set is simply the set of vertices adjacent to the given vertex.

For example,

GRAPH = (V1 → (V2,V3), V2 → (V5), V4 → (V3), V5 → ( ), V3 → (V2,V1,V4) ) defines the graph with the name GRAPH shown in Figure 2.2.

The language contains numerous primitives which perform tests on the lists as well as handle such functions as I/O. There are also many primitives which are used solely for operating on graphs. Since the graphs are themselves lists any list function may be used on a graph. The graph primitives perform such functions as the addition and deletion of vertices and edges. There are also functions for finding the complement of a graph, the shortest path between vertices, the valence of a vertex and other elementary graph properties. These two languages are just an indication of the current interest in graph processing languages. As well, Friedman has implemented GRASPE 1.5, a graph processing extension of LISP 1.5 (i) and an extension to FORTRAN called GTPL has been used by King (xii) to experiment with many graph theoretic algorithms.

This list does not cover all such existing graph processing languages, but it does illustrate their general nature. The above languages have concentrated on improving the ease of implementing graph theoretic algorithms by providing the basic functions and data structures used in manipulating graphs. There is no provision for interaction between the user and the system, nor has there been any attempt to incorporate a visual representation of the graphs. The next step in the development of graph processing languages was the design of an interactive graph processing system using a visual display.

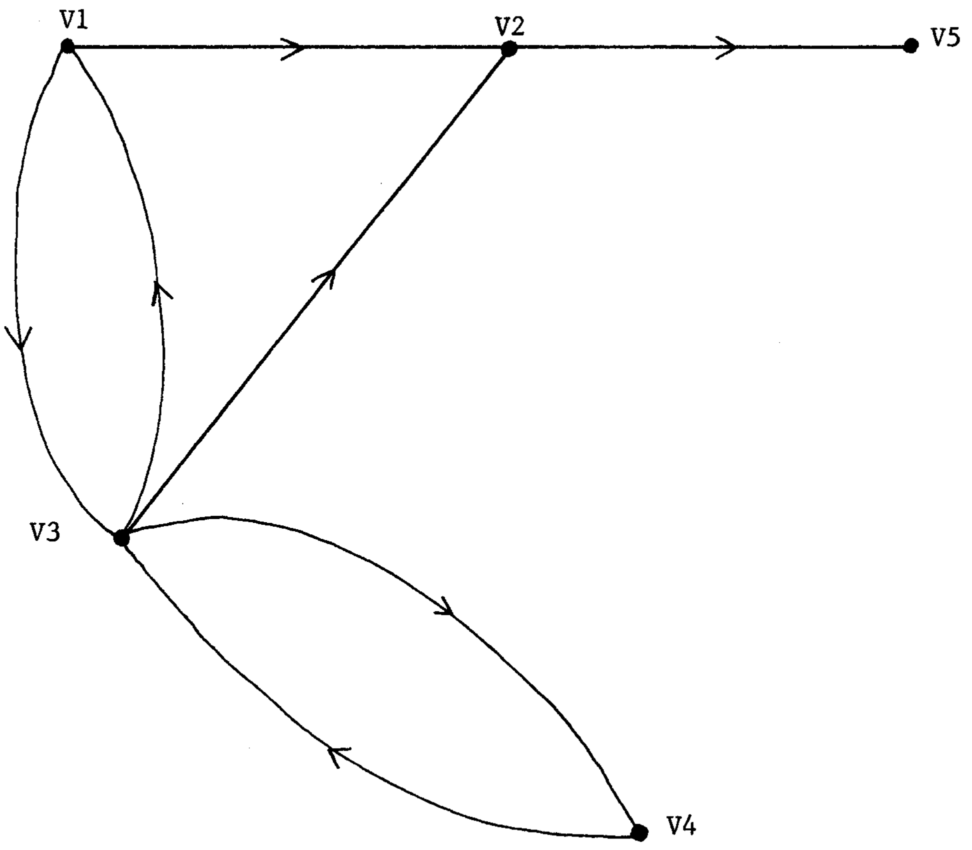


FIGURE 2.2 A GRAPH IN HINT

Wolfberg's Interactive Graph Theory System is just such a system.(ix) The user communicates with the system in an interactive mode via a DEC-338 graphics display terminal.(x) Once more, a set-theoretic approach is taken in order not to restrict the power of the language. Graphs may be defined in terms of entities which include atomic objects, ordered pairs and sets. Each atom, pair or set may have any amount of associated data as its properties. An atom is a data type having no structure other than its associated properties. A set contains any number of elements, each of which is again an entity. Normally a graph is defined as an ordered pair where the left-element of the pair is the set of vertices and the right-element is the set of edges. Each edge is an ordered pair where the left-element is the positive vertex and the right-element is the negative vertex. Each vertex will ordinarily be an atom, but the set-theoretic nature of a pair allows any element of a pair to be another pair or set. Hence, it is possible to represent a graph of graphs. Some of the more common structures which are possible are:

1. A set of edges whose order represents a path,
2. A set of edges as a property of a vertex - these might be the set of out-directed edges,
3. A set of vertices as a property of a vertex - possibly the adjacent vertices, and
4. An integer as a property of a vertex - depth in a tree for example.

The visual representation of the graph consists of vertices and edges with or without labels. The edges take the form of straight lines between the vertices unless the edge is a loop in which case it is displayed as an elliptical closed curve. Edges may also have arrows displayed at their midpoint if they are directed edges. An arrow has one of eight orientations depending on the angle of the edge. The user uses the light-pen to select commands on the screen in order to direct the operation of the system. For example, he selects the creation of a vertex as the next operation and then uses the light-pen to position the vertex on the screen. The user also has available a set of special control buttons and a typewriter keyboard. The display is under the supervision of an interactive monitor which allows the user to execute programs on the central computer. These programs may be system routines or, as is more likely, they may be user written routines which have been incorporated into the system. These routines are coded in the special graph processing language used on the system. This language is interactive in that it can react to user intervention at the terminal.

Wolfberg has created a powerful graph processing language with the added attraction of having a display capability which permits the user to see the graphs with which he is working. Of course, the user must draw the graphs himself, either while working at the screen or through his programs. As already mentioned, the thesis is concerned with what we consider must be the next stage in this evolution, that is, the automated generation of visual representations of graphs. Besides being able to see a graph that he has drawn, the user should be able to request the system to show him various visual representations of the graph in which he is interested. It must also be possible for the user to easily implement his own visual representation schemes and call them as if they were system routines.

As indicated earlier, the work of several individuals on the problem of calculating the automorphism group of a graph is reviewed briefly here. As can be seen in Appendix A the concept of the automorphism group of a graph is quite straightforward. An automorphism is simply a permutation of the vertices of a graph which preserves the structure of the graph. One way to find the automorphism group then would be to generate all the permutations of the vertex set, checking each one against the definition of an automorphism. Here is the key to the problem. The number of permutations of the vertices of a graph with N vertices is N!. Thus, for a graph with 10 vertices it would be necessary to examine over three and a half million permutations. Actually the problem is to find an efficient algorithm for calculating the automorphism group. By 'efficient' is meant an algorithm where the number of operations required tends to some small finite power of N rather than N to the power N.

Kagno (xvii) has calculated the automorphism groups of all graphs having up to six vertices, while Hemminger (xviii) has extended this list to include directed graphs having up to six vertices. Prins has investigated the automorphism group of a tree.(xiii) However, he has not developed any calculation methods suitable to graphs in general.

Instead of attempting to compute the automorphism group directly there has been significant interest in attempting to find the automorphism group of a composite graph using the automorphism groups of the graphs in the composition. For example, the group of the sum of nonisomorphic graphs is the direct sum of the groups of the graphs in the sum. Harary has investigated the groups of the cartesian product and composition of undirected graphs.(xix) Unfortunately, this line of attack has yet to produce an efficient algorithm.

Mowshowitz has used a result of Chao (xx) to suggest an algorithm for calculating the automorphism group of a digraph.(xv) Chao has shown:

Lemma (xx, p.489):

Let $\alpha$ be a one-one mapping of the vertices of a graph G onto the vertices of a graph H and let $P_\alpha$ be the permutation matrix corresponding to $\alpha$ . Let $A_G$ be the adjacency matrix of the graph G and $A_H$ the adjacency matrix of the graph H. Then $\alpha$ is an isomorphism of G onto H if and only if $A_G*P_\alpha = P_\alpha*A_H$.

Corollary (xx, p.490)

Let $\alpha$ be a permutation of the vertices of a graph G onto itself and $P_\alpha$ be the permutation matrix corresponding to $\alpha$ . Then $\alpha$ is an automorphism of G if and only if the adjacency matrix A of the graph G commutes with $P_\alpha$ , that is, $A*P_\alpha = P_\alpha*A$.

Therefore, one can find the automorphism group of a graph by generating all the permutations of its vertex set and finding those permutations whose matrix commutes with the adjacency matrix of the graph. To do this directly is out of the question because of the amount of time that would be required. Instead, Mowshowitz has created an algorithm for constructing the general solution M for the matrix equation $M*A = A*M$, for all permutations with permutation matrix $P = M$. The solution M gives the automorphism group of the graph. Mowshowitz uses the algorithm to derive conditions on the automorphism group which insure that the information content of two graphs is identical. The information content is calculated using an information measure defined by Mowshowitz.

In order to use the algorithm it is necessary to find a nonsingular matrix U which transforms A into Jordan canonical form, that is, $A=U*\tilde{A}*U^{-1}$. Substituting $U*\tilde{A}*U^{-1}$ for A in the equation $A*M = M*A$ gives $U*\tilde{A}*U^{-1}*M = M*U*\tilde{A}*U^{-1}$, implying that $\tilde{A}*(U^{-1}*M*U) = (U^{-1}*M*U)*\tilde{A}$. If we let $M_{\tilde{A}} = U^{-1}*M*U$, then $M_{\tilde{A}}$ is an arbitrary solution to the matrix equation $M_{\tilde{A}}*A = A*M_{\tilde{A}}$. Thus, to find all the solutions M of the equation $A*M = M*A$ we need the matrix $M_{\tilde{A}}$ (which is a function of a fixed number of parameters) and the transforming matrix U. The final result of all this is that the general solution M is given by $M = U*M_{\tilde{A}}*U^{-1} = (x_{ij})^{n}_{i,j=1}$.

If N is the number of parameters of M in the above general solution, then Mowshowitz shows that a permutation matrix corresponding to a permutation of the vertices commutes with the adjacency matrix of the graph if andonly if a nonhomogeneous system of $n^2$ linear equations in N ($\leq n^2$) unknowns is consistent. This is summarized by the following theorem:

Theorem 2.1 (xv, p. 83)

Let $\alpha$ be a permutation of VG and $P_\alpha$ the corresponding permutation matrix, $M = (x_{ij})^n_{i,j=1}$ the general solution to $M*A = A*M$ where G is an n vertex digraph and N is the number of parameters of M. Then $\alpha$ is in the automorphism group of G if and only if the nonhomogeneous system

$$x = \begin{cases} 1, \text{ if } k = j \\ \\ 0, \text{ otherwise} \end{cases} \qquad 1 \leq i, k \leq n,$$

of $n^2$ linear equations in N unknowns is consistent.

This algorithm is not the 'efficient' algorithm being sought owing to the amount of computation involved in finding $M_A^{\sim}$ and U. Moreover, it is still necessary to determine for each permutation whether the system of parameters $x_{ij}$ is consistent.

Kately appears to have been the first to develop an algorithm based on information which is readily available from the structure of the graph.(xiv) He attempts to reduce the number of permutations which must be considered as possible elements of the automorphism group. He gives the following definitions and theorems:

Definition (xiv, p. 23)

$$F_s = \{u \varepsilon VG \mid (s,u) \varepsilon EG\}.$$

$$T_t = \{v \varepsilon VG \mid (v,t) \varepsilon EG\}.$$

$F_s$ contains the successors of s while $T_t$ contains the predecessors of t. A 'successor' of a vertex v is a vertex w such that there is a directed edge from v to w. A 'predecessor' of a vertex v is a vertex w such that there is a directed edge from w to v. Note that for undirected graphs an automorphism must map the vertices adjacent to a given vertex onto those vertices adjacent to the image of the given vertex.

Definition (xiv, p.23):

Given a graph G, let

$$O_k = \{s \mid \text{outvalence of } s = k\},\ k = 0,1,2,\ldots$$

$$I_k = \{s \mid \text{invalence of } s = k\},\ k = 0,1,2,\ldots$$

Hence, $O_k$ is the set of vertices in VG having outvalence k and $I_k$ is the set of vertices in VG having invalence k.

Theorem 2.2 (xiv, p.24)

Let G be a graph. If f is in the automorphism group of G then

$$f(O_k) = O_k,$$

$$f(I_j) = I_j \text{ for all } k \text{ and } j.$$

This theorem only states the rather obvious fact that an automorphism preserves the valencies of the vertices of the graph.

Theorem 2.3 (xiv, p.25)

Let G be a graph. If f is in the automorphism group of the graph then

$$f(F_s) = F_{f(s)},$$

$$f(T_t) = T_{f(t)}.$$

This theorem states that an automorphism maps the successors (predecessors) of a vertex onto the successors (predecessors) of the image of the vertex.

Theorem 2.4 (xiv, p.25):

Let G be a graph and let f be a permutation on VG. If $f(F_a \cap I_j) = F_{f(a)} \cap I_j$ for all $I_j$ defined by G and for all a in VG with the provision $f(\emptyset) = \emptyset$, then f is in the automorphism group of the graph.

Kately then suggests using the above theorem to calculate the automorphism group of a graph. This is accomplished by considering the possible permutations according to theorems 2.2 and 2.4 For example, consider the graph in Figure 2.3. The intersection table for this graph is given in Table 2.1.

Theorem 2.2 indicates that the set of vertices {1,3,5} must map onto {1,3,5,} and the set {2,4,6} must map onto {2,4,6} for any automorphism f in the automorphism group. A permutation taking 1 onto 2 is therefore impossible. For example, Table 2.1 indicates that only cycles starting with 13, 15, 35, 53, 51, 31, 24, 26, 46, 64, 62 and 42 can be permutations in the automorphism group. It is therefore necessary to consider only those permutations which have not been eliminated, checking them using Theorem 2.4, that is, $f(F_a \cap I_j) = F_{f(a)} \cap I_j$. Kately considers this method to be suitable for calculating the automorphism groups of graphs by hand. Actually, this would be highly impractical for all but relatively small graphs.

The result of Theorem 2.3 has been used by W. Jackson to implement a computer program for finding the automorphism group of a graph.(xxi) His algorithm builds tree-like structures where each node represents a mapping of a vertex in the graph. For example, all the possible permutations of a graph with three vertices are given by the three trees

TABLE 2.1 : INTERSECTION TABLE FOR GRAPH IN FIGURE 2.3

| | I = {1,3,5} | I = {2,4,6} |
|---|---|---|
| F = {2,3} | 3 | 2 |
| F = {2,4} | | 2,4 |
| F = {4,5} | 5 | 4 |
| F = {4,6} | | 4,6 |
| F = {1,6} | 1 | 6 |
| F = {2,6} | | 2,6 |

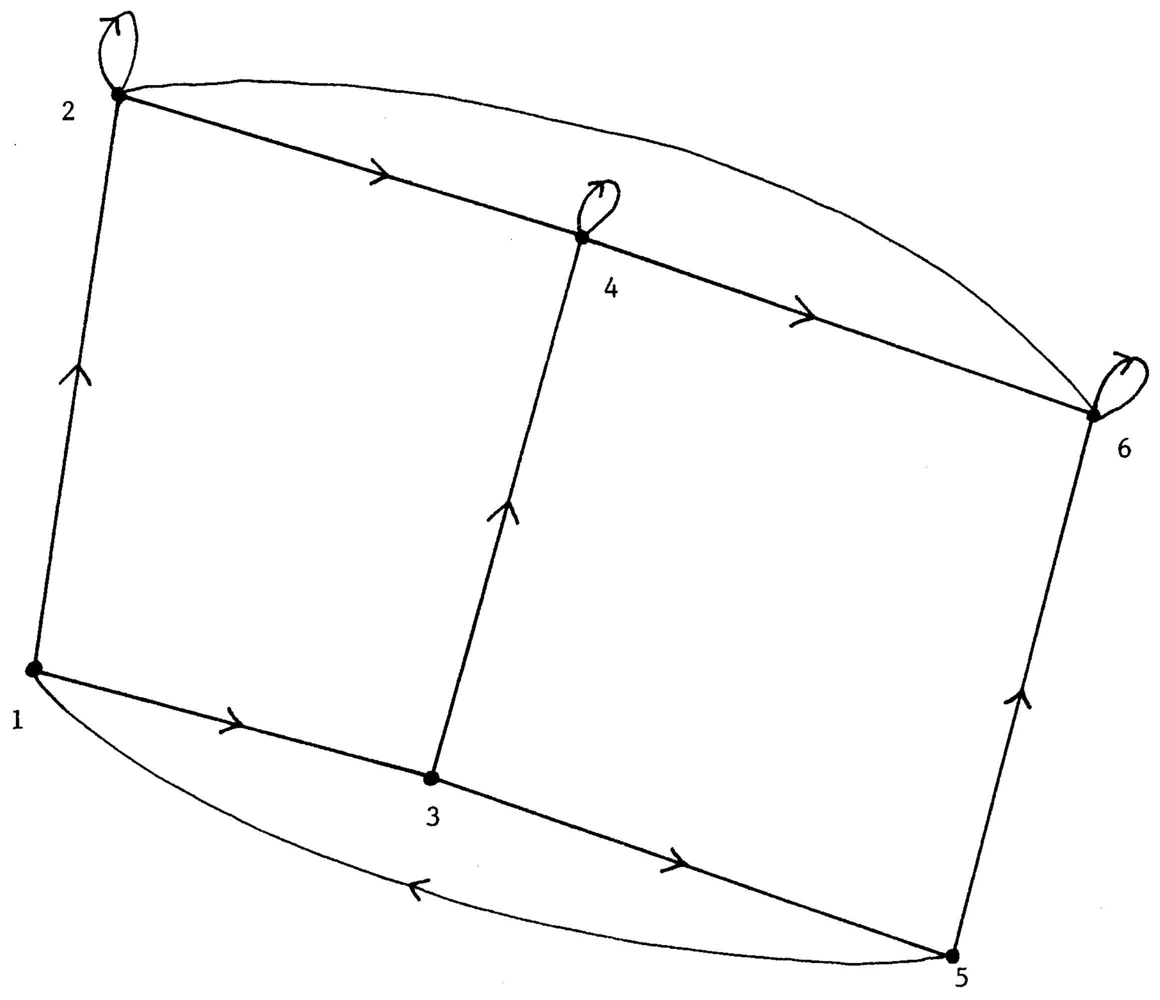


FIGURE 2.3

in Figure 2.4. A permutation is found by starting at the root vertex of any tree and moving down the tree until an endpoint is reached.

Jackson's algorithm creates each tree branch by branch and as the permutation represented by a branch is generated the program checks to see if it can be an automorphism. If this check fails at any node then the remainder of the current branch is abandoned; that is, the rest of the branch is not generated. The concept is similar to the technique used by Kately in that time is saved because it is not necessary to check all the permutations, nor, in many cases, is it necessary to check a complete permutation. Since essentially the same algorithm is used in Chapter IV it is explained in some detail at this point. A discussion of the speed and efficiency of the algorithm is postponed until Chapter IV.

The notation used in the discussion is as follows:

a) "$i \to j$" indicates a mapping of vertex i onto vertex j. Thus, if a branch of a tree generated by Jackson's routine contains a node where $i \to j$ the the permutation corresponding to the branch maps vertex i onto vertex j. However, this does not imply that j is mapped onto i by the permutation. This is true only if the branch also contains a node $j \to i$. Thus, the two mappings $i \to j$ and $j \to i$ are distinct mappings and one does not imply the other.

b) "$\{v_1, v_2, \ldots, v_n\} \to \{w_1, w_2, \ldots, w_n\}$" indicates a mapping of a set of vertices $v_1$, $v_2$, ..., $v_n$ onto a set of vertices $w_1$, $w_2$, ..., $w_n$. With n elements in each set there are n! possible permutations. Thus, the above notation indicates all the possible permutations of a set onto another set.

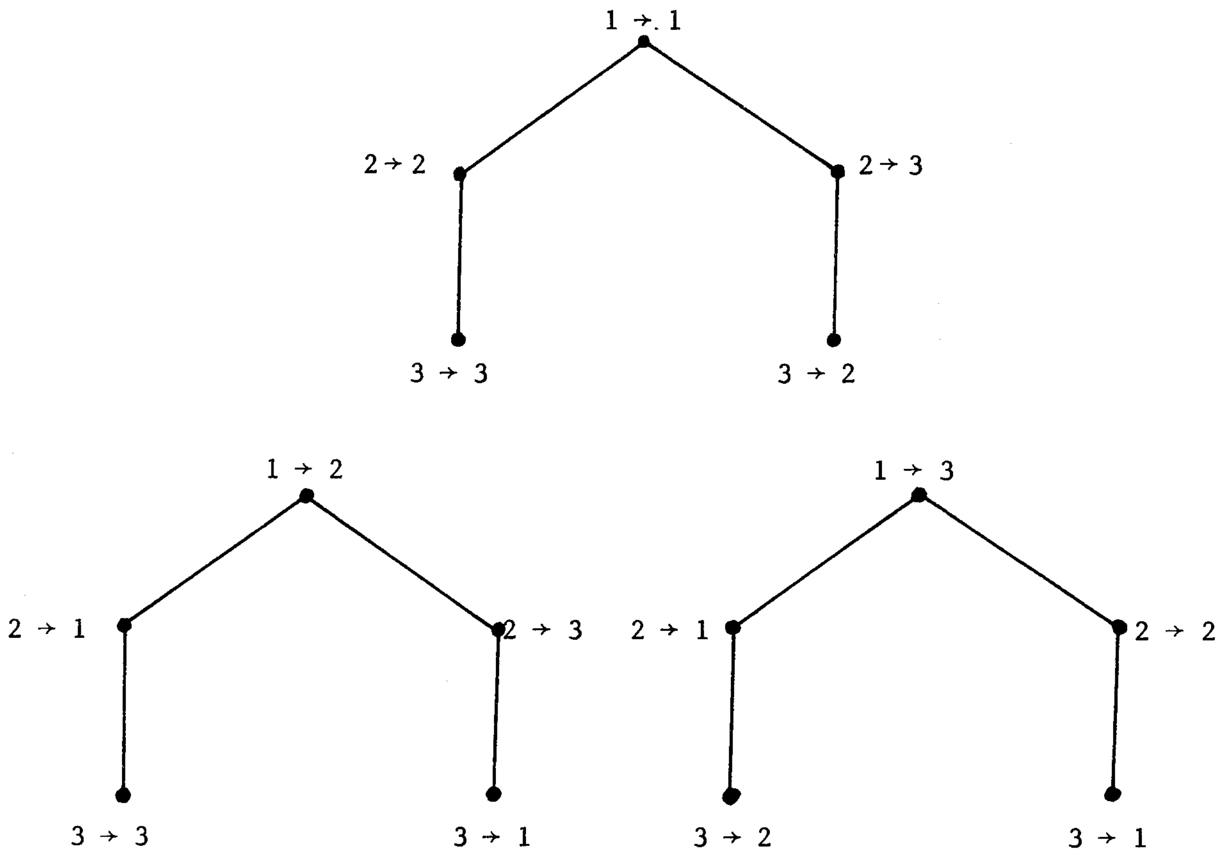


FIGURE 2.4 PERMUTATIONS REPRESENTED BY TREES

c) The notation "=>" is used to indicate that one mapping implies another. That is, by Theorem 2.3 a mapping of a vertex (set of vertices) will imply a mapping of the successors of the vertex (set of vertices) onto the successors of the image(s) of the vertex (set of vertices).

d) The usual notation for permutations as found in Dean (xxiv) is used for vertex permutations. Thus $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix}$ indicates a permutation where $1\to4$, $3\to1$, $2\to3$ and $4\to2$.

The program numbers the vertices of the graph from 1 to n where n is the number of vertices in the graph. The root of each tree generated by the program represents a mapping of vertex 1 onto another vertex in the graph. Thus, a total of n trees may be generated, one each for the mappings $1\to1$, $1\to2$, ..., $1\to n$. Beginning with the root of the first tree, $1\to1$, Theorem 2.3 says that the successors of vertex 1 must map onto the successors of vertex 1. That is, the mapping $1\to1$ may imply the mappings which become the nodes on the next level of the tree.

In order to find all the automorphisms in the automorphism group, the program actually considers at the $i^{th}$ level of the tree only mappings of vertex i onto another vertex.

The same procedure is used throughout the whole process of generating the tree. By Theorem 2.3, the mapping represented by any node indicates which mappings will be considered next. This, of course, means that the algorithm attempts to include in each branch only mappings which may be in an automorphism. It is possible that the vertices involved in a mapping at level i-1 of the tree have no successors or do not imply any mappings of vertex i. In this case, the program must consider all the possible mappings of vertex i.

Thus, for the second level of the first tree, the program checks all mappings of vertex 2 implied by $1\to1$. If $1\to1$ implies no such mappings, then all the mappings $2\to1$, $2\to2$, ..., $2\to n$ are checked. When a mapping is 'checked' it means that the valencies of the two vertices are compared. If they do not match then the mapping and all subtrees that would have this mapping as a root are abandoned. For example, the program in considering a root mapping of vertex 1 actually only generates trees for mappings of vertex 1 onto a vertex with matching valence. In general, the number of trees generated by the program is much less than the number of vertices in the graph.

If the valencies of the vertices do match, the program also checks to see that the mapping does not contradict any previous mapping in the branch being traversed. A contradiction exists in the mapping $i\to j$ if vertex j has been already used as the image of some other vertex. This follows from the fact that an automorphism is a 1-1 onto mapping of the vertex set of the graph. Finally, the program checks that the mappings implied by this mapping (if any) do not contradict any previous mappings. In this case it is also necessary that the implied mappings do not map any vertex onto a different image from an earlier mapping of the vertex. This point is made clear by the example below.

The next example describes the operation of the algorithm for the graph in Figure 2.5. This detailed explanation will indicate how Theorem 2.3 is used to limit the number of mappings that must be considered and how often only part of a branch need be traversed.

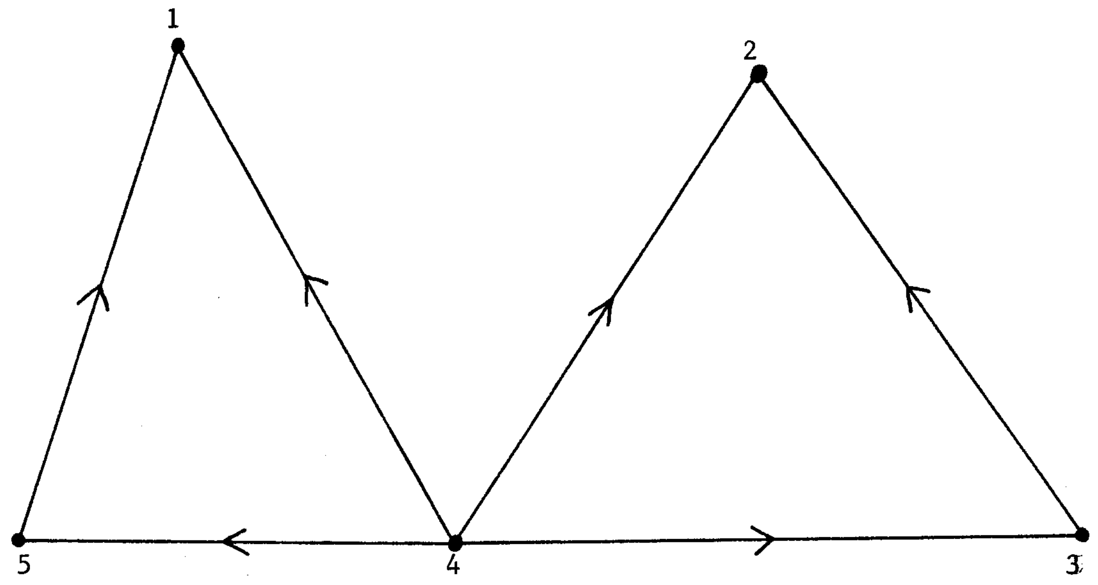


FIGURE 2.5

Example:

Start with 1→1. Vertex 1 has no successors so no mappings for the next level of the tree are implied by 1→1. Thus, the program must check all the possible mappings of vertex 2. 2→1 contradicts the previous mapping 1→1, but 2→2 is possible. The program goes to the next level of the tree. The possibilities 2→3, 2→4 and 2→5 will be considered later. Vertex 2 has no successors meaning that all the possible mappings of vertex 3 must be considered on the next level. 3→1 and 3→2 contradict the previous mappings 1→1 and 2→2 respectively. 3→3 checks out because the implied mapping 2→2 (i.e. 3→3 =>2→2) agrees with the previous mapping of vertex 2.

No other mappings are implied by 3→3 so the program proceeds to the next level. No mappings of vertex 4 were implied by 3→3 so all the possible mappings of 4 must be considered. 4→1, 4→2 and 4→3 all contradict previous mappings. The next possibility is 4→4. 4→4 => {1,2,3,5} → {1,2,3,5}. This seems to indicate that there are 24 implied mappings to be checked. However, any use of vertices 1, 2 or 3 would contradict a previous mapping in the current branch (permutation). Thus, only 5→5 is possible. It checks out and the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$, the identity automorphism has been generated.

The program now backs up the tree to consider the possible mappings which were not checked and to generate branches (permutations) as far as possible. The first such mapping is 4→5. The valencies do not match and this is the last possible mapping of vertex 4 in this subtree so the program backs up one more level to consider 3→4. The valencies do not match. $3 \to 5 \Rightarrow 2 \to 1$ which contradicts the previous mapping $1 \to 1$. Backing up again, the mappings $2 \to 3$, $2 \to 4$ and $2 \to 5$ all fail because the valencies do not match. There is nowhere to back up to from here so this tree is finished and only one automorphism has $1 \to 1$.

The program now considers the next possible mapping of vertex 1, $1 \to 2$. A tree with this root is generated and the result is a single automorphism given by the permutation $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$. No more trees are generated as the root mappings $1 \to 3$, $1 \to 4$ and $1 \to 5$ all fail the valence check. The program is finished and has found only one non-trivial automorphism. Figure 2.6 illustrates part of the tree structure that would be generated in finding the automorphism group of the graph in Figure 2.5. The tree with root node $1 \to 2$ has not been included.

The above discussion has examined attempts to find an efficient algorithmic solution to the automorphism group problem. However, the same problem can be attacked in a heuristic fashion. Unger has designed a heuristic program which attempts to find a solution for the related problem of graph isomorphism.(xi) Given two graphs the problem is to determine whether the graphs are isomorphic and, if so, to find an isomorphism of the graphs. As with the graph automorphism problem there is a simple enumerative, algorithmic solution, but one which is impractical for almost all graphs because of the amount of time required to complete the algorithm. Unger's program GIT, short for Graph Isomorphism Tester, partitions the graphs into possible vertex pairing lists according to various vertex properties. For example, if two graphs are isomorphic they must have the same number of vertices with the same valence. Moreover, the set of vertices of the first graph having valence k must map

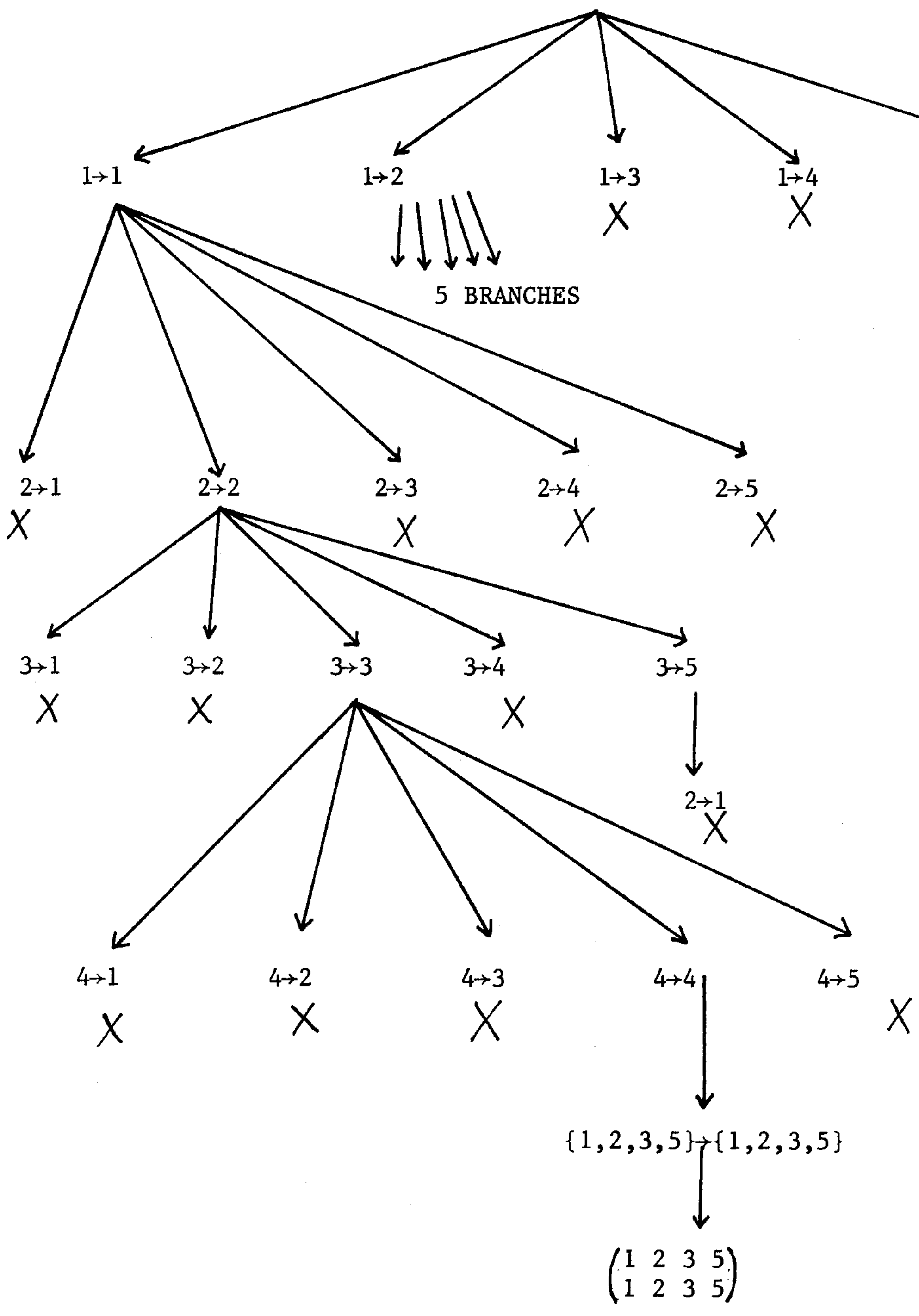


FIGURE 2.6 : CALCULATION TREE

onto the set of vertices of the second graph having valence k. These pairing lists may be further refined by comparing other properties of the vertices in the two graphs. In this manner GIT attempts to reduce the number of possible mappings which might be isomorphisms. Should there be a mismatch between two pairing lists, that is, one graph has a different number of vertices having some property than are found in the other graph, then the graphs cannot be isomorphic and GIT is finished.

The final routine in GIT is an algorithm similar to Jackson's which considers all the possible mappings according to the pairing lists. This routine will determine whether the graphs are isomorphic regardless of how much the heuristic tests have refined the pairing lists. The only difference is that the more the pair lists are refined the fewer the number of mappings that must be checked. Among the various vertex properties that Unger suggests for refining the pair lists are the following:

1) The number of vertices which can be reached from a vertex by a path of length n. The path need not be simple.
2) The number of vertices which can be reached from a vertex by a simple path of length n.
3) A binary valued function which indicates whether a vertex is in a closed circuit or not.
4) Variations and combinations of the above including variations for undirected and directed graphs.

The next question is "What has all this got to do with the problem of calculating the automorphism group of a graph?". The question will be answered in full in Chapter IV when we investigate a heuristic approach to the automorphism group problem. In fact, this discussion will show that the heuristics used by Unger as well as many others can be applied to the problem of reducing the number of permutations of the vertex set which may be automorphisms of the graph.

There has not yet been any mention of any research related to the visual representation of graphs. The reason for this is quite simple; there has been very little work in this area although there has been considerable research into such indirectly related problems as determining whether a graph is planar. Wolfberg has written a tree layout routine using his interactive graphics system.(ix) His routine accepts a user drawn tree and a user selected root vertex. The tree is then redrawn on the screen with the root at the top of the screen. The vertices are positioned at different levels according to their distance from the root vertex. They are also drawn so that no edges intersect. The endpoints of the tree are equidistant from each other along the horizontal direction although they may obviously be at different levels vertically. The positioning of the end points determines the rest of the positioning since each remaining vertex is positioned so that its horizontal location is the average of the horizontal co-ordinates of the connected vertices which are directly below it.

The problem with this routine is that it requires the user to draw the tree beforehand. Moreover, the ordering of the vertices at each level depends upon the order in which they appear in the original tree. This order is preserved by the layout routine with the exception of any reordering necessary to prevent edges from intersecting. However, the order of those vertices at level i+1 with respect to the vertex at level i to which they are connected will always be the same as in the original graph. In Chapter V we shall present a tree representation routine which uses only information based on the structure of the graph in positioning the vertices.

Most of the research to date on the visual representation of graphs has centered around the question of whether a graph is planar and, if so, how to represent the graph. For instance, Fary has proven that if a finite graph can be drawn on the plane it can be drawn using only straight line edges provided the graph does not contain two edges joining the same vertices and does not have any loops.(xxii) Tutte has also investigated the problem of drawing a graph.(xxiii) However, Tutte was mainly interested in nodally three connected graphs. Moreover, neither Fary nor Tutte was primarily motivated by the practical nature of their constructions for straight line representations of planar graphs, but were more interested in showing that such a representation was possible. As we shall see in Chapter V we do not restrict ourselves to visual representations of only planar graphs. But more to the point, the problem of whether the final visual representation of the graph is useful and pleasing to the user cannot be ignored.

# CHAPTER III : SYMMETRY IN GRAPHS

In the previous chapter it was stated that the choice of the automorphism group of a graph as a basis for generating visual representations of the graph was not an obvious choice to make. This chapter examines an attempt to create visual representations for a single class of graphs without using the automorphism group of the graphs. The representation routines described in this chapter were written before the GSYM system existed.

The purpose of the chapter is to give the reader a better understanding of the difficulties involved in creating visual representations for graphs. The discussions show how experience gained from these first representation schemes led to the decision to use the automorphism group symmetries as a basis for further investigation of representation techniques for graphs. It is also shown how deficiencies and problems encountered with these representation routines influenced the design of the GSYM system and, in fact, indicated the actual need for an interactive graphics system such as GSYM.

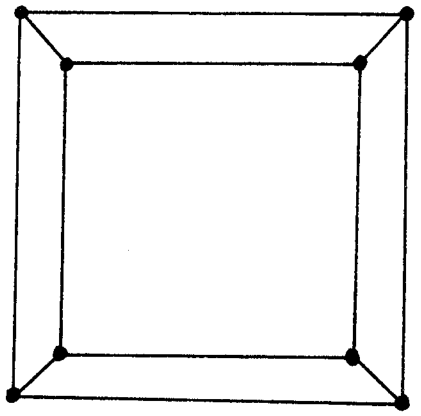
The problem of interest was how one could generate visual displays of the class, C, of cyclically 4-connected, cubic planar maps. G. B. Faulkner, of the University of Waterloo, had created an algorithm for generating these maps. The difficulty was to devise a method whereby displays of the maps could be produced on a display screen without having to draw each one manually. Initially, the problem was restricted to a special subclass of the larger class C. Faulkner catalogued the class C according to region size, that is, the number of regions in a graph, starting with six region graphs, the smallest graphs in class C. For every region size $n \geq 6$, there exists a graph consisting of two polygons of n-2 edges each such that each vertex in one of the polygons is adjacent to exactly one vertex in the other polygon. Such graphs will be referred to as prism graphs. Figure 3.1 gives several examples of prism graphs represented visually as concentric polygons. The prism graphs are actually only chains of 4-gons. By restricting the problem to that of generating visual representations for prism graphs the representation problem becomes much simpler. If the two concentric polygons can be isolated a screen display of the graph can be easily produced using the following representation scheme:

Let N be the number of vertices in a polygon and D the screen distance desired between vertices in the polygon. Let $\Theta = 180^{0} - ((n-2)*180^{0})/N$. If $(x_1,y_1)$ are the screen co-ordinates of a vertex of the polygon then the screen co-ordinates of the remaining vertices are given by the equations:

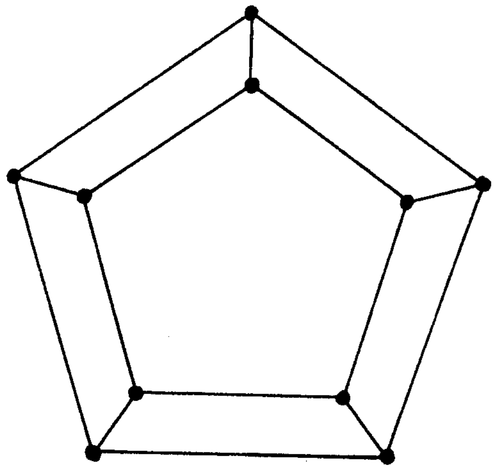
$$x_{i+1} = x_i + D*\cos(i*\Theta)$$

$$y_{i+1} = y_i + D*\sin(i*\Theta)$$

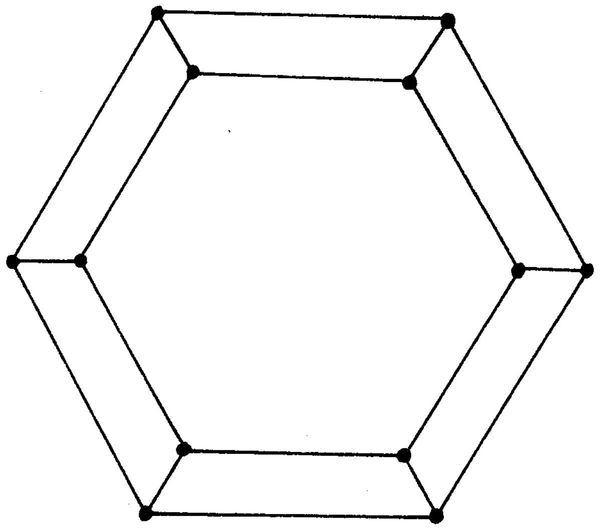
Using this scheme it is possible to display the two polygons, lining up the adjacent vertices on the two polygons in order to obtain a concentric polygon picture of the graph. The edges of the graph can be displayed as straight lines between the appropriate vertices once all the vertex co-ordinates have been calculated. Figure 3.2 illustrates



6 REGIONS

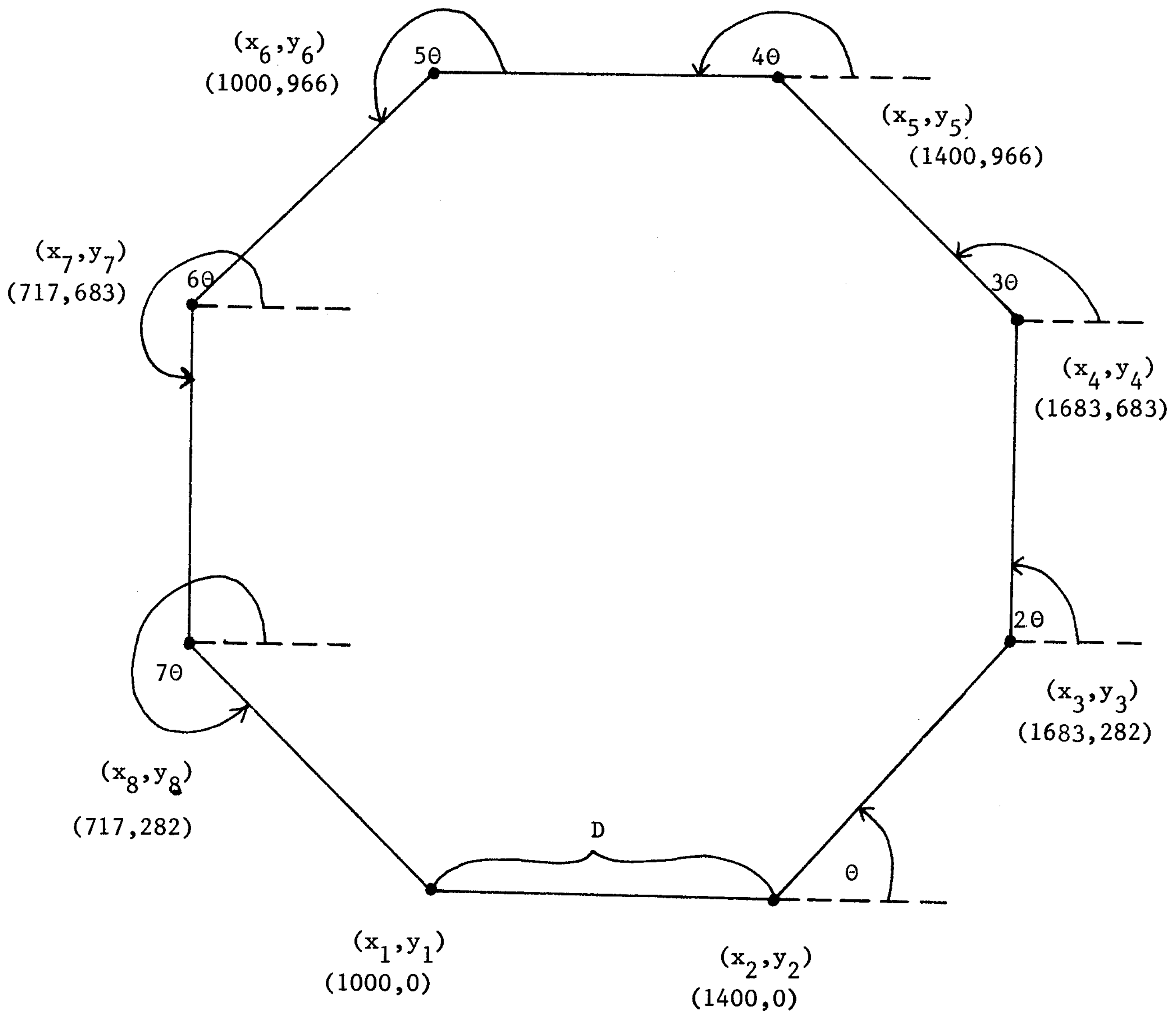


7 REGIONS



8 REGIONS

FIGURE 3.1 : SEVERAL PRISM GRAPHS



N = 8

$= 180^\circ - \frac{(N-2)\ 180^\circ}{N}$

$= 45^\circ$

$x_{i+1} = x_i + D\cos(i\Theta)$

$y_{i+1} = y_i + D\sin(i\Theta)$

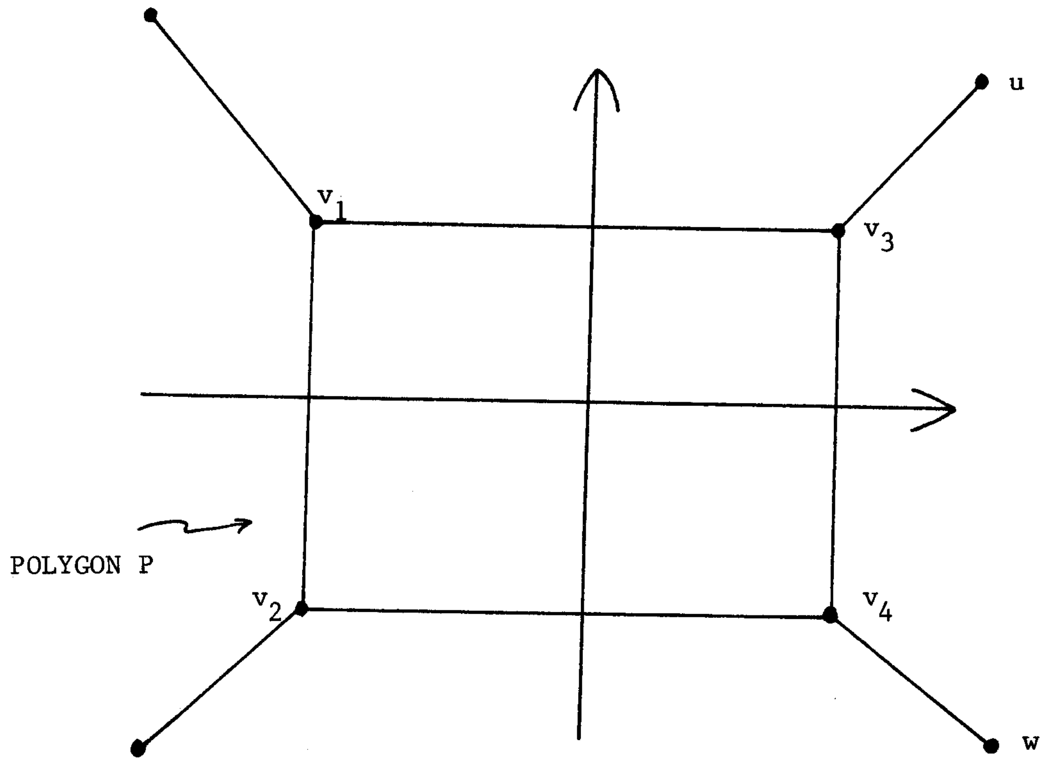
FIGURE 3.2

how a polygon with 8 edges might be displayed. If this were to be the outer polygon of a prism graph, then one would add the inner polygon by simply changing the starting co-ordinates and the value of D and repeating the above scheme.

Locating two polygons in a prism graph to serve as the concentric polygons of the visual representation is not difficult. The term 'concentric' is used to indicate the polygons being sought; it does not have any graph-theoretic meaning. These two polygons are also referred to as the 'ring' polygons of the graph because of the nature of the visual representation as a ring or a chain of 4-gons.

The algorithm for finding two concentric polygons for the visual representation builds this chain 4-gon by 4-gon. If P is the first 4-gon of this chain then P has both concentric polygons passing through it. Two of the vertices in P are in the outer ring polygon and the other two vertices are in the inner ring polygon. The choice of which pair of adjacent vertices in P is placed in the outer ring is immaterial as the algorithm will always find two concentric polygons for the visual representation. Although the ring polygons used may differ, the visual representation for a prism graph of n vertices will always be a chain of n/2 4-gons.

Figure 3.3 illustrates the possible 'directions' of the ring polygons through the starting 4-gon P. 'Direction' is another geometric term which in the strict sense is only meaningful when used with an actual display of the graph. Such terms are used here only to simplify the description of the algorithm. Since the choice of which vertex in P is to be placed in the outer ring polygon is quite arbitrary, the



CHAIN OF 4-GONS PASSES
THROUGH STARTING 4-GON P
IN ONE OF THESE "DIRECTIONS"

FIGURE 3.3 : STARTING POLYGON OF RING STRUCTURE

vertex with the smallest vertex number is always placed in the outer ring.[1]

Beginning with P and a direction through P, the chain of 4-gons is extended a pair of vertices (i.e. the next 4-gon) at a time. Thus, a 4-gon is added to the chain by adding a vertex to the outer ring and a vertex to the inner ring.[2] This process is repeated until all the vertices in the graph have been placed in the ring polygon. The following is an algorithm which locates two ring polygons in a prism graph:

1) Starting with any vertex, $v_1$, in the graph find a 4-gon P containing $v_1$. Put $v_1$ in the outer ring polygon.

2) Let $v_2$ and $v_3$ be the two vertices in P adjacent to $v_1$ and let $v_4$ be the fourth vertex in P.[3]

3) Set i = 3.

4) Let u and w be vertices which are not yet in a ring polygon and which are adjacent to $v_i$ and $v_{i+1}$ respectively. If no such vertices exist[4] the algorithm is complete and the two ring polygons have been found. Otherwise, u is the next vertex in the outer ring polygon and w is the next vertex in the inner ring polygon.[5]

5) Set i = i+2. Go to step 4.

1. This choice may be overridden by the user who may specify a starting polygon and direction through it.

2. These vertices are of course adjacent.

3. Unless overridden by the user the program will place $v_2$ in the inner ring polygon. This sets the 'direction' of the ring polygons through P. Thus, $v_3$ is in the outer ring and $v_4$ is in the inner ring.

4. That is, all the vertices in the graph have been placed in a ring polygon.

5. As shown in Figure 3.3.

This algorithm produces a visual representation for a type of graph which occurs only once for each region size in the class C. However, it forms the basis for a heuristic program for generating visual representations for the whole class C.

The next step was to investigate the method used by Faulkner to produce graphs from class C. The prism class is obtained by starting with the 6 region prism graph and adding to the graph a new edge with new end-vertices for each consecutively larger region size. The edge must be added so that it splits one of the 4-gons in the ring into two new 4-gons. Figure 3.4 illustrates how this is done. As it happens, roughly the same method is used to generate all the graphs in class C, again starting with the six region prism graph. One adds a new region by adding a 'diagonal edge' to an existing map. This diagonal edge and its new end-vertices must divide a polygon of the original map into two new polygons, both having at least 4 edges. By "diagonalizing' every polygon of every map of region size n in class C one can produce every map of region size n+1 in class C.

This procedure is simple, except that it results in the creation of many different maps of graphs which are isomorphic. Figure 3.5 shows how the only six region graph in class C can be 'diagonalized' by adding the new edge $(v_1,v_2)$ to get two different maps of what is really the same 7 region graph. The significance of Faulkner's algorithm is that is makes it possible to systematically generate one map for each non-isomorphic graph in class C. However, the point of interest is that all the graphs of class C are essentially being generated from the simple

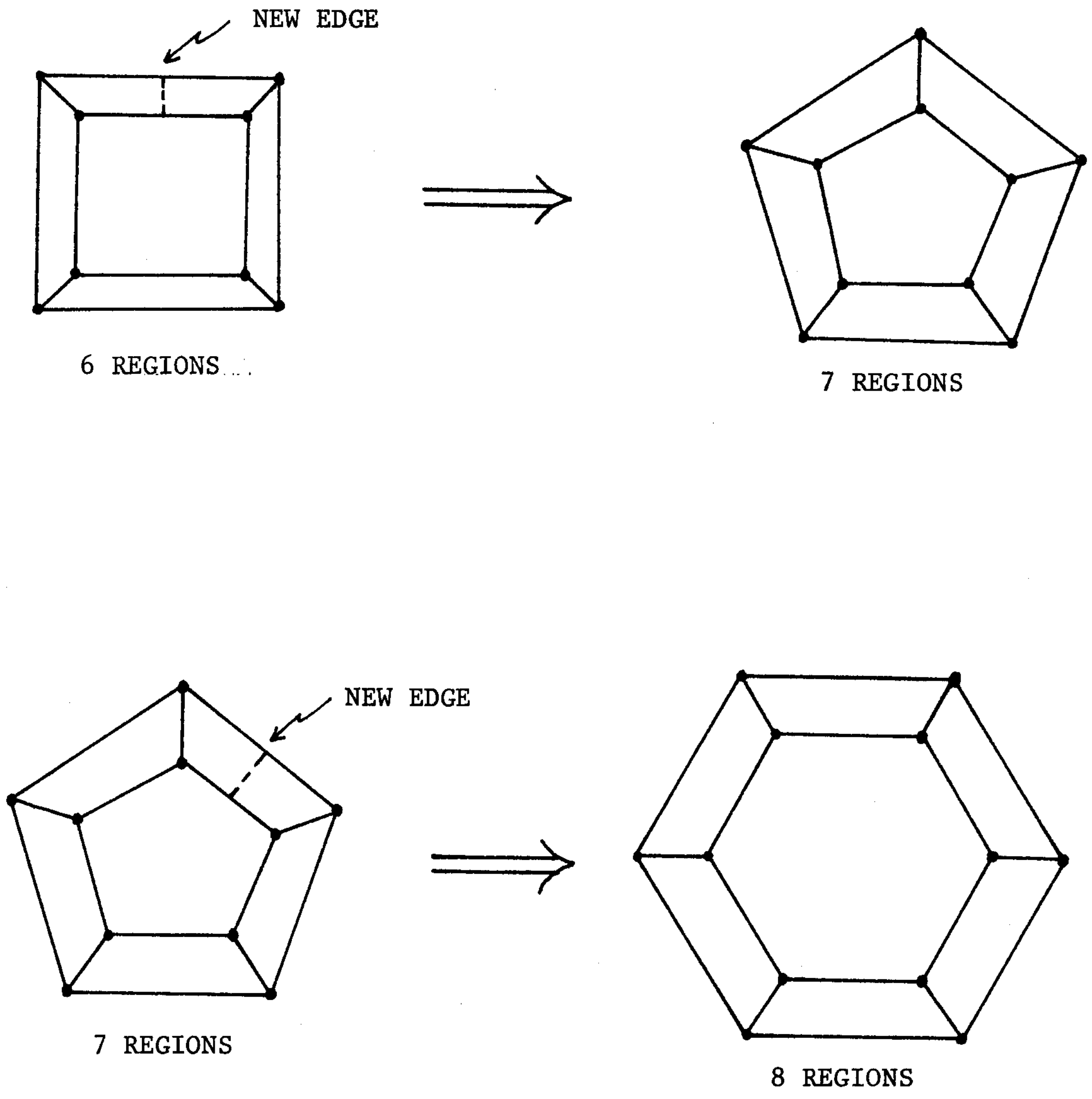


FIGURE 3.4 : FORMATION OF PRISM GRAPHS

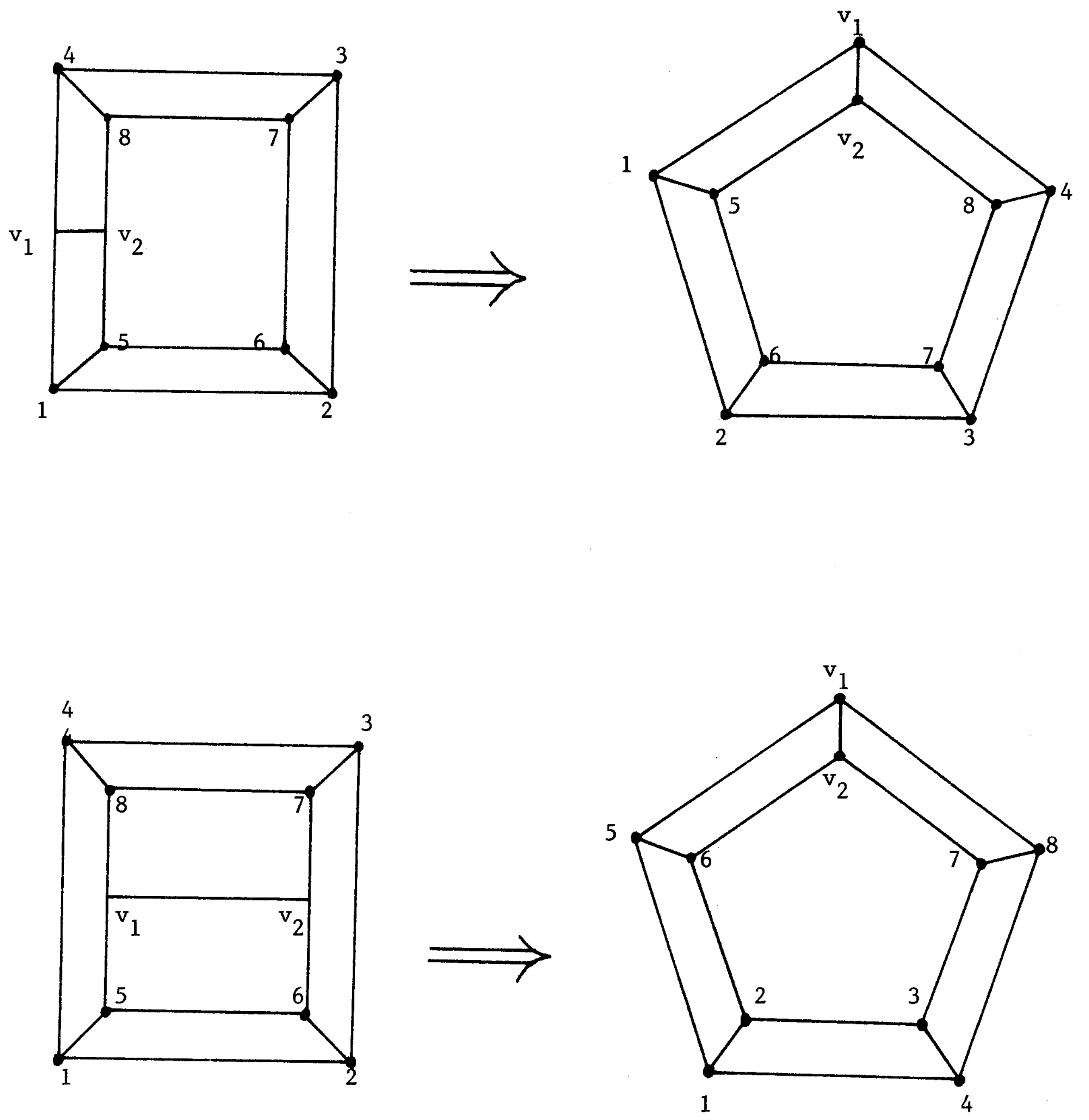


FIGURE 3.5

prism graphs, suggesting that the prism graph structure might be common to all class C graphs. That this statement is false is evident from the graph in Figure 3.6 which shows a map of an 8 region graph which cannot be transformed into a map of an 8 region prism graph. Of course, this must be the case or the prism class would be the same as the whole class C. The ring structure in Figure 3.6 is, however, very similar to that used in the maps of the prism graphs. The only real difference is the pair of vertices on the inner ring which are the end-vertices of an edge inside the inner ring polygon. This suggests that it might be possible to use the ring structure as the starting point of a representation routine for class C graphs.

With this more complex ring structure in mind the previous algorithm was modified to allow for the inclusion of vertices on the inner polygon ring which are not adjacent to vertices on the outer ring. This in turn means that it is necessary to allow for the possibility that some or maybe all of the polygons in the ring chain might contain more than 4 edges. Instead of an algorithm for finding the chained 4-gons of a prism graph, a heuristic routine has been derived which attempts to find a chain of n-gons as the ring structure of the map.

The representation routine will still attempt to display the graph as a chain of polygons. However, these polygons no longer are restricted to 4-gons but may be n-gons, that is, polygons with n edges (and n vertices) where $n \geq 4$. Figure 3.7 is an example of one such representation. The ring structure contains two 4-gons but it also contains three 5-gons. The 5-gons are placed in the ring structure so that two edges of each 5-gon are found on the inner ring polygon. The same approach will be used

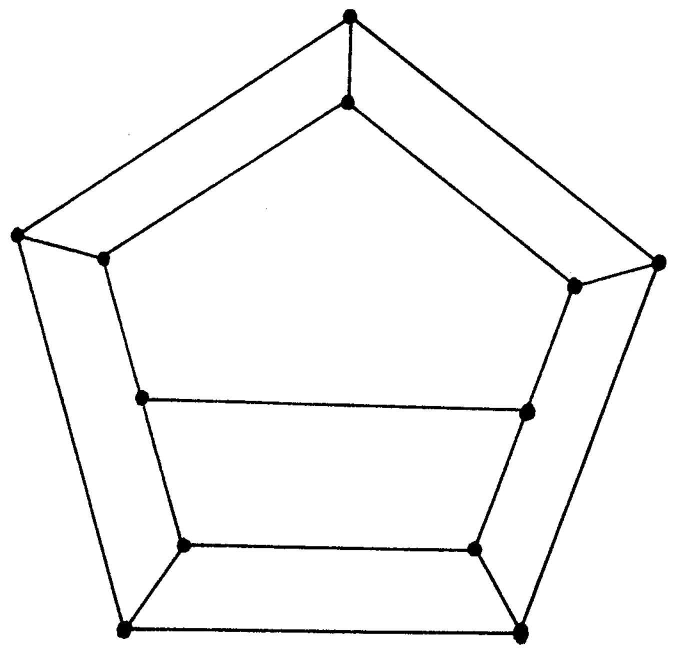


FIGURE 3.6

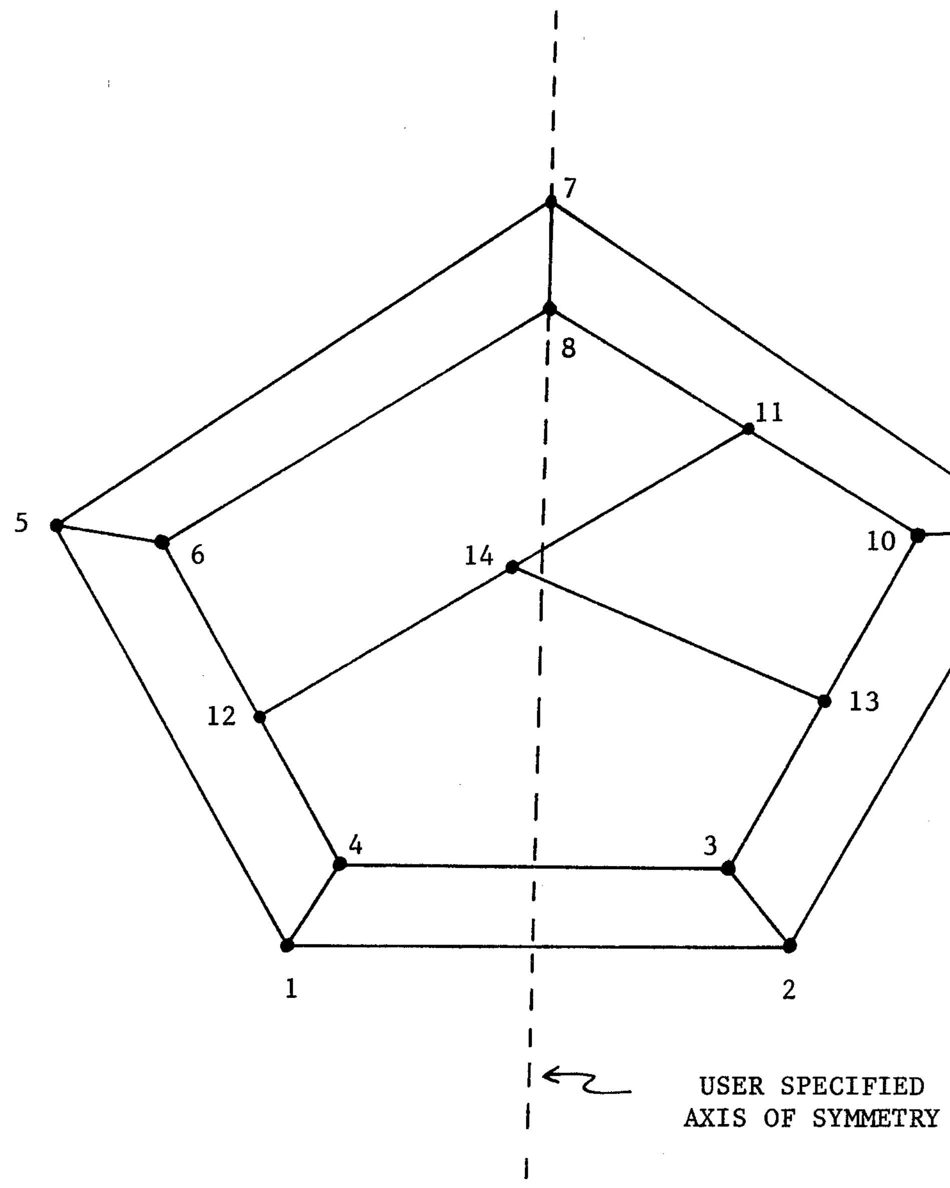


FIGURE 3.7

to add any n-gon where $n>4$. That is, where one edge was added to the inner ring polygon to add a polygon to the ring structure of a prism graph, a path of one or more edges is added to the inner ring polygon in incrementing the ring structure of a class C graph.

The representation routine was designed so that when it seeks to increment the ring structure, it attempts to add a polygon with as few edges as possible. If it fails to find a 4-gon for the next polygon it will search for a suitable 5-gon, then a 6-gon and so on. Similarly the routine attempts to find a starting polygon with only four edges. Failing this, it attempts to find a starting 5-gon, then a 6-gon and so on.

It became apparent that a limit on the number of edges allowed in any polygon in the ring structure was needed. Such a limit was needed to force the program to attempt to build a chain of small polygons if possible before admitting very large ones. Thus, N indicates the maximum number of edges allowed in any polygon in the ring structure. It is possible for the user to specify the value of N. Thus, it is possible to prevent the program from using excessive execution time testing all the possible combinations of polygons that may go in the ring structure. Moreover, it gives the user greater control over the form the representation is to take.

The program attempts to find a starting 4-gon ($n=4$) for the ring structure. If the graph does not contain a 4-gon, n is incremented by 1 and the program attempts to find a starting 5-gon. This process is repeated until a starting n-gon is found or until n exceeds the limit N.

If n>N then the user is asked whether N should be incremented by one. If so, then the process is repeated again. Otherwise, the program is finished and no representation has been found.

Once a starting n-gon is located one of the edges of the n-gon is chosen to go on the outer polygon ring. If the program is unable to complete the ring structure with this edge on the outer ring it will try all the other edges in the starting n-gon before searching for an alternative starting n-gon.

The process of extending the ring structure is very similar to that used for prism graphs. The program again attempts to add a pair of adjacent vertices to the ring polygons, one vertex to the outer ring polygon and the other to the inner ring polygon. However, for non-prism graphs the new pair of vertices[6] may not be adjacent. If they are the newest element of the ring structure is a 4-gon. Otherwise, however, the program attempts to find a path joining the two vertices. This path may not pass through any vertices already in the ring polygons[7].

If no such path exists the program backs up and begins again with another edge of the starting n-gon on the outer ring polygon. If such a path exists then another n-gon is added to the chain. The maximum length of the path is N-3. The vertex in the new n-gon adjacent to the

6. Which are found the same way as in the prism graph routine. That is, by taking the pair of vertices (which are not yet in the ring structure) which are adjacent to the last pair of vertices added to the ring polygons.

7. The case of adjacent vertices may be considered as an instance where the path has length 1.

to the last vertex in the outer ring polygon is added to the outer ring polygon.[8] The vertex in the path that is adjacent to this vertex is added to the inner ring polygon.

Whenever the path discussed above has length greater than one there will be one or more vertices on the path which are not added to either of the ring polygons. These vertices are placed in a special list, L, and are referred to as 'inner' vertices because they will be displayed inside the ring structure.

The ring structure is complete when the last n-gon of the chain passes through the first pair of vertices in the ring polygons. This is the only case when the n-gon added to the chain may contain vertices already in the ring polygons.

Once two ring polygons have been identified, the ring structure is displayed using the method for drawing concentric polygons. If all the polygons in the chain are 4-gons (graph is a prism graph), then the display is finished. Otherwise, the vertices in L are placed on the inner ring polygon. These vertices are spaced equidistantly along the portion of the inner ring polygon in which they occurred. For example, the ring polygons (1,2,9,7,5) and (4,3,10,8,6,) in Figure 3.7 would have been displayed first (including the edges joining the adjacent vertices of the ring polygons). Vertices 11,12 and 13 would then have been displayed in the middle of the lines originally connecting the vertex pairs 10 and 8, 6 and 4, and 3 and 10 respectively.

8. This vertex is now referred to as the last vertex added to the outer ring polygon.

The display is only partially finished at this point because the vertices in L have only two edges drawn incident to them and there may be vertices which are not yet in the display.

If any vertices in L are adjacent then these vertices are joined by straight lines in the display. All such adjacent pairs of vertices are removed from L. If L is now empty the representation is finished. The graph in Figure 3.7 does not contain any such pairs of adjacent 'inner' vertices.

Otherwise, L is examined for any pairs of vertices which are connected by a path of length 2, the middle vertex of the path being one which is not yet in the display. Straight lines are drawn joining all such vertex pairs. The middle vertex of each path is displayed at the mid-point of the corresponding line. The middle vertices are added to L while the vertices in L joined by the paths are removed from L. The vertices 11 and 12 in Figure 3.7 are joined by such a line with vertex 14 at the mid-point of the line.

If L was altered by the previous step it is necessary to go back to the step where L is checked for adjacent vertices. For the graph in Figure 3.7 this is necessary. At this point L = {14,13}. But vertex 13 is adjacent to vertex 14 so that the two vertices will be joined by a straight line and the representation is finished.

If L was not changed by the previous step,[10] then an edge is drawn incident to each vertex in L. These edges are drawn towards the centre

10. This means that none of the vertices in L are adjacent and none can be joined by a path of the type described above.

of the display. The length of the edge is under user control and depends upon the desired size of the map of the graph. The new end-vertex of each edge is added to L while the vertices previously in L are deleted from L. The program now loops back to the point where L is checked for adjacent vertices. The heuristic used by the representation routine to find a suitable ring structure and then display the graph is given below:

1) Let N ($N \geq 4$) be the maximum number of edges allowed in any polygon in the ring structure.[11] Let n = 4 and L = ∅.
2) Find a (another[12]) starting polygon with n edges. If the graph does not contain such an n-gon set n = n+1 and go to step 3. Otherwise go to step 4.
3) If n>N then set N = N+1.[13] Go to step 2.
4) Choose one of the edges of the starting n-gon to be an edge on the outer ring polygon.[14] If all the edges in the starting n-gon have already been tried, go to step 2.
5) Find a pair of vertices, u and w, which are not yet in the

11. The user specifies a value for N before the routine begins.

12. One which has not yet been tried.

13. The user is asked if he wishes N incremented. If not the routine halts without finding a representation of the graph.

14. The initial choice is quite arbitrary and is made in the same manner as for prism graphs.

ring structure[15] such that u is adjacent to the last vertex added to the outer ring polygon and w is adjacent to the last vertex added to the inner ring polygon. If no such vertices u and w exist then go to step 4.

6) If u is adjacent to w then add u and w to the outer and inner ring polygons respectively and go to step 5. Otherwise, go to step 7.

7) Find a path from w to u with length $\leq$ N-3 such that the path does not contain any vertices already used in the ring structure. If no such path exists then go to step 4, otherwise go to step 8.

8) Add vertex w to L. Also add to L all the other vertices in the path except vertex u and that vertex (in the path) which is adjacent to u. Let w now denote the vertex in the path which is adjacent to u.

9) Add u and w to the outer and inner ring polygons respectively. Go to step 5.

10) Draw the ring structure using the display method for concentric polygons.

11) If L is empty the representation is finished. Otherwise, add all the vertices in L to the display.[16]

15. The only exception is when vertex u is the first vertex of the outer ring polygon. If vertex w is the first vertex of the inner ring polygon or there is a path from w to the first vertex of the inner ring polygon of length $\leq$ N-3 (path of the type described in step 7), then the ring structure is complete. Go to step 10 in this case; otherwise, go to step 4.

16. The vertices are spaced equidistantly along the portions of the inner ring polygon in which they occurred.

12) Join all adjacent pairs of vertices in L by straight line edges. Delete all such vertex pairs from L. If L is now empty the representation is finished.

13) For every pair of vertices in L for which there exists a path of length 2 between the vertices (the middle vertex of the path must not be in the display yet), draw a straight line joining the vertices. Display the middle vertex of each path at the mid-point of this straight line. Delete each such vertex pair from L, at the same time adding the middle vertex of each path to L. If no such vertex pairs existed then go to step 14, otherwise go to step 12.

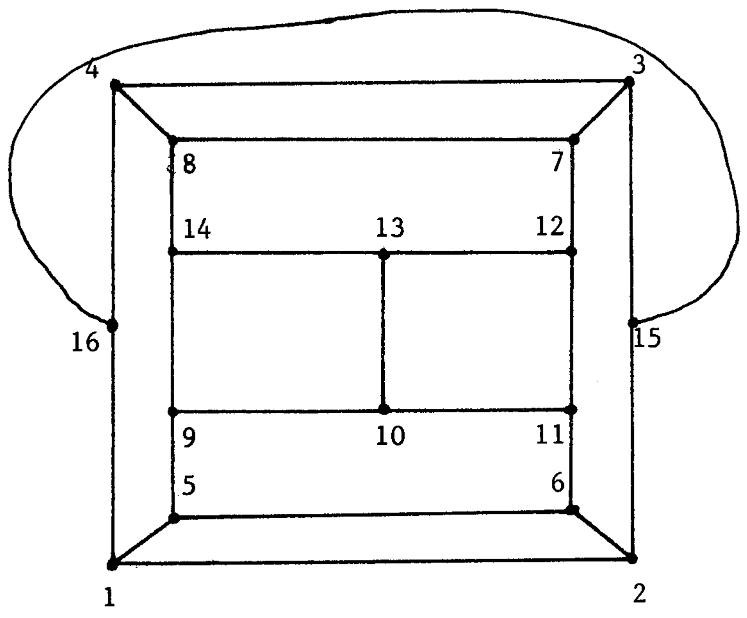
14) Draw a straight line from each vertex in L towards the centre of the display screen.[17] For each vertex, x, in L add to L the vertex, y, which has not yet been displayed and which is adjacent to x. Delete x from L. Display y as the end-point of the edge which was just drawn incident to x. Go to step 12.

Appendix B contains examples of some of the displays obtained using this representation routine. There is also a detailed flowchart of the representation routine. Just as Faulkner generated planar maps for the class C, the representation routine creates planar representations. Once the ring structure has been displayed steps 12 to 14 add vertices and edges to the display in the form of polygons with no intersecting edges.

17. The length of these lines is specified by the user before the representation routine begins.

In order to illustrate how the representations in Appendix B were created, the following example describes how the routine might create a representation of a 10 region graph.

Example:



The graph does not contain a chain of 4-gons so that if N were initially set equal 4 the user would eventually be asked to increment N as described in step 3 of the heuristic. In order to shorten the example the assumption has been made that N is initially set to 5.

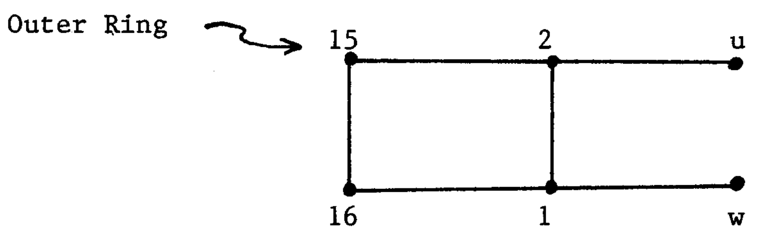
1) N = 5, n = 4 and L = ∅.

2) The starting 4-gon (1,2,15,16) is located.[18]

4) Let the edge joining vertex 15 to vertex 2 be the first edge in the outer ring polygon. Vertex 15 is then the first vertex in the outer ring polygon and vertex 16 is the first vertex in the inner ring polygon.

18. The order in which starting n-gons are tested depends upon the numbering of the vertices.

Vertices 2 and 1 become the last vertices in the outer and inner ring polygons as the 4-gon (1,2,15,16) becomes the first polygon in the ring structure.



5) u = vertex 6 and w = vertex 5.

6) Vertex 6 is added to the outer ring polygon and vertex 5 to the inner ring polygon as vertex 6 is adjacent to vertex 5.

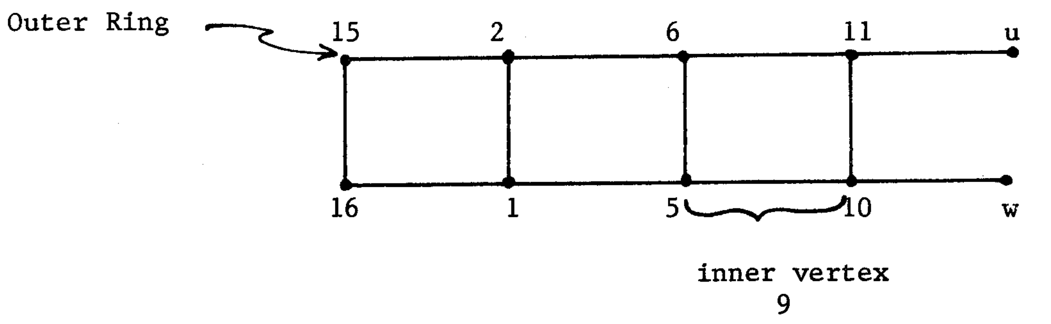
5) Vertices 6 and 5 are adjacent to vertices 11 and 9 respectively. Therefore, u = 11 and w = 9.

6) Vertex 11 is not adjacent to vertex 9.

7) The path (9,10,11) is the required path joining vertex 9 to vertex 11. The length is $\leq$ N-3 (N=5) and vertex 10 has not yet been used.

8) Add vertex 9 to L. w now denotes vertex 10, the vertex in the path (9,10,11) which is adjacent to vertex 11.

9) Vertex 11 is added to the outer ring polygon and vertex 10 to the inner ring polygon.



5) u = vertex 12 and w = vertex 13.

6) Vertex 12 is added to the outer ring polygon and vertex 13 to the inner ring polygon as vertex 12 is adjacent to vertex 13.

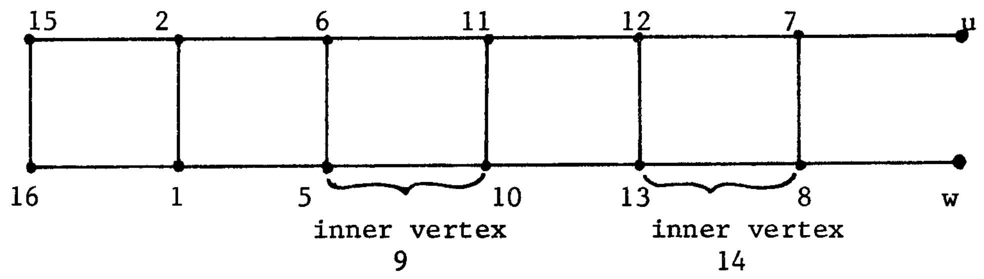
5) Vertices 12 and 13 are adjacent to vertices 7 and 14 respectively. Therefore, u = 7 and w = 14.

6) Vertex 7 is not adjacent to vertex 14.

7) The path (14,8,7) is the required path. The length is $\leq$ N-3 (N=5) and vertex 8 has not yet been used.

8) Add vertex 14 to L. w now denotes vertex 8, the vertex in the path (14,8,7) which is adjacent to vertex 7.

9) Vertex 7 is added to the outer ring polygon and vertex 8 to the inner ring polygon.

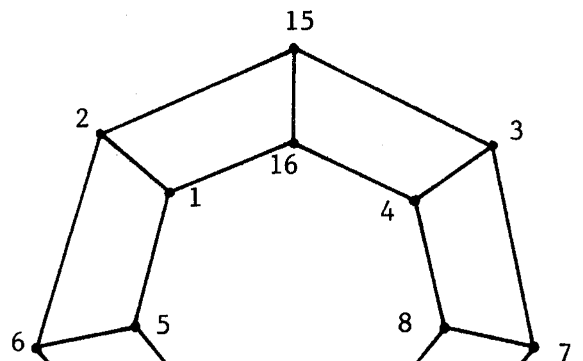


5) u = vertex 3 and w = vertex 4.

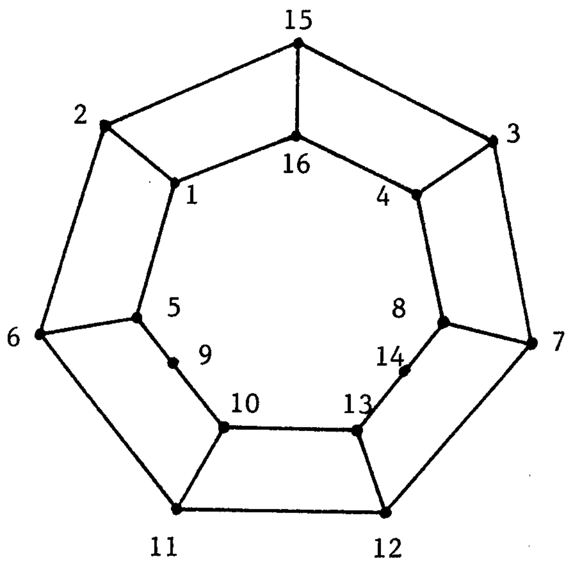
6) Add vertex 3 to the outer ring polygon and vertex 4 to the inner ring polygon.

5) There is no vertex adjacent to vertex 3 which is not in the ring structure. However, vertex 3 is adjacent to vertex 15, the first vertex in the outer ring polygon. Thus u is set to vertex 15. Similarly, there is no vertex adjacent to vertex 4 which is not already in the ring structure. However, vertex 4 is adjacent to vertex 16, the first vertex in the inner ring polygon. Therefore, the ring structure is complete.

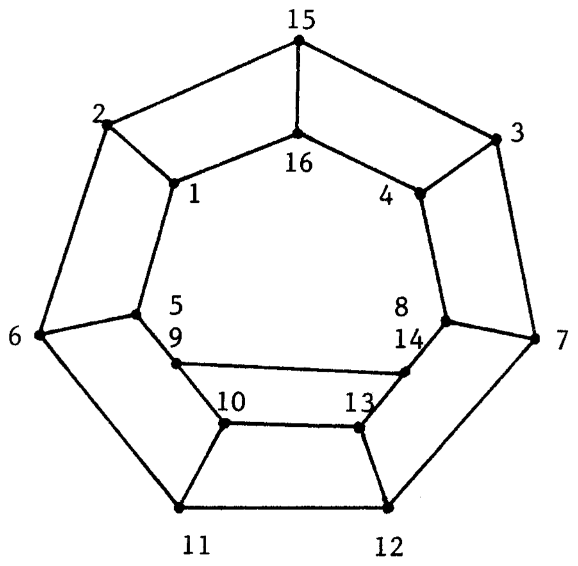
10) The ring structure is displayed as shown below using the concentric polygon display method.



11) The inner vertices 9 and 14 in L are added to the display.



12) A straight line representing the edge (9,14) is drawn joining the adjacent vertices 9 and 14. Vertices 9 and 14 are deleted from L. Since L is now empty the representation routine is finished.



The representation routine for prism graphs is an algorithm. Given a prism graph it generates a ring structured display of the graph. However, the more general representation routine for class C graphs is a heuristic.

Given a class C graph it may not succeed in creating a visual representation of the graph. It fails to do so whenever it is unable to find the required ring structure. In step 7 when attempting to extend the polygon chain only one polygon is considered even if several suitable polygons exist.[19] Should the routine then fail at a later point it does not back up to test the other possibilities.

It would be quite easy to have the routine back up and try all the possibilities. However, the routine already requires a relatively large amount of execution time. In fact several features which are described below were added to the routine in an attempt to reduce the amount of execution time it consumes. Checking all the possible paths would only use up even more execution time.

Moreover, there is no proof that the ring structure used in the representations must exist in every class C graph. Even if all the possibilities were tried the routines might still not find a pair of ring polygons. However, by the time the representation routine had been implemented it was clear that the routine was using too much execution time to be satisfactory and proving that a ring structure existed for every class C graph would not eliminate this problem. The reasons for abandoning this representation routine in order to investigate automorphism group based routines are discussed later in the chapter.

19. That is, there may be more than one path from w to u of the type required.

Several features were used in the final implementation of the representation routine in order to reduce the amount of execution time required to produce a map. The user is able to set the maximum polygon size to be used, preventing the routine from trying all the possible polygon sizes up to and including the largest polygon size in the graph. In addition, the user may set an extension limit for edges. If this is greater than 0 then the routine can try to form a chain of n-gons, $n \geq 4$, immediately rather than attempt to find a chain of 4-gons first. Only occasionally will the graph actually be a prism graph. The user may also specify the starting polygon for the ring structure. This does not mean that if the routine fails with the specified polygon it might not have succeeded with another. Rather, it prevents the program from running away with execution time trying all the different possible starting polygons. Finally, the user may indicate that there may be an axis of symmetry (mirror-image symmetry) through the starting polygon he has specified. This forces the routine to consider only one direction through the starting polygon. Notice that if the routine does succeed in finding a visual representation using the specified starting polygon and axis of symmetry it does not follow that the graph is actually symmetric about the given axis. This is evident from the map of the graph shown in Figure 3.7.

It is rather difficult to attempt to evaluate this representation scheme. The visual representations it does produce are quite interesting and pleasing. As can be seen from Appendix B it is also very successful in finding a representation for a graph, although it may require several repetitions of the same graph with different user specified starting polygons to do so. In fact, a class C graph has not yet been found for which the routine would not eventually produce a representation. The representations are also planar and stress the polygonal nature of this class of graphs very effectively.

The heuristic is also capable of creating visual representations of 'nearly symmetric' graphs. In other words, the heuristic may produce an almost symmetric display of a graph which has only trivial automorphisms. The visual representation routines discussed in Chapter V require at least one non-trivial automorphism before they will create a display.

Unfortunately, the heuristic is equally capable of creating a nearly symmetric display when there actually exists a symmetric visual representation which it did not find. The routine is also expensive in terms of the execution time required to check all the possible combinations of starting polygons and ring structure polygons which might be used in a map. This follows from the fact that the routine does not make use of any information which could be obtained regarding the structure of the graph. The user has very little control over the operation of the program. Once it begins execution he cannot even suggest which paths seem to offer the greatest chance of success and which are likely to prove fruitless. The result is a very simplistic trial and error type of heuristic. Finally, although the ring structure display usually proves quite satisfactory, the routine is overly rigid in that the user cannot alter the final visual representation.

While working with this routine it became apparent that there were several improvements which were necessary. These were changes needed not so much to reduce the execution time used by the heuristic as to provide a better graphics tool for implementing the visual display segments of the program. It was obvious that there had to be more interaction between the user and the representation process. This would allow the user to direct the operation of the routine but, more important, would make it easier to test different algorithms for creating representations. In connection with the representations the graphics software must allow the user to save the visual representations in a backup file from which they can be easily recovered at a later date. It should also be possible to manipulate the displays without reference to the representation routines, thereby permitting the user to insert his own improvements in any of the representations. It would also be very useful to be able to obtain copies of the representations without having to copy the display from the screen by hand. That is, it should be possible to use a plotter to create printed copies of the graphs shown on the screen.

As already mentioned the above heuristic fails to use, other than indirectly, any of the information readily available on the structure of the graph. As a result the visual representations are obtained in a rather haphazard fashion. It is impossible to tell how many representations might be obtained from a graph and there is no systematic way to catalogue and compare the representations of different graphs.

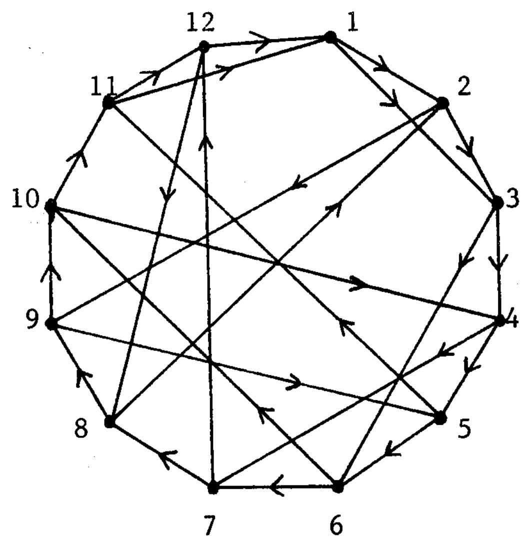
It was primarily this lack of a systematic approach that led to an investigation of the feasibility of using the automorphism group of a graph as a basis for generating its visual representation. The main obstacle, of course, was the fact that obtaining the automorphism group of the graph could be very expensive in terms of execution time. At the time however, the major concern was whether the automorphism group would be useful in creating the representation and not whether the cost of calculating the actual group would be prohibitive. The major consideration was how well the symmetries of a graph as represented by the mappings of the automorphism group would lend themselves to a visual portrayal. Should we be successful here we could then worry about the problems and costs of calculating the automorphism group.

Thus, the problem became one of interpreting the automorphism group mappings visually by transforming the symmetries present in the mappings into concrete display images. A large number of graphs were chosen at random and their automorphism groups calculated. The automorphism groups were then classified according to their group structure. That is, the automorphism groups were considered as abstract permutation groups. In many cases the automorphism groups were cyclic groups. Occasionally, one also encountered a dihedral group structure. This suggested that an attempt be made to create a visual representation based on the properties of such groups. For example, if an automorphism group was cyclic of order n one might try to create a visual representation showing a cycle or rotation of n vertices.

The most immediate objection to this approach is that one of the reasons for using the automorphism group was to be able to generate a different visual representation for each element in the group and the above method only permits one representation per group. Another objection is evident when one considers the rather well known fact that any two cyclic groups of the same order are isomorphic. But it is possible for non-isomorphic graphs to have isomorphic automorphism groups. Therefore, by rigidly applying this approach one could be forced to generate almost the same representation for two non-isomorphic graphs because they both had cyclic automorphism groups of order n. The two graphs in Figure 3.8 are clearly not isomorphic since they have different numbers of vertices; however, they both have automorphism groups which are cyclic of order 4. Thus, a representation routine using the above approach would try to generate the same representation for both graphs. This, of course, is quite absurd.

The last and most strenuous objection to the previously mentioned approach is that it does not really attempt to represent the symmetries of the graph at all. What happens is that the method actually trys to represent an abstract group structure. Only rarely will the structure of the automorphism group correspond to the structure defined by the symmetries of the graph. The graph shown in Figure 3.9 is one such case. The automorphism group is cyclic of order 5 while the automorphisms, considered as permutations of vertices, all have a single orbit containing all 5 vertices in the graph. It would be reasonable to represent these mappings as rotations of 5 elements, the same form the visual representation would take if the previous approach was used.

On the other hand, the graph in Figure 3.10 has an automorphism group which is cyclic of order 6 while the automorphisms themselves contain 9 vertices. The closest one can come to a cycle of 6 elements is with the two permutations (automorphisms) in the group which each



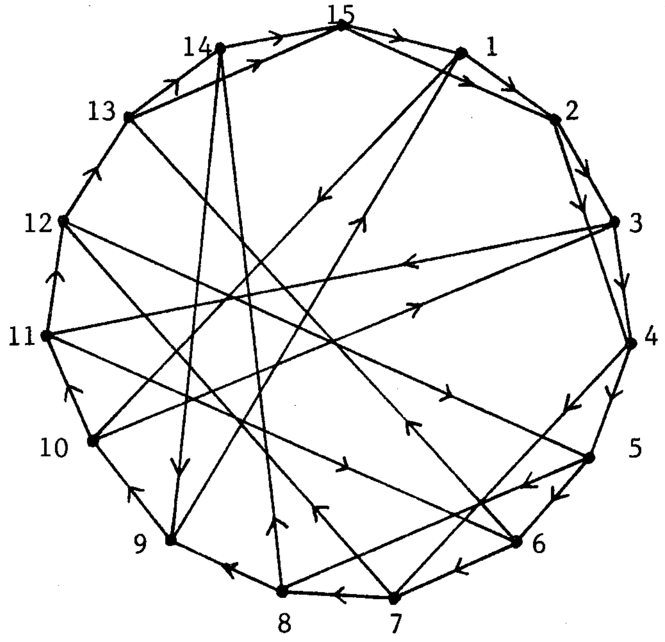
AUTOMORPHISM GROUP:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 1 | 2 | 3 | 6 | 10 | 4 | 7 | 8 | 9 | 5 | 11 | 12 |
| 8 | 2 | 9 | 10 | 4 | 5 | 11 | 1 | 3 | 6 | 7 | 12 |
| 8 | 2 | 9 | 5 | 6 | 10 | 11 | 1 | 3 | 4 | 7 | 12 |

AUTOMORPHISM 2 GENERATES SUBGROUP {1,2}. AUTOMORPHISMS 3 AND 4 GENERATE THE WHOLE GROUP

MULTIPLICATION TABLE FOR BOTH GRAPHS:

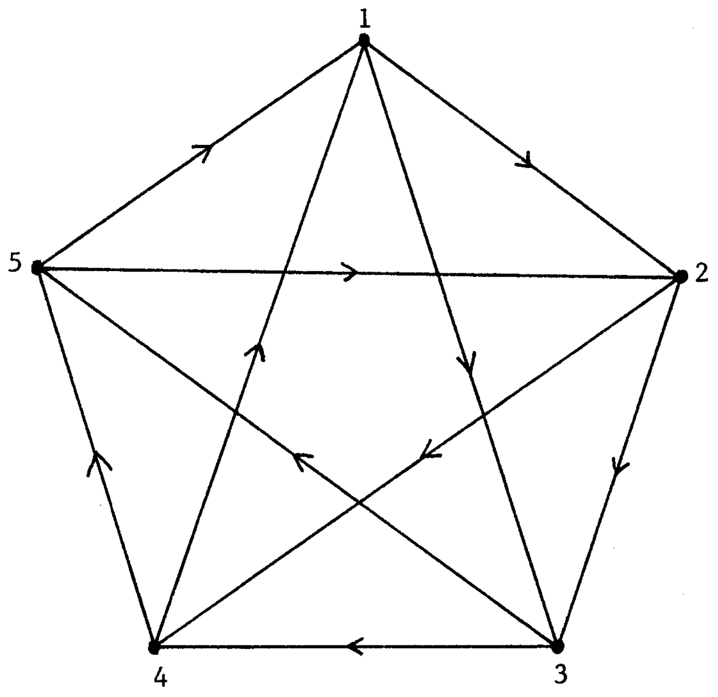
| ● | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 |
| 2 | 2 | 1 | 4 | 3 |
| 3 | 3 | 4 | 2 | 1 |
| 4 | 4 | 3 | 1 | 2 |



AUTOMORPHISM GROUP:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 2 | 3 | 4 | 7 | 12 | 5 | 8 | 9 | 10 | 11 | 6 | 13 | 14 | 15 |
| 1 | 10 | 3 | 11 | 12 | 5 | 6 | 13 | 15 | 2 | 4 | 7 | 8 | 14 | 9 |
| 1 | 10 | 3 | 11 | 6 | 7 | 12 | 13 | 15 | 2 | 4 | 5 | 8 | 14 | 9 |

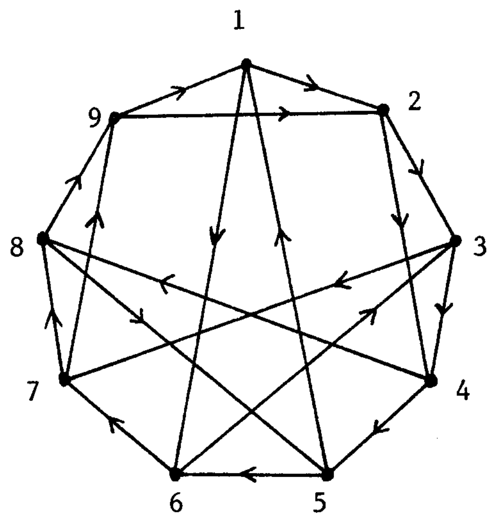
FIGURE 3.8



AUTOMORPHISM GROUP

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 1 |
| 3 | 4 | 5 | 1 | 2 |
| 4 | 5 | 1 | 2 | 3 |
| 5 | 1 | 2 | 3 | 4 |

FIGURE 3.9



MULTIPLICATION TABLE:

| ● | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 2 | 1 | 4 | 3 | 6 | 5 |
| 3 | 3 | 4 | 6 | 5 | 1 | 2 |
| 4 | 4 | 3 | 5 | 6 | 2 | 1 |
| 5 | 5 | 6 | 1 | 2 | 4 | 3 |
| 6 | 6 | 5 | 2 | 1 | 3 | 4 |

AUTOMORPHISM GROUP:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 6 | 3 | 7 | 9 | 2 | 4 | 8 | 5 |
| 3 | 4 | 8 | 5 | 6 | 7 | 9 | 1 | 2 |
| 3 | 7 | 8 | 9 | 2 | 4 | 5 | 1 | 6 |
| 8 | 9 | 1 | 2 | 4 | 5 | 6 | 3 | 7 |
| 8 | 5 | 1 | 6 | 7 | 9 | 2 | 3 | 4 |

ORBITS:

{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}

{1}, {2,6}, {3}, {4,7}, {5,9}, {8}

{1,3,8}, {2,4,5,6,7,9}

{1,3,8}, {2,7,5}, {4,9,6}

{1,8,3}, {2,9,7,6,5,4}

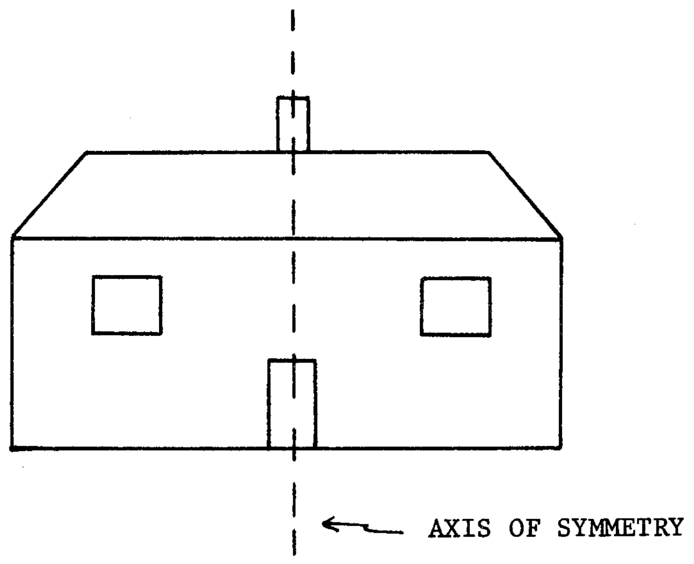
{1,8,3}, {2,5,7}, {4,6,9}

FIGURE 3.10

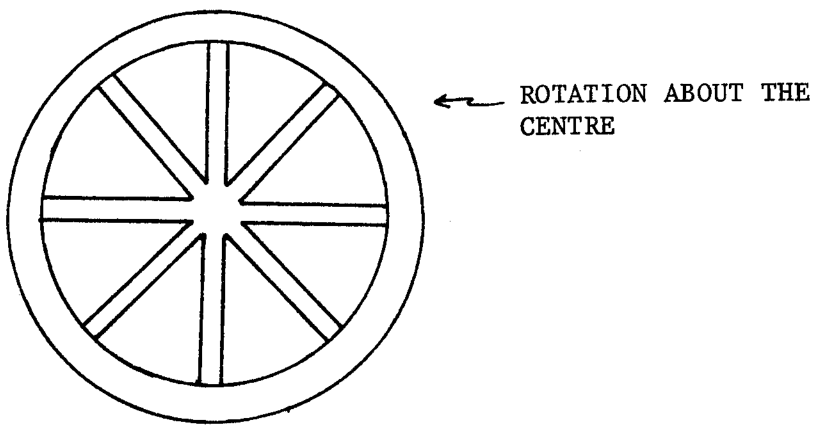
have an orbit with 6 elements. In each case there remains the problem of how to interpret the other orbit with 3 elements. Therefore, this approach is rejected in favour of one which takes into account the structure defined by the automorphism group mappings rather than the structure of the automorphism group itself.

With this goal in mind an investigation of the structure of the automorphism group mappings themselves was begun. These mappings are only permutations of the vertex set of the graph and may therefore be partitioned into orbits. An orbit containing a single vertex means that the vertex is fixed by the respective automorphism, whereas an orbit containing two vertices indicates that these vertices are interchanged by the automorphism. Lastly, an orbit containing more than two vertices represents a cyclic permutation of these vertices by the automorphism. These three types of orbits coincide with the two common forms of symmetry with which most people are familiar, mirror-image symmetry and rotational symmetry. Mirror-image symmetry or, as it is sometimes called, biplanar symmetry, consists of a reflection about an axis of symmetry. Rotational symmetry is a repetition of an image about a point, that is, a rotation of the image about a central point. The examples in Figure 3.11 should provide a better explanation of these two concepts than any detailed and verbose written description.

The point is that mirror-image symmetry in an object can be represented by a mapping which maps the points on the axis of symmetry onto themselves and interchanges those points on opposing sides of the axis. A rotation is simply a cyclic permutation of the same element



MIRROR-IMAGE SYMMETRY

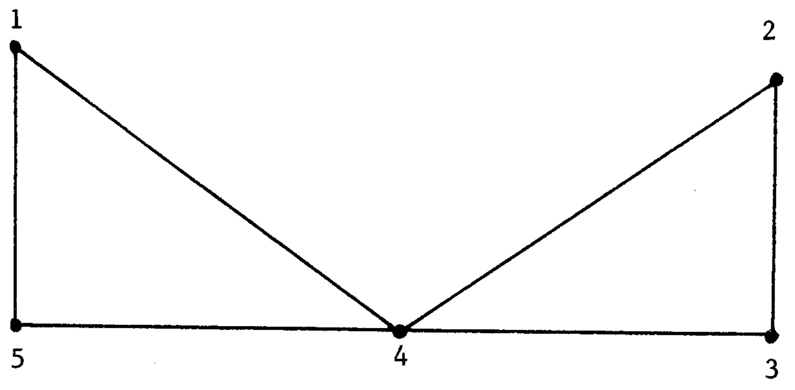


ROTATIONAL SYMMETRY

FIGURE 3.11

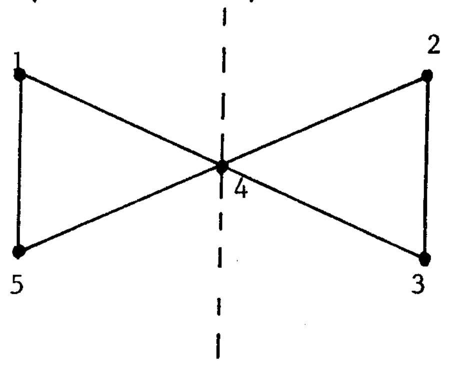
or portion of an object about a central point. For example, a rotation of a spoke and part of the rim produces the wheel in Figure 3.11. The conclusion is that any automorphism can be separated into mirror-image and rotational mappings of the vertices. This provides the link between the symmetries of a graph and a suitable visual image that was needed. Each mapping in the automorphism group defines a partition of the vertex set into three classes: fixed vertices, mirror-image vertex pairs and rotational vertices. The combination of fixed vertices and mirror-image pairs corresponds to any mirror-image symmetry in the graph while the rotational vertices correspond to the rotational symmetries. In Chapter V it is shown how the visual images corresponding to these symmetries may be used to create a representation illustrating the structure of the graph.

However, there are several peculiarities of the orbits of the automorphism mappings which will affect any visual representation scheme based on these orbits. The diagrams in Figures 3.12A and 3.12B show a simple yet typical example of the structure often found in automorphism mappings. The visual representation shown for each mapping is designed to illustrate the mirror-image and rotational symmetries in the respective automorphism. Notice that the representation for automorphism 3 is identical to that for automorphisms 4 and 5. The reason for this is that in automorphisms 4 and 5 only two vertices are 'moved' by the automorphism while the rest of the vertices are fixed. The result is that the visual representations are not unique, indicating that the automorphism group approach will not (directly) provide a unique visual representation for every mapping in the group. Moreover, for all but



| AUTOMORPHISM GROUP: | | | | | ORBITS: |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | {1}, {2}, {3}, {4}, {5} |
| 2 | 1 | 5 | 4 | 3 | {1,2}, {3,5}, {4} |
| 3 | 5 | 1 | 4 | 2 | {1,3}, {2,5}, {4} |
| 5 | 3 | 2 | 4 | 1 | {1,5}, {2,3}, {4} |
| 5 | 2 | 3 | 4 | 1 | {1,5}, {2}, {3}, {4} |
| 1 | 3 | 2 | 4 | 5 | {1}, {2,3}, {4}, {5} |
| 2 | 5 | 1 | 4 | 3 | {1,2,5,3}, {4} |
| 3 | 1 | 5 | 4 | 2 | {1,3,5,2}, {4} |

#1 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 5 & 4 & 3 \end{pmatrix}$



#2 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix}$

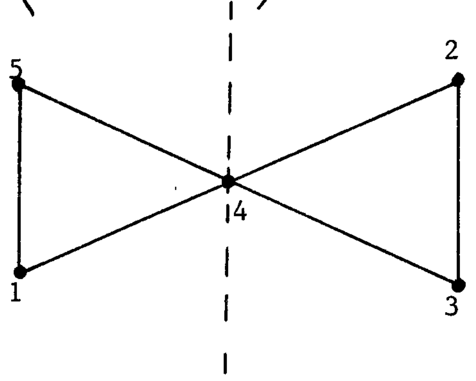
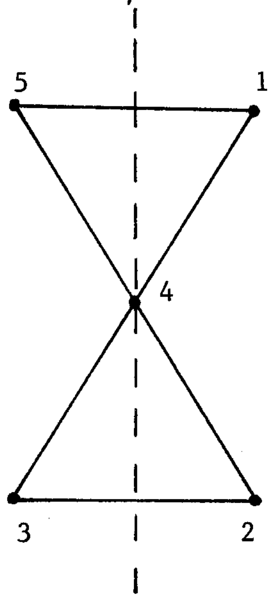
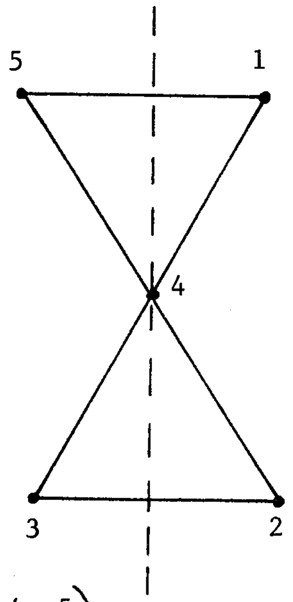


FIGURE 3.12A

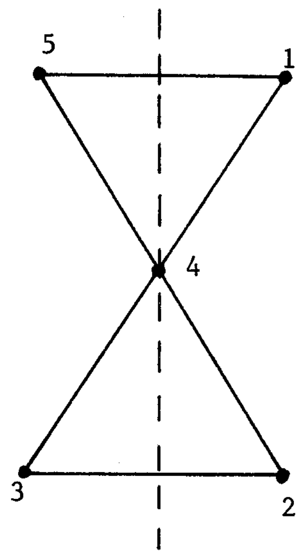
#3 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 2 & 4 & 1 \end{pmatrix}$



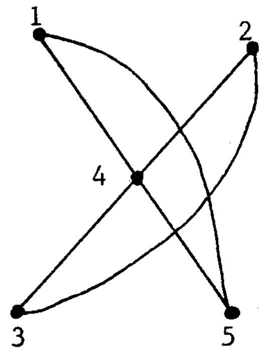
#4 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 2 & 3 & 4 & 1 \end{pmatrix}$

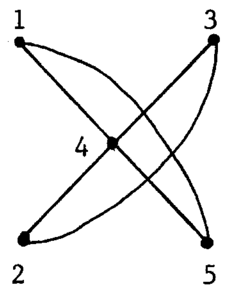


#5 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 2 & 4 & 5 \end{pmatrix}$



#6 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 4 & 3 \end{pmatrix}$





#7 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 4 & 2 \end{pmatrix}$

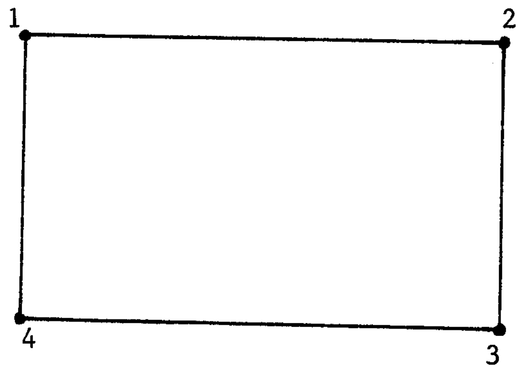
FIGURE 3.12B

the simplest graphs it is very difficult to determine when repetitive representations will be obtained from different automorphisms.

As indicated above a mapping of a mirror-image symmetry in a graph contains mirror-image vertex pairs and fixed vertices only. However, it does not follow that all the fixed vertices must be placed on the axis of symmetry in a visual display of this symmetry. Such is not the case for the above representations of automorphisms 4 and 5. In both instances the visual representation is effectively split into two parts at a vertex on the axis of symmetry. As shall be seen in Chapter V it often happens that a mapping of a mirror-image symmetry may be divisible (with respect to its visual representation) into sets of vertices which are mapped across the axis of symmetry and sets which, though they are fixed, are interchanged by another mapping. This fact results in the 'best' representations of mirror-image symmetry being obtained from automorphisms containing a comparatively large ratio of mirror-image vertex pairs to fixed vertices.

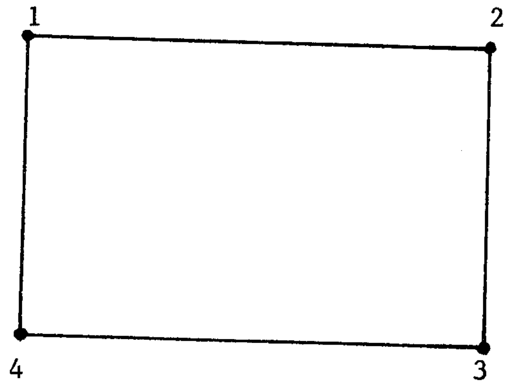
The representations for automorphisms 6 and 7 are clearly different. Yet, on closer examination it becomes evident that one can be obtained from the other by reversing the positions of either vertices 1 and 5 or vertices 2 and 3. This is a situation which occurs frequently with automorphisms containing rotations. There are often several automorphisms which have cyclic orbits (i.e. 2 or more elements) containing the same vertices. The only difference is in the order of the vertices as taken from the corresponding automorphism. As will be seen in Chapter V this situation usually leads to very similar representations being generated for these automorphisms. The problem may be even more pronounced in the case of automorphisms containing a mirror-image symmetry as well as one or more rotational symmetries. Moreover, identical or similar visual representations may result when rotational symmetries duplicate the effect of a mirror-image symmetry. For example, Figure 3.13 shows two identical representations derived from two different automorphisms, one composed of mirror-image pairs and fixed vertices and the other having a single cyclic orbit.

In conclusion then, the automorphism group approach satisfies most of the requirements for a useful basis for the creation of visual representations. While it may not be possible to obtain a unique representation for each automorphism it should be possible to generate a sufficient number of representations to reveal all the symmetries in the graph. Moreover, the automorphism group method satisfies the criterion that the approach must be systematic. By breaking the graphs down into their component mirror-image and rotational symmetries it is possible, for example, to examine graphs using a logical basis for comparison. Lastly, the concept of mirror-image and rotational symmetries seems quite natural and should prove satisfactory in most situations.



$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

ROTATIONAL SYMMETRY



$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$$

MIRROR-IMAGE SYMMETRY

FIGURE 3.13

# CHAPTER IV : AUTOMORPHISM GROUP CALCULATION

In this chapter the problem of calculating the automorphism group of a graph is considered. Of primary interest is a heuristic approach to automorphism group calculation and various heuristics used in finding the automorphism group will be discussed. The execution time requirements of such procedures will be discussed in some detail. The chapter will conclude with a discussion of an interactive approach to automorphism group calculation.

The problems involved in finding the automorphism group of a graph were first introduced in Chapter II where it was indicated that this was still essentially an unsolved problem. There is as yet no fast and efficient algorithm for finding the automorphism group. Nor do we claim to have found such an algorithm.

We are more interested in using the automorphism group to generate visual representations of a graph than with how the automorphism group can be obtained. For instance, were there an existing algorithm for calculating the automorphism group quickly, we would be quite content to use it. However, there is no such algorithm available. Yet we have indicated that the automorphism group is an integral part of our plan for generating visual representations of graphs. Moreover, it is necessary to be able to calculate the automorphism group while the user is waiting at the screen. The execution time required for the algorithm described in Chapter II would result in the user waiting for long periods of time between representations. This assumes of course that the user would even be willing to pay the large costs in execution time.

As Unger has discussed, when faced with the choice of a long investigation into the problem of obtaining a suitable algorithm (with no guarantee of success) or trying a heuristic approach, it may be more profitable to take the heuristic route.(xi) It was for this reason that the decision was made to investigate the possibilities of using heuristic routines for the calculation of the automorphism group.

It was mentioned in Chapter II that the heuristics used by Unger to tackle the graph isomorphism problem could also be applied to the automorphism group problem. Unger's program GIT attempts to reduce the number of possible isomorphism mappings between two graphs. The vertices of each graph are partitioned into sets of vertices having the same property or properties. The graph properties used by Unger to partition the vertex sets of the graphs were primarily the valence of the vertices, the number of vertices that could be reached from a vertex v by a path of length n and whether or not a vertex was included in a polygon of length n.

The plan is to use different graph properties to refine these partitions further and further until it becomes evident that the graphs are not isomorphic or until the number of possible mappings that must be checked is considerably reduced. If at some point in the process of refining the partition, there is created for one graph a partition for which there is no equivalent partition for the other graph, then the two graphs cannot be isomorphic. If this refinement process does not preclude the possibility of an isomorphism then GIT calls on a routine which tests for isomorphism all the possible mappings of the vertices according to the last vertex set partition. That is, vertices in each partition set of a graph can only be mapped onto vertices in the corresponding partition set of the other graph.

This concept of working with sets of vertices that can be mapped onto each other is very similar to the algorithm used by Kately (see Chapter II). Kately has used the valencies of the vertices to eliminate mappings which cannot be automorphisms. That is, he uses a property of the vertices of the graph to partition the vertex set into sets of vertices which must be mapped onto themselves by any automorphism. But we can clearly continue in this direction by using any property of the vertices of the graph which is invariant under automorphism to further refine the vertex set partition. In particular, the properties mentioned above that were used by Unger for the isomorphism problem can be used for this purpose. Again, the aim is to refine the partition as far as possible. Unlike the GIT program, however, there is no question of searching for a contradiction between the resulting vertex sets. Each set in the partition always represents a set of vertices which might be mapped onto themselves by an automorphism. It is possible that the partition may be refined to the point where each member of the partition contains only one element. This, of course, indicates that the automorphism group contains only one automorphism, the trivial identity mapping. In any other case, the sets of vertices in the partition can be used to check all the possible mappings of these vertices to determine which mappings are actually automorphisms.

Checking for automorphisms is simply a matter of applying an altered version of the algorithm of W. Jackson that was described in

Chapter II. As seen there, Jackson's routine tested the possible mappings of each vertex using the valence of the vertex as well as the fact that with an automorphism the successors of a vertex have to map onto the successors of the image of the vertex. But there is now more information on the possible mappings of the vertices. Each set in the partition is the set of possible images of each vertex in that particular set. Thus, an altered version of Jackson's routine has been written which uses only these 'image sets' in generating the automorphisms.

The problem was to determine which properties of the vertices of a graph would be most useful in calculating the image sets. Each suitable property could then be incorporated into a heuristic routine which would attempt to further refine the vertex set partition. It is important to remember that these routines are heuristics because it is impossible to tell beforehand whether or not they will succeed in further refining the partition. Moreover, even if they succeed in generating a refinement of the partition it is impossible to tell if this new partition can be further refined. The goal of all these heuristics is to obtain the automorphism partitioning of the vertex set. Unfortunately, it is impossible to tell whether this goal has been reached without actually generating the automorphism group.

There are several reasons why the algorithm of Corneil and Gotlieb (xxxiv) is not used to generate the vertex set partition. While it appears that their algorithm actually does find the automorphism partitioning, the fact remains that this is still a conjecture which has not yet been proven. Moreover, the algorithm is not a general algorithm in that it is not applicable to all classes of graphs. This in itself was sufficient reason not to incorporate the algorithm into what was intended to be a generalized graph representation system. Moreover, the heuristic tests would still have to exist in order to allow the user to work with graphs the Corneil and Gotlieb algorithm could not handle. The task of determining whether a graph is of the type that could be partitioned by the Corneil and Gotlieb algorithm is also a non-trivial process. This process would have to be repeated for every graph.[1]

Had the heuristic approach to calculating the automorphism groups failed, that is, had it proved impractical because of the execution time required to use heuristic routines to calculate automorphisms of every graph to be displayed, then it might have been necessary to use the Corneil and Gotlieb algorithm. However, as discussed later in this chapter, the use of interactive heuristics for partitioning the vertex set has made the calculation of automorphisms of a graph computationally feasible. The symmetries indicated by an automorphism can then be used to generate a picture of the graph.

It has been mentioned that the heuristics used by Unger may be used in a routine for calculating the automorphism group. There are many other properties of a graph related to its structure such as the cutsets of the graph, the blocks of a graph and so on which might be used as the basis for other such heuristics. In deciding

1. This would be unnecessary if the user knew the graph could be partitioned by the algorithm.

which heuristics to finally implement a number of considerations determined our choice. Primarily we wished to use heuristics which required very little execution time since the user is sitting in front of the display screen impatiently waiting for a picture of the graph. However, the same heuristics must also be useful in the sense that they frequently find a refinement of the vertex partition. For example, there is little point in implementing a routine which, though it may execute very quickly, is applicable only to an exotic class of graphs which occur infrequently. Finally, there is the question of adapting the choice of heuristics to the type of graph being investigated. This implies that it would be worthwhile when working with a special class of graphs to include tests based upon the special properties found in the class. However, this raises the question of how large a variety of tests are necessary. For instance, heuristics based on similar or related properties of a graph will tend to be redundant, in which case there is little point in having both tests available.

In light of the above criteria, those heuristics which have been included in a GSYM user routine for calculating the automorphism group of a graph will now be examined. This consists of a collection of heuristic tests which attempt to reduce the number of mappings which might be automorphisms. The user indicates which tests are to be tried and the values of any required or necessary parameters. Once all the specified tests have been tried, a routine for generating the automorphism group from the vertex set partition obtained by the heuristics is invoked. This routine will calculate the automorphism group regardless of how successful the heuristics were in partitioning the vertex set. However, the execution time required to compute the automorphism group depends on the degree of success of the heuristics.

Before discussing the various heuristics it should be noted that these heuristics and also the visual representation routines discussed in Chapter V assume the graph to be a simple directed or undirected graph. The graph may not contain a mixture of both directed and undirected edges. The restriction on loops and multiple edges is made only to simplify the routines used. These situations could easily be treated as special cases if it became necessary or desirable. The following is a discussion of the heuristics which have been implemented on the GSYM system for finding the automorphism group of a graph.

## Valence Test

The user must always invoke this test before any others unless the graph is regular. The vertex set is partitioned according to the valencies of the vertices. A vertex can only be mapped by an automorphism onto a vertex having the same valence. If the graph is directed then the vertex set is partitioned using both the invalence and outvalence of the vertices. This routine is the simplest and the fastest of all the heuristics as the required valence information is directly available from the GSYM data structures for representing graphs. It is also very useful as the majority of the graphs treated will not be regular.

## Path Test

This test attempts to refine the vertex set partition by calculating the number of vertices that can be reached from each vertex by a path of length N. The user must specify the value for N. This routine has been written so that it creates a vector of N elements for each vertex, the elements of the vector representing the number of vertices that can be reached by paths of length 1, 2,...N. The vertex set partition is then refined on the basis of the complete vector, not just the value of the last element. Vertices with identical vectors are placed in the same set.

The paths need not be simple and may pass through the same vertex more than once; they are, however, prevented from passing through their origin a second time. A similar heuristic was tested which used only simple paths. It was found that in generating the paths it was simpler and faster to allow repeated vertices. Since the two heuristics appeared to be equally successful in refining vertex set partitions, the simple path heuristic was removed from the list of heuristic tests. If the graph is directed the test is first performed using the positive paths beginning at each vertex. The test is then repeated using the negative paths beginning at each vertex.

## Polygon Test

The polygon test determines the number of polygons of length N through each vertex. This is an improvement on the test suggested by Unger which only determined if a vertex was in a polygon of length N. While this heuristic requires more execution time than that used by Unger, it has been found to be considerably more effective. A further motivation was the need to have a heuristic that is suitable for treating graphs with a polygonal type of structure. The user specifies the maximum length N to be used in searching for polygons through each vertex. As in the path test, a vector is created for each vertex representing the number of polygons of length 3, 4,...N passing through the vertex. Again, in order to simplify the procedure and to reduce the amount of execution time required, the polygons need not be simple. The heuristic has been implemented as a simple recursive routine which creates a tree structure representing all the paths starting at each vertex. As each level is added to the tree it is checked to see if a polygon through the root vertex has been found. Obviously, it is sufficient to use positive polygons when the graph is directed as repeating the test using negative polygons would only find the same polygons by going in the opposite direction.

## Fixed Point Test

As a result of applying the various heuristics it may come about that there exist sets in the partition that contain only one vertex. Such vertices are 'fixed vertices', that is, every automorphism of the graph must map such vertices onto themselves. For example, if a graph contains only one vertex with valence 4, then this vertex would be 'fixed' by every automorphism. Moreover, this fact would be evident after the valence test had been executed.

Such vertices are important because every neighbourhood of a fixed vertex must be mapped onto itself. Thus, the fixed point heuristic when invoked identifies these neighbourhoods and uses them to refine the vertex set partition. The user must identify the fixed vertex and specify a value N, the neighbourhoods 1, 2,...N being used by the test. It is also possible to specify that the test be tried for all fixed vertices in the graph. The reason for this is that this test tends to isolate other fixed vertices (when they exist) which in turn can be used to identify others. For directed graphs both positive and negative directions are used to determine the neighbourhoods.

This heuristic, while it appears very simple and is, in fact, quite inexpensive in terms of execution time requirements, has proven extremely suitable for graphs containing one or more fixed vertices.

## Tree Centre Test

This heuristic was especially written to be used with graphs which are known to be trees. It was included in the collection because it has proved ideally suited for refining the vertex set partition of a tree. Every tree has a unique vertex or pair of vertices called the centre or bicentre of the tree. The centre of a tree is mapped onto itself by every automorphism of the tree, that is, it is a fixed vertex. A bicentre pair must also be mapped onto itself by every automorphism in that every automorphism either fixes these vertices or exchanges them.

The algorithm for finding the centre or bicentre of a tree is very simple and also very fast. All vertices of valence 1 are deleted from the graph. This process is then repeated until the graph contains only one vertex, the centre, or a pair of adjacent vertices, the bicentre pair. If the graph is directed then it is treated as an undirected graph in order to find the centre or bicentre. Although this algorithm is clearly not going to require a lot of execution time, the amount it does require may seem rather excessive just to distinguish one or two vertices from the rest of the graph. However, this vertex or pair of vertices may then be used to great advantage in refining the vertex partition. A centre vertex is a fixed vertex so the fixed point test is automatically invoked for the centre vertex. Almost as much information can be gained from a bicentre pair by using the fact that all vertices at distance N from one or other of the bicentre vertices must be mapped onto each other. Thus, the neighbourhood concept is applied to a pair of vertices rather than to one vertex as is the case in the fixed point test.

The user specifies the parameter N to be used in the neighbourhood test for bicentres or to be passed to the fixed point test for centres. If the graph is directed both positive and negative distances from the centre or bicentre are utilized. As mentioned this routine is extremely useful when working with trees. It is a case of implementing a special purpose heuristic especially suited to a particular class of graphs and it is not intended to be generally applicable. The user must know that the graph is a tree since the routine does not check this fact. Naturally it would be possible to test if the graph were a tree and inform the user. However, this is contrary to the idea that the routine is reserved for users working with trees.

Trees are a very interesting class of graphs that are frequently used in applications of graph theory and it was felt that this warranted the inclusion of a heuristic that facilitates the handling of such graphs. Checking every graph passing through the system to see if it happens to be a tree would prove too costly to be practical.

## Undirected Version of Tests

It is possible for the user to invoke undirected versions of the path test and the polygon test. In both cases an undirected version of the current graph is created by removing all edge directions from the graph. The path or polygon test is then applied to this new undirected graph. The undirected path test and the undirected polygon test are included because the results obtained from these tests may differ from the results obtained when the original tests are applied to the digraph. Moreover, removing the edge directions does not affect the structure of the graph so that any further refinement of the vertex set partition obtained in this manner is still valid.

Figure 4.1A illustrates a digraph which yields the following results for the regular path test.

Path Length

| Vertex | +1 | +2 | -1 | -2 |
|---|---|---|---|---|
| 1 | 2 | 0 | 0 | 0 |
| 2 | 0 | 0 | 2 | 0 |
| 3 | 2 | 0 | 0 | 0 |
| 4 | 0 | 0 | 2 | 0 |
| 5 | 2 | 0 | 0 | 0 |
| 6 | 0 | 0 | 2 | 0 |
| 7 | 1 | 0 | 0 | 0 |
| 8 | 0 | 0 | 2 | 0 |
| 9 | 2 | 0 | 0 | 0 |
| 10 | 0 | 0 | 2 | 0 |
| 11 | 2 | 0 | 0 | 0 |
| 12 | 0 | 0 | 3 | 0 |
| 13 | 1 | 0 | 0 | 0 |
| 14 | 1 | 0 | 0 | 0 |

These results give the vertex set partition:

(1,3,5,9,11), (2,4,6,8,10), (7,13,14), (12)

There are no vertices which can be reached from any vertex by a positive or negative path of length 2. Thus the above partition has been refined as far as is possible.with the regular path test. However, if the undirected path test is then applied for paths of maximum length 2 the following results are obtained:

Undirected Path Length

| Vertex | 1 | 2 |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 2 | 2 |
| 3 | 2 | 2 |
| 4 | 2 | 2 |
| 5 | 2 | 2 |
| 6 | 2 | 1 |
| 7 | 1 | 1 |
| 8 | 2 | 2 |
| 9 | 2 | 2 |
| 10 | 2 | 2 |
| 11 | 2 | 3 |
| 12 | 3 | 1 |
| 13 | 1 | 2 |
| 14 | 1 | 2 |

Applying these results to the above partition gives a further refinement of the vertex set partition:

(1,3,5,9), (2,4,8,10), (13,14), (6), (11), (7), (12)

Moreover, by continuing with the undirected path test one would eventually find that all the vertices in the graph are fixed except vertices 13 and 14.

Similarly, Figure 4.1B shows a graph whose vertex set partition cannot be refined by using the polygon test since there are no positive polygons through any of the vertices. However, the undirected polygon test will find a polygon of length 3 through vertices 1, 5

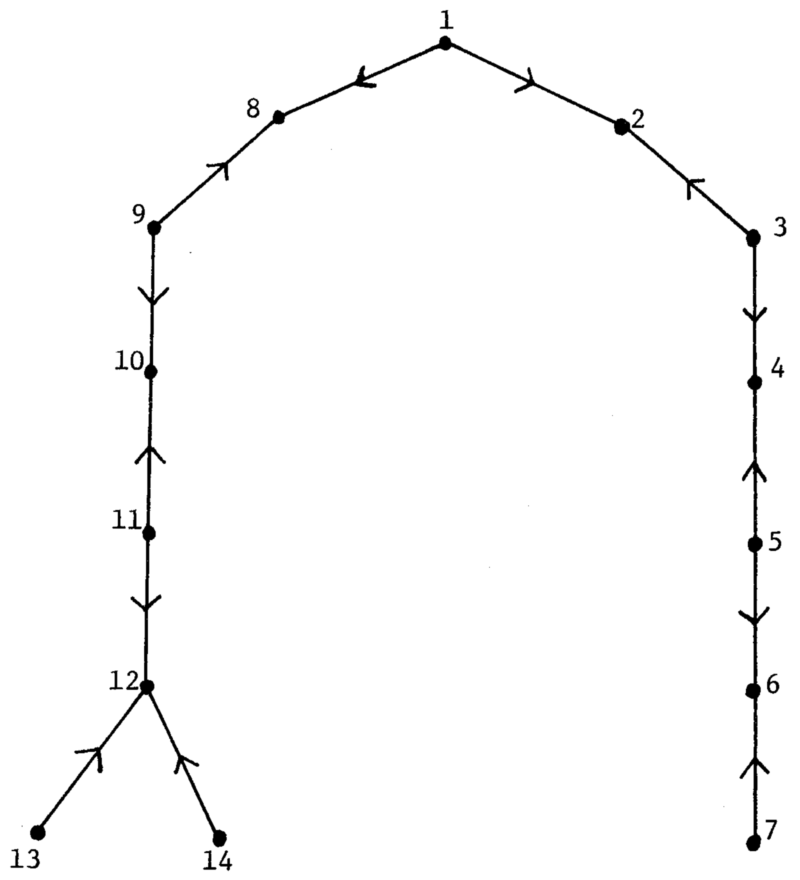


FIGURE 4.1A

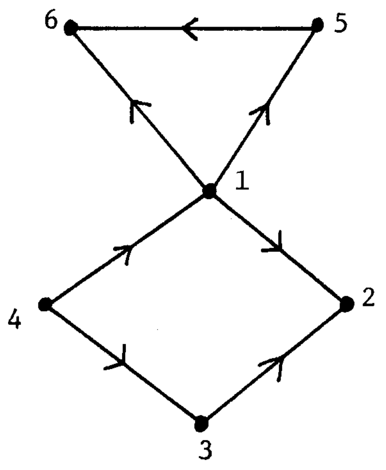


FIGURE 4.1B

and 6 and a polygon of length 4 through vertices 1, 2, 3 and 4. This, in turn, gives the following vertex set partition:

(1), (5,6), (2,3,4)

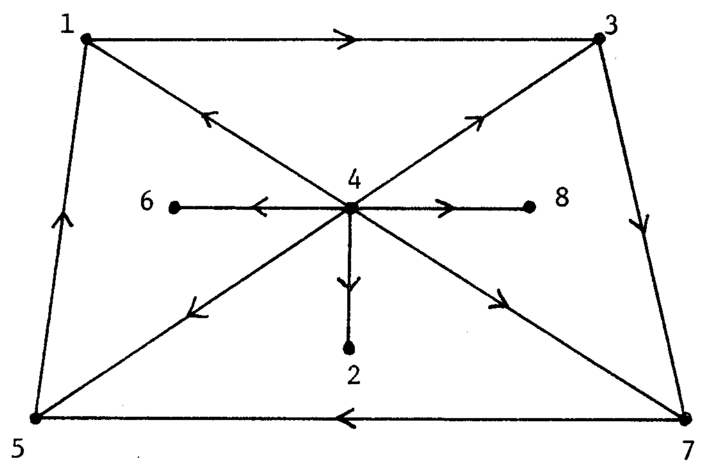
Thus, it is evident that the undirected path test and the undirected polygon test may be useful either by themselves or in conjunction with the regular path or polygon test.

There is one other routine which the user may invoke before entering the actual automorphism group calculation algorithm. Unlike the above routines it is not a heuristic test, nor is it designed to refine the vertex set partition. However, it is described here because it too is intended to reduce the execution time required by the automorphism group algorithm.

After using the various heuristics the sets obtained from the vertex partition will be different sizes. This routine relabels the vertices of the graph so that the largest sets are moved to the bottom of the list of sets. Thus, the new vertex 1 (which may be the same as the old vertex 1) now is in the smallest set while vertex n ($|VG|=n$) is in the largest set. The following example illustrates how this relabelling is done.

Example 4.1:

The graph below would result in the following partition after the valence test had been applied.



| Vertex | Object Set |
|---|---|
| 1 | (1,3,5,7) |
| 2 | (2,6,8) |
| 3 | (3,1,5,7) |
| 4 | (4) |
| 5 | (5,1,3,7) |
| 6 | (6,2,8) |
| 7 | (7,1,3,5) |
| 8 | (8,2,6) |

The above partition is changed by relabelling the vertices into the following partition:

| Vertex | Object Set |
|---|---|
| 1 | (1) |
| 2 | (2,3,4) |
| 3 | (3,2,4) |
| 4 | (4,2,3) |
| 5 | (5,6,7,8) |
| 6 | (6,5,7,8) |

| Vertex | Object Set |
|---|---|
| 7 | (7,5,6,8) |
| 8 | (8,5,6,7) |

The relabelling is given by:

| Old Vertex Label | New Vertex Label |
|---|---|
| 1 | 5 |
| 2 | 2 |
| 3 | 6 |
| 4 | 1 |
| 5 | 7 |
| 6 | 3 |
| 7 | 8 |
| 8 | 4 |

The routine which calculates the automorphism group is, as previously mentioned, very similar to Jackson's. In the course of analysing this routine it became evident that reordering the sets by relabelling the vertices could, in many cases, save a significant amount of execution time. The same type of tree-like structure described in Chapter II is created by the routine. The root of this tree is vertex 1. The vertex labelling used by the automorphism group calculation algorithm is independent of the vertex labelling in the GSYM system. Originally the labels are ordered from 1 to n ($|VG|=n$) according to the order in which the vertices were created. The problem is to relabel the vertices from 1 to n so that the smaller the label the smaller the set in which the vertex is found.

The logic for doing this is based on the knowledge that the algorithm will, in almost all cases, discover branches of the tree that do not represent automorphisms. As described in Chapter II, if a contradiction is found while a branch is being created the remainder of the branch representing the permutation being tested need not be generated. Therefore, if it is discovered that the current vertex permutation is not an automorphism before too much of the permutation has been generated, a significant amount of execution time can be eliminated.

For example, Figure 4.2 shows part of the tree structure that would be generated by the automorphism group routine using the original vertex labelling and partition of the graph in Example 4.1. Figure 4.3 shows the tree structure that would be generated for the same graph using the second labelling and reordered list of sets from Example 4.1.

The notation used in Figures 4.2 and 4.3 is the same as that found in Chapter II. The structure in Figure 4.2 begins, as usual, by examining all the possible mappings of vertex 1, in this case $1\to1$, $1\to3$, $1\to5$ and $1\to7$ as the object set for vertex 1 is (1, 3, 5, 7). Mappings implied by other mappings are shown in both figures. Thus, in Figure 4.2, $1\to1 \Rightarrow 3\to3$, $3\to3 \Rightarrow 7\to7$, $7\to7 \Rightarrow 5\to5$ and $5\to5 \Rightarrow 1\to1$. All these implied mappings must be checked before proceeding to the second level of the tree. This is again true when proceeding down the branches which begin with $1\to3$, $1\to5$ and $1\to7$.

Once the second level of the tree is reached the mappings $2\to2$, $2\to6$ and $2\to8$ must be examined[2] as vertex 2 has the object set

2. This must be repeated for each of the subtrees beginning with $1\to1$, $1\to3$, $1\to5$ and $1\to7$.

(2, 6, 8). Thus, in this labelling of the graph there are many mappings that must be examined in the first few levels of the tree structure. However, for the second labelling of the graph there is only one possible mapping at the first level, $1 \rightarrow 1$, as vertex 1 has the object set (1). Vertex 2 has only three possible objects 2, 3 and 4 so there are only three subtrees starting as the second level. Figuratively speaking, the second tree structure is longer and narrower than the "bushy" tree structure of Figure 4.2 where many subtrees begin in the first few levels.

The mapping checks[3] in Figure 4.3 tend to be near the bottom of the tree. This postponement of possible vertex mappings and their examination is the result of reordering the sets so that the smaller the vertex label the smaller the object set of the vertex. As explained in Chapter II, the automorphism group algorithm considers the possible vertex mappings beginning with vertex 1, then vertex 2 and so on up to vertex n. With the new ordering of the sets the upper levels of the generated tree structure will tend to have fewer branches[4]. The lower levels of the tree structure will tend to have correspondingly more branches.

However, it is possible and, in many cases, quite likely

3. See Chapter II for a discussion of the tests that are necessary to determine whether a vertex permutation is an automorphism.

4. There are fewer possible mappings at this stage because the object sets are smaller.

that these lower branches will never be generated as contradictions in the upper levels of the tree structure cause branches to be abandoned. The savings in execution time, which can be considerable, are a result of the tendency of a single contradiction to eliminate more mapping tests in the second type of tree structure than in the first type of structure.

If the partition created by the heuristics actually represents the automorphism partitioning of the graph, however, there will be no reduction in execution time. In this case, all of the possible mappings given by the sets are in automorphisms and the whole tree structure will be generated.

The above collection of heuristics does not, of course, begin to exhaust the list of possible heuristics that might have been used. The set of heuristics that has been implemented has been kept quite small partially in order to limit core requirements, but mainly because the present set has proved very successful in actual use. There have been very few cases when these routines did not succeed or come quite close to obtaining the automorphism partitioning of the vertex set. However, any property of a graph that is preserved under automorphism could conceivably be used as the basis for a heuristic test.

For instance, one such heuristic might be based on the cut vertices in the graph. The cut vertices of a graph must clearly map onto themselves under automorphism. Thus, a routine which found the cut vertices of a graph could then use them to further refine the vertex set partition. At one stage such a routine had been implemented. However, graphs with cut vertices tend to have rather distinct structures. There is usually not a high degree of similarity between the various parts of the graph. Thus, while the cut vertex routine was not very expensive in terms of execution time, it was found that the results obtained from it could normally be obtained using the path test and the polygon test.

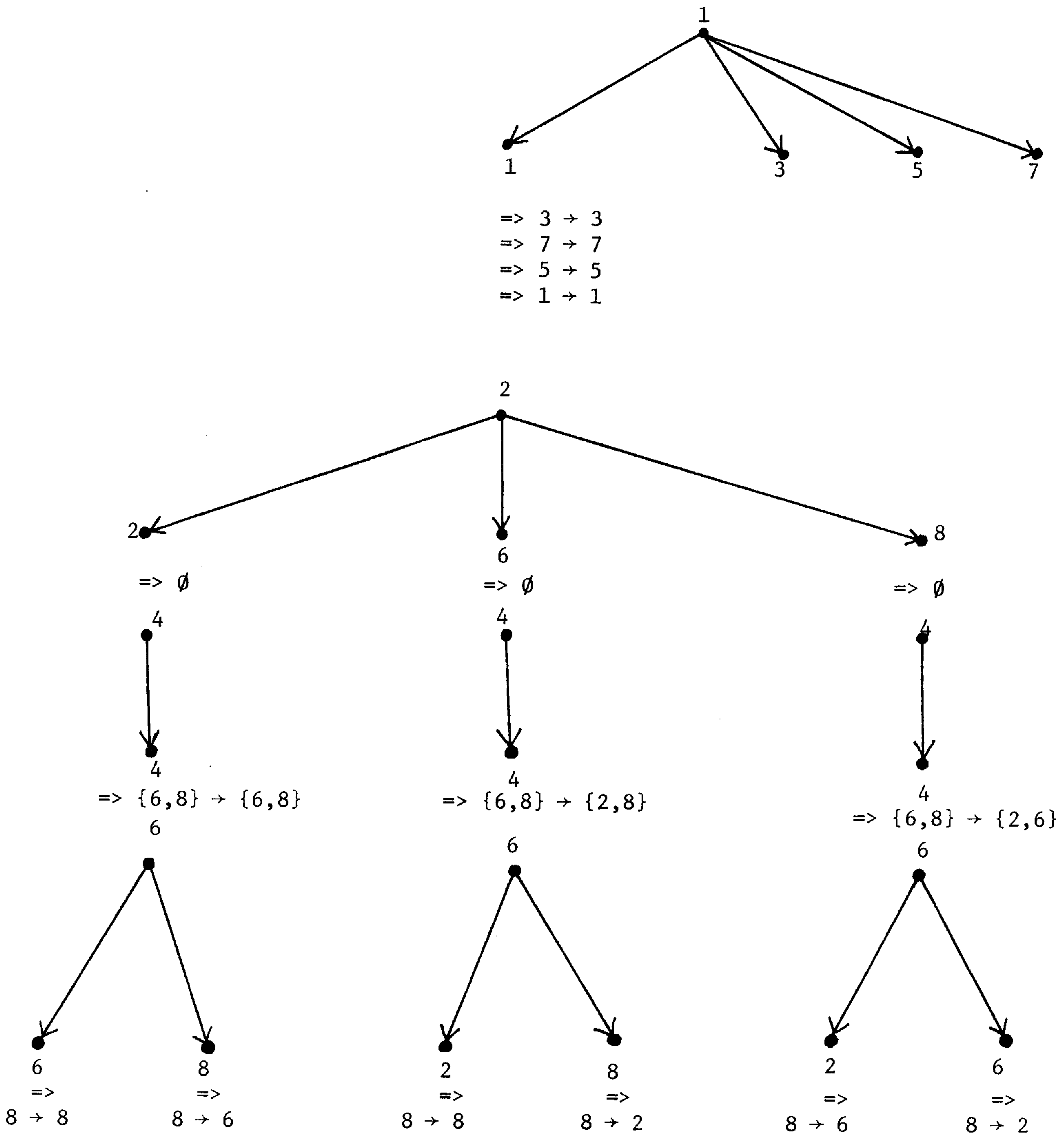


FIGURE 4.2 : ORIGINAL VERTEX LABELLING

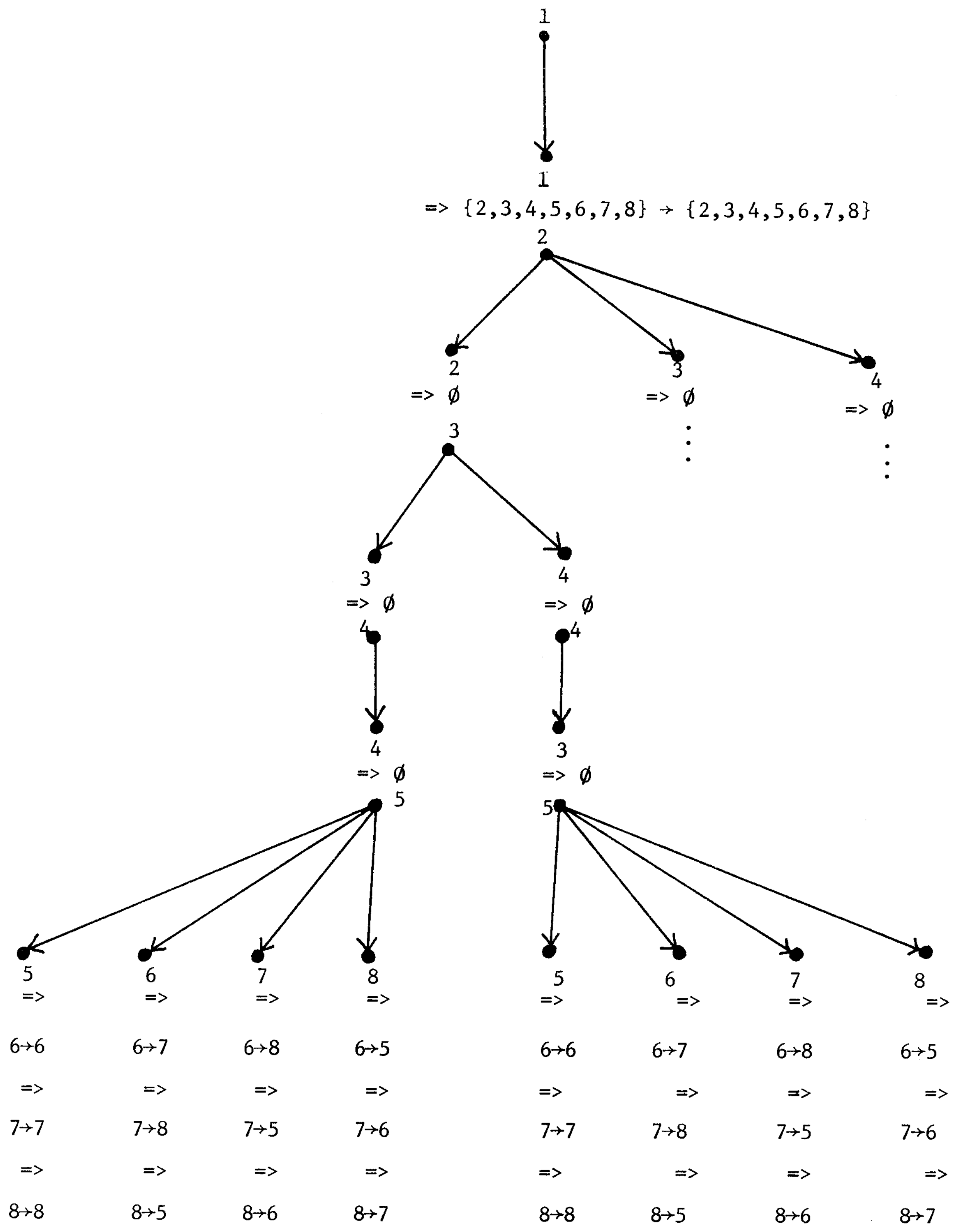


FIGURE 4.3 : REORDERED VERTEX LABELLING

The implementation of a cutset heuristic, one which used the cutsets of the graph to refine the vertex set partition, was also considered. As with cut vertices, vertices which are in cutsets can only be mapped by an automorphism onto vertices which are also in cutsets. Moreover, vertices in a cutset containing n elements cannot map onto vertices in a cutset with m elements, $n \neq m$. After trying numerous examples, it was again felt that the information obtained from such a heuristic could usually be obtained by using heuristics that were already available. Moreover, in comparison with the other heuristics a cutset routine would have been quite costly in terms of execution time.

There may be situations where a user is working with graphs that model a special type of relationship. For example, the graph might represent a switching circuit. In such cases there are often special properties assigned to the elements of the graph which should be considered when calculating the automorphism group. In the case of a switching circuit some of the vertices might represent AND gates while some might be OR gates. The user would wish to eliminate from consideration any mappings exchanging these different gate types. While such a heuristic has not been implemented it would be a simple matter to do so. Any special properties of the vertices and edges would be represented using the vertex and edge property lists of the GSYM system. It would be quite simple to implement a heuristic which when passed a list of the properties to be compared would use them to further refine the vertex set partition.

The type of heuristic used by Unger to 'extend' the existing vertex set partition by calculating special functions of the existing partition has not been mentioned. One of the functions suggested by Unger assigns each set and therefore each vertex in the set a unique number, say a prime number. One then goes through each set and applies the function to the vertices adjacent to each vertex in the set. For example, if prime numbers were used, multiplying the prime numbers assigned to the vertices adjacent to each vertex in a set would produce different results when two or more of these neighbourhoods were distinct. That is, if the neighbourhoods of each vertex of a set do not produce the same result, then that set and hence the partition can be further refined. This example is indicative of the type of functions that might be used, although as Unger points out it might prove impractical to retain the product of large prime numbers. Moreover, it is also necessary to generate the prime numbers.

The main reason for omitting such heuristics was that after experimenting with the heuristics that were already implemented it did not seem necessary. In addition, closer examination proves this type of heuristic to be based on the neighbourhood concept. That is, they attempt to refine the vertex set partition by examining and comparing the neighbourhoods of every vertex. As already indicated, several of the heuristics including the path test, the fixed point test and the tree centre test use the same neighbourhood concept, albeit in a more direct fashion. Thus, it was felt that the 'extend' type of heuristic would be somewhat redundant.

The automorphism group heuristics were originally implemented and tested in a batch processing environment. The user specified which tests were to be tried and then specified the parameters which were to be used. From the discussion about the nature of these heuristics and their suitability to different classes of graphs, clearly the user would not wish to use all of the heuristics on one graph. However, in a batch processing environment it is necessary to anticipate which tests and parameters should be used. Moreover, a test would use the parameters specified even if long before reaching the parameter limit it had become obvious (as the user could tell from later printed output) that no more information was going to be obtained.

This situation had already been anticipated. In fact, it was partially responsible for the decision to implement GSYM as an interactive system facilitating communication between the user at the screen and the routines running on the main machine. The plan was to test the theory that in the process of interacting with the automorphism group heuristics the user would develop an intuitive "feeling" about the graph or, if not about the graph, then at least about the class of graphs to which the graph belonged. As a result of this "feeling" the user would then be able to make more effective use of the available heuristics when treating other graphs in the same class. Thus, the decision was made to make the heuristics as interactive as possible without significantly impairing their efficiency or speed.

USE LIGHT-PEN TO INITIATE ACTION:

VALENCY TEST - MUST CALL IF GRAPH IRREGULAR

FIXED POINT TEST

TERMINATE

PATH TEST

POLYGON TEST

TREE CENTRE TEST

UNDIRECTED PATH TEST

UNDIRECTED POLYGON TEST

DISPLAY OBJECT LISTS

CALCULATE A(G)[5]

FIGURE 4.4: HEURISTIC SELECTION LIST

5. Calculate the automorphism group of the graph.

The user is presented with a selection list of all the heuristic routines as shown in Figure 4.4. He continues to select various heuristics to be tried until he finally invokes the automorphism group calculation algorithm using the "CALCULATE A(G)" option. At any time during this process the user may request a display of the current vertex set partition in order to judge how effective the heuristics have been and whether he should now invoke the automorphism group algorithm. Once a heuristic has been selected the user is requested to type in the parameter to be used (if such is required). After the heuristic has finished execution the system returns with the heuristic selection list and the process begins anew. Thus, it is now possible for the user to select and vary the order in which the heuristics are applied to the graph. Moreover, if it ever seems warranted he can repeat tests at will. Some of the heuristics, however, have been rewritten to make them even more interactive.

For example, the output from the path test is displayed whenever the path length is incremented. Figure 4.5 shows an example of one such display. The upper limit of 17 is a limit which in practice is never reached.

| LENGTH | 1 | 2 | 3 | 4 | 5 | 6 | .... | 17 |
|---|---|---|---|---|---|---|---|---|
| V001 | 1 | 3 | 2 | 3 | 3 | 4 | | |
| V002 | 1 | 2 | 2 | 0 | 0 | 0 | | |
| V003 | 2 | 2 | 2 | 0 | 0 | 0 | | |
| V004 | 1 | 3 | 3 | 3 | 2 | 1 | | |
| V005 | 2 | 3 | 2 | 3 | 3 | 4 | | |
| V006 | 1 | 3 | 1 | 1 | 0 | 0 | | |
| . | | | | | | | | |
| . | | | | | | | | |
| . | | | | | | | | |
| . | | | | | | | | |
| V00N | | | | | | | | |

FIGURE 4.5 : PATH TEST VECTOR DISPLAY

The user can tell at any point how well the path test is performing. More to the point, he is able to halt the test if it becomes obvious that no more information will be forthcoming. For example, if the vector columns are beginning to repeat themselves the test should be halted. It is no longer necessary for the test to execute until the maximum distance specified is reached. The user indicates after each increment whether or not the test is to continue. For directed graphs the process of displaying the path list is repeated for both positive and negative paths. When the undirected path test is invoked the same process is used. That is, it is just the same as if the graph were originally undirected and the normal path test had been invoked.

In the case of the fixed point test an attempt is made to refine the vertex set partition after each neighbourhood of the fixed vertex is calculated. The user is asked after each such attempt whether he wishes the test to continue. At this point the user may ask to see the current vertex set partition in order to judge whether the test is succeeding and whether it seems likely to continue to succeed. What is more important he may find that another fixed vertex has been uncovered. In this case he may wish to stop the test in order to start it again with the new vertex. This is important because it effectively provides a means for the user to, so to say, march about the graph using this test to examine the structure of different parts of the graph.

The tree centre test first informs the user of the identity of the centre or bicentre. In the case where the tree has a centre the fixed point test is invoked and the user controls the operation of this heuristic in the manner described above. If there is a bicentre pair then a test based on the neighbourhood concept and similar in nature to the fixed point test is applied to the tree. This routine attempts to refine the partition on the basis of the first neighbourhood of the bicentre pair. The first neighbourhood consists of those vertices which can be reached from either vertex of the bicentre by a path of length 1. The user is then asked if he wishes to continue the test. He may, of course, examine the current vertex set partition in order to help him decide. This process is repeated for each neighbourhood just as with the fixed point test, the difference now being that the neighbourhoods are calculated by examining paths originating at either of two vertices rather than at just one vertex.

It is difficult to evaluate with any real degree of precision just how advantageous the interactive automorphism group heuristics are in comparison to the original batch environment heuristics. Our experience has been that they can be used considerably more effectively. What is most pronounced is the saving in execution time. For example, when testing the non-interactive heuristics it was found that it was usually possible after several tries to select a set of tests and corresponding parameters which were the 'best' possible selection from the viewpoint of execution time required versus results obtained. That is, several tests were tried to determine how much time was subsequently required to calculate the automorphism group. The same graph would be tried again but with some of the tests and/or parameters changed. This process would be repeated until the chosen combination worked quite well for the given graph. Of course, the user at the display screen cannot do this; moreover, he would not wish to recalculate the automorphism group once he had it.

However, experience has shown that when using the interactive heuristic tests it is possible to come very close to selecting the 'best' set of tests and parameters the first time. Actually, what happens is that it is now possible to evaluate the tests and the corresponding parameters almost immediately. Tests which are having little or no effect can be halted quickly without their use involving a significant loss of execution time. Other tests can then be tried and evaluated in the same manner until the time appears ripe to invoke the automorphism group calculation algorithm.

Unfortunately, we cannot define or meaningfully describe in concrete terms the intuitive "feeling" we have been discussing. Were it possible to do so it might help the user in his own selection of tests. What is possible, however, is to compare the number of operations and the time requirements of the various heuristics. In this way it is possible to better explain some of the tradeoffs and considerations involved in selecting the heuristics and determining the point at which the automorphism group algorithm should be invoked.

The processing time required by each heuristic is to be determined as a function of n ($|VG|=n$). In order to do this the steps involved in each heuristic (other than the valence test) and the number of operations required by each step are examined. The following assumptions are made:

$\delta_1$ = machine cycles required to access an element in storage.

$\delta_2$ = machine cycles needed to compare two elements.

$\delta_3$ = machine cycles needed for an integer addition.

$\delta_4$ = machine cycles needed for an element replacement.

Since the heuristics are written in PL/1, the above constants are taken to mean the respective operations as coded in PL/1.

Example 4.2:

IF A=B THEN... - denoted by $\delta_2$

I+J, 2+3, I+3 - are all denoted by $\delta_3$

A=B is considered to require $\delta_1 + \delta_4$ cycles since the value of B must be accessed and then assigned to A.

We make use of two other constants in the discussions below:

$\ell$ = average number of successors of a vertex.

$\beta$ = average number of vertices in any neighbourhood of a vertex.

These two constants are used only to simplify the formulae below. However, it is important to note that $0 < \ell < n$ and $0 < \beta < n$. Thus, when these constants appear as coefficients we know that the formula is bounded by the same formula with the constant(s) $\ell$ and/or $\beta$ replaced by n.

The following is a list of the number of operations required by each heuristic routine.

## Timing Considerations

### Fixed Point Test

Undirected Graph:

Invariant: $n(4\delta_1 + 3\delta_2 + 3\delta_4)$

1st Neighbourhood:

$n^2\beta(2\delta_1 + 2\delta_2 + 2\delta_4) + n\beta(2\delta_2 + \delta_4) + \ell(2\delta_1 + 2\delta_4)$

For each additional neighbourhood:

$n^2\beta(2\delta_1 + 2\delta_2 + 2\delta_4) + n\beta(2\delta_2 + \delta_4) + \beta\ell(4\delta_1 + \delta_3 + 4\delta_4)$
$+ \beta(2\delta_1 + 2\delta_4)$

Directed Graph:

Invariant: $n(5\delta_1 + 3\delta_2 + 4\delta_4)$

1st Neighbourhood:

$n^2\beta(4\delta_1 + 4\delta_2 + 4\delta_4) + n\beta(4\delta_2 + 2\delta_4) + n(10\delta_1 + 3\delta_2$
$+ 2\delta_3 + 10\delta_4)$

For each additional neighbourhood:

$n^2\beta(4\delta_1 + 4\delta_2 + 4\delta_4) + n\beta(5\delta_1 + 4\delta_2 + \delta_3 + 5\delta_4)$
$+ \beta\ell(4\delta_1 + \delta_3 + 4\delta_4) + \beta(4\delta_1 + 4\delta_4)$

### Path Test

Undirected Graph:

1st Increment:

$n\ell(\delta_1 + \delta_4) + n(3\delta_1 + 3\delta_4)$

Each additional increment:

$n^2(3\delta_1 + \delta_2 + \delta_3 + 3\delta_4) + n\beta\ell(\delta_1 + \delta_4) + n\beta(\delta_1 + \delta_4) + n(3\delta_1 + 3\delta_4)$

Attempt to refine partition:

$n^2d(2\delta_1 + 2\delta_2 + 2\delta_4) + nd(2\delta_1 + 2\delta_2 + 2\delta_4) + d(\delta_1 + \delta_4)$,

where d = maximum path length.

Directed Graph:

The number of operations required for positive paths is the same as is required for an undirected graph. The same number of operations is again required to go through the negative paths.

The same number of operations is required to attempt to refine the partition. That is,

$n^2d(2\delta_1 + 2\delta_2 + 2\delta_4) + nd(2\delta_1 + 2\delta_2 + 2\delta_4) + d(\delta_1 + \delta_4)$,

where d = maximum path length.

This number of operations is required twice, however, once after creating the positive paths and then again after creating the negative paths.

Polygon Test

$n^2d(2\delta_1 + 2\delta_2 + 2\delta_4) + n\ell^2d(\delta_1 + 5\delta_2 + 4\delta_3 + \delta_4)$

$+n\ell d(2\delta_3 + 2\delta_4) + nd(2\delta_1 + 2\delta_2 + 2\delta_4) + n\ell(\delta_1 + \delta_4)$

$+n(3\delta_1 + 3\delta_4) + d(\delta_1 + \delta_4)$, where d = maximum polygon length used.

Tree Centre Test

Let w = maximum distance from any vertex to centre or bicentre of tree. $0 < w < n$.

To find centre or bicentre:

$n^2w(2\delta_1 + \delta_2 + \delta_3 + 3\delta_4) + nw(4\delta_1 + 3\delta_2 + \delta_3 + 5\delta_4)$

$+ n(\delta_1 + 2\delta_2 + \delta_4)$

If the tree has a centre:

$n^2(2\delta_1 + \delta_2 + 2\delta_2) + n\delta_2$ + fixed point test is invoked.

If the tree has a bicentre:

$n^2(4\delta_1 + 2\delta_2 + 4\delta_4) + n\delta_4$ + following # of operations:

1st Neighbourhood:

$n^2\beta(2\delta_1 + 2\delta_2 + 2\delta_4) + n\beta(2\delta_2 + \delta_4)$

$+ \ell(8\delta_1 + 2\delta_2 + 2\delta_3 + 8\delta_4)$

Each additional neighbourhood:

$n^2\beta(2\delta_1 + 2\delta_2 + 2\delta_4) + n\beta(2\delta_2 + \delta_4) + \beta\ell(4\delta_1 + \delta_2 + \delta_3 + 4\delta_4)$

$+ \beta(2\delta_1 + 2\delta_4)$

Directed Graph:

$n^2(2\delta_1 + \delta_2 + \delta_4)$ operations are needed to create an equivalent undirected tree.

The rest of the test is the same as required for an undirected graph except that when there is a bicentre the test that is then used takes twice as many operations as for an undirected graph with a bicentre since now the bicentre neighbourhood test is performed for both positive and negative paths.

<u>Undirected Path and Polygon Tests</u>

The number of operations required to remove and later replace the edge directions is given by:

$n\ell(\delta_1 + \delta_3 + \delta_4) + n(5\delta_1 + \delta_3 + 4\delta_4)$.

The appropriate test is then invoked with no change in the number of operations required from what is needed for the original tests.

As indicated previously all of the tests are very quick. In fact, all of them are bounded by an upper bound of $n^3$ operations. The upper bound of $n^3$ operations is, in practice, on the order of $n^2$ operations. The reason for this is that the values for $\ell$, $\beta$ and d are normally relatively small in comparison to n. Listing the heuristics according to the number of operations required and hence by the amount of execution time required places them in the following order:

Fixed Point Test

Polygon Test

Path Test

Undirected Polygon Test

Undirected Path Test

Tree Centre Test

$\Big\downarrow$ number of operations required increases in this direction

However, this list is misleading. For example, it suggests that the fixed point test is the 'best' test to invoke first. However, as fixed vertices are normally revealed as a consequence of the application of other tests so that one usually delays invoking the fixed point test (unless the results of the valence test reveal a number of fixed vertices). Furthermore, if the graph is a tree there is no point whatsoever in invoking the polygon test in preference to the tree centre test. However, if the user truly has no conception whatsoever of the nature of a graph, he might well be guided by the above list into choosing the polygon test over the path test.

It is necessary to remember that the number of operations and the time required by the various tests are not as important as the results they obtain. The tests are heuristic rather than algorithmic. If they were algorithms that were all equally capable of producing the same vertex set partition then one would be compelled to use only the fixed point test. However, as they are heuristic in nature there is no definite rule as to which test is better in any given situation. The main benefit of the above timing considerations is to stress again the difference in the number of operations required by the heuristics and the number required by the automorphism group algorithm. The algorithm takes on the order of n! operations ($|VG|=n$) when the vertex set has not been partitioned. However, a successful partitioning or refinement of the vertex set partition by any heuristic results in a large reduction in the number of operations later required by the algorithm, especially in comparison to the number of operations required to execute the heuristic. Consider, for example, a graph with 10 vertices. The number of possible vertex mappings is 10! = 3,628,800. Suppose, however, that the path test partitions the vertex set into just two sets of 5 vertices each. Then the number of possible mappings is only 5!x5! = 14,400, a much smaller number of mappings. The saving is much larger than the upper limit of at most $n^3 = 1000$ operations that would be required by the path test.

Now that the various tradeoffs involved in obtaining the automorphism group have been considered, the question is how to actually use the automorphism group. Thus, in Chapter V the problem of generating the visual representations of graphs and their symmetries is investigated.

# CHAPTER V : VISUAL REPRESENTATION OF GRAPHS

In this chapter the discussion centres about the use of the automorphism mappings of the graph in generating three dimensional visual representations.

In Chapter III an investigation was begun into how one might interpret visually the mappings of the automorphism group. At that time the relationship between the automorphisms and a corresponding visual image was discussed. Every automorphism can be separated into orbits which represent mirror-image and/or rotational symmetries. This chapter describes some of the ways that these symmetries have been or can be used to create visual representations of graphs. It is important to remember that the primary goal is to devise representation techniques whose end product, the display, reveals the structure of the graph. Thus, while different representation criteria and methods for generating these displays will be discussed, the main emphasis is on the portrayal of the symmetries of the graph. For example, a three dimensional representation of a planar graph may be preferable to a planar embedding if the three dimensional representation better illustrates the symmetry in the graph.

Two representation schemes which have been implemented using the GSYM system will be discussed in detail. In addition, several representation schemes which have not been implemented but which appear to be suitable candidates for implementation will also be discussed. The discussion will centre around the advantages and disadvantages of these various schemes in hopes of providing some insight into the manner in which one might go about devising representation algorithms.

Once the user has invoked the automorphism group calculation routine, the routine returns with a display of the type shown in Figure 5.1. This indicates that an automorphism mapping has been found. If there are no more automorphisms in the automorphism group of the current graph the routine displays the message "AUTOMORPHISM GROUP COMPLETE" instead of a display such as the one in Figure 5.1. The display in Figure 5.1 gives the orbits found in the automorphism

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 2 & 3 & 4 & 5 & 7 & 6 & 8 & 16 & 11 & 15 & 12 & 9 & 14 & 10 & 13 \end{pmatrix}.$$

This automorphism is only given as an example of the method the automorphism group routine uses to report the automorphisms it finds. The graph from which the automorphism was obtained has not been shown. This, in fact, is the situation which the user normally faces. An automorphism of the current graph is displayed on the screen but, as yet, there is no picture of the graph itself.

It is possible to determine the automorphism mapping from the display in Figure 5.1 as the display give the orbits of the automorphism. In this example vertices [1] 1,2,3,4,5,8,12 and 14 are fixed by the automorphism. There is one mirror-image symmetry mapping vertex 6 onto vertex 7 and 7 onto 6. There are two rotational symmetries, 9 →16 16→13, 13→9 and 10→11, 11→15, 15→10. It is also possible to have the

1. The display in Figure 5.1 uses the internal names V001, V002,... V0016 for the vertices 1,2...16 respectively. The "V" notation is used in the GSYM system to indicate vertex labels as opposed to edge labels which start with an "E".

```
                WAITING FOR LIGHT PEN RESPONSE :



(V001)  (V002)  (V003)  (V004)  (V005)  (V006  V007)  (V008)



    (V009  V016  V013)  (V010  V011  V015)  (V012)  (V014)
```

FIGURE 5.1 : DISPLAY OF ORBITS IN CURRENT AUTOMORPHISM

actual mapping displayed in the manner shown in Figure 5.2 as each automorphism is calculated.[2] This display shows the actual automorphism mapping in terms of image and object vertices. That is, if vertex v is mapped onto vertex w then v is the image vertex and w is the object vertex of the mapping $v \rightarrow w$.

The mapping given in the form in Figure 5.2 is exactly the same as that given as a display of the automorphism orbits. However, the orbit display is usually preferable since the user can tell immediately from it the types of symmetries represented by the automorphism. It is also possible to determine immediately from the orbits how many vertices are actively involved[3] in the automorphism. As discussed in Chapter III the automorphism with the smaller number of fixed vertices usually produces the better display. Thus, when the user is presented with the display in Figure 5.3[4], he can better judge whether or not to invoke one of the representation routines.

Figure 5.3 indicates that at this point the user has three options open to him. He can specify that he does not wish to have a representation generated for the current automorphism. In this case, the automorphism group routine begins to search for the next automorphism mapping. If the user wishes to see a display of the graph he can invoke the General Representation Routine by selecting the "YES" response. He may also select the Tree Representation Routine.[5]

2. This option is available using a GSYM feature which allows the user to set special flags which are available to system and user written routines in the GSYM system.
3. Actively involved means here that a vertex is not fixed by the mapping.
4. This is the next display produced by the system after the user has finished examining the automorphism mapping.
5. Provided that the graph is a tree.

```
PAGE CONTROL PFK28 - BACKWARD  PFK31 - FORWARD

                PFK30 - STOP




   AUTOMORPHISM


IMAGE   V001    OBJECT   V001

IMAGE   V002    OBJECT   V002

IMAGE   V003    OBJECT   V003

IMAGE   V004    OBJECT   V004

IMAGE   V005    OBJECT   V005

IMAGE   V006    OBJECT   V007

IMAGE   V007    OBJECT   V006

IMAGE   V008    OBJECT   V008

IMAGE   V009    OBJECT   V016

IMAGE   V010    OBJECT   V011

IMAGE   V011    OBJECT   V015

IMAGE   V012    OBJECT   V012

IMAGE   V013    OBJECT   V009

IMAGE   V014    OBJECT   V014

IMAGE   V015    OBJECT   V010

IMAGE   V016    OBJECT   V013
```

FIGURE 5.2 : AUTOMORPHISM MAPPING DISPLAY

FIND REPRESENTATION ?

YES

NO

DISPLAY TREE

FIGURE 5.3 : REPRESENTATION SELECTION LIST

General Representation Routine

This routine is a very general display procedure which attempts to handle all classes of graphs by creating representations based solely on the mirror-image and rotational symmetries in the automorphism group mappings. The flowchart in Figure 5.4 shows the overall logic of the routine. This flowchart contains two 'black boxes' representing routines for handling mirror-image and rotational symmetries. An explanation of what actually happens when these routines are invoked is given below. The procedure is basically quite simple. Fixed vertices are placed on an imaginary axis of symmetry through the centre of the screen. The mirror-image vertex pairs are then placed in corresponding positions on either side of this axis. Finally, the rotational vertices are placed in separate planes of rotation for each rotational orbit. These rotations are drawn using the ring structure algorithm described in Chapter III.

There is a serious fault in the above plan for displaying the fixed vertices about a single axis of symmetry. The problem arises when fixed vertices are found adjacent to other fixed vertices. Consider the example shown in Figures 5.5A, 5.5B and 5.5C. Figure 5.5A shows the desired form of the display. The problem of coincident edges along the axis of symmetry can in certain cases, such as the above, be resolved by the order in which the vertices are displayed. An alternative solution is to use curved edges (via the light-pen edge feature of GSYM). However, as can be seen in Figure 5.5C this solution can destroy the desired mirror-image form of the display. The problem is even more pronounced

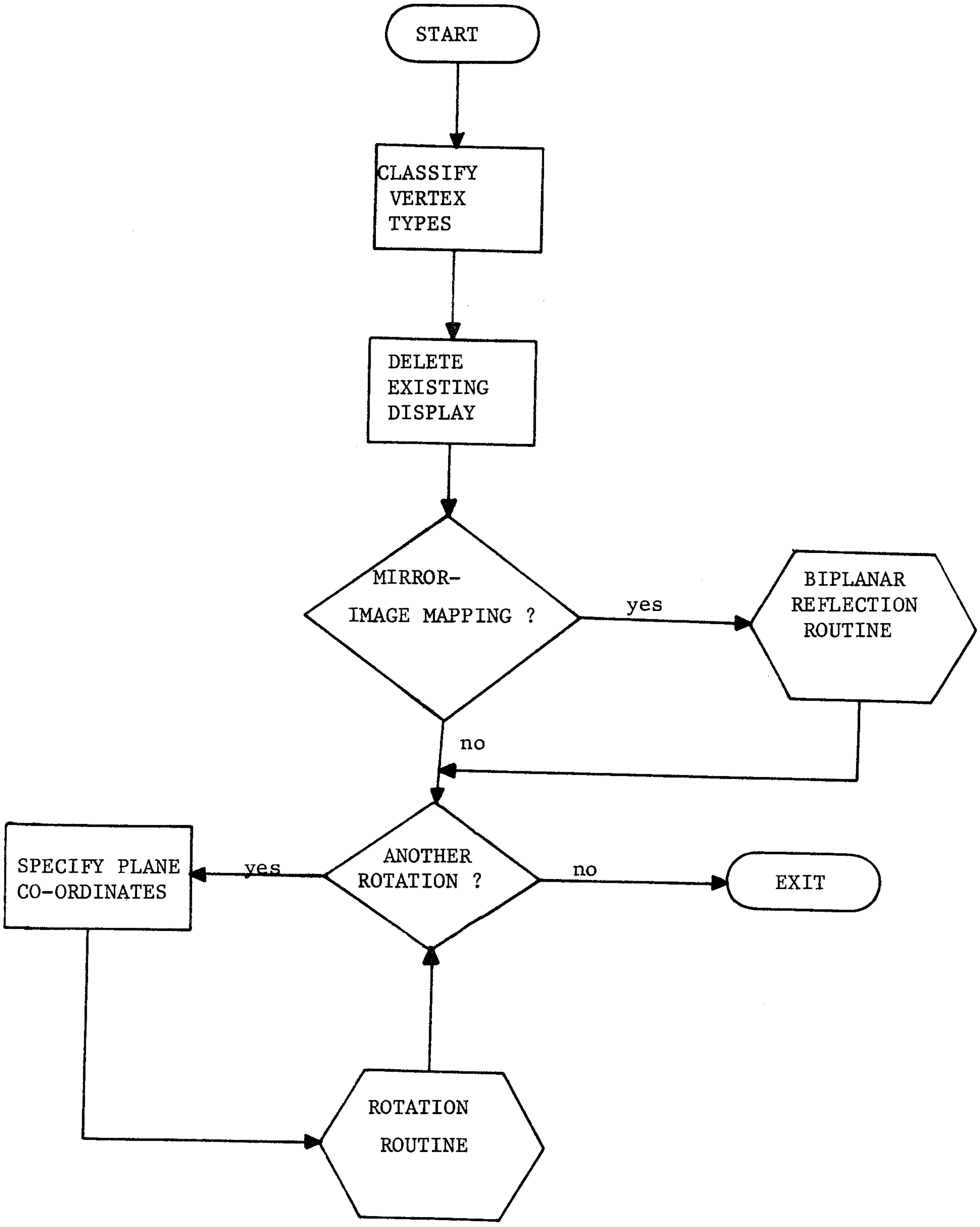
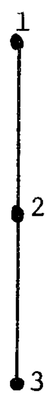
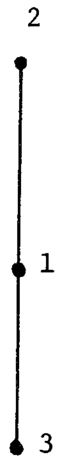


FIGURE 5.4 : GENERAL REPRESENTATION ROUTINE



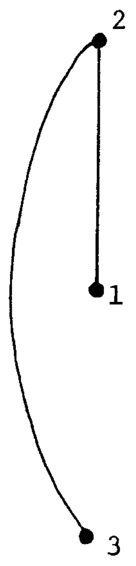
FIXED VERTICES
DISPLAYED WITHOUT
EDGE OVERLAPPING

FIGURE 5.5A



MISLEADING DISPLAY
OF FIXED VERTICES
INCORRECTLY SUGGESTS
VERTEX 2 IS NOT ADJACENT
TO VERTEX 3 AND THAT
VERTEX 1 IS ADJACENT
TO VERTEX 3.

FIGURE 5.5B



USE OF CURVED
EDGES TO RESOLVE
PROBLEM

FIGURE 5.5C

when there exists a set of fixed vertices which form a polygon. In this case there is no possible positioning of the vertices along the axis of symmetry which will prevent the edges of the polygon from overlapping. The resultant display is usually quite misleading as to the number of edges being displayed and the identity of the proper end-vertices of these edges.

The solution that has been adopted is to create another axis of symmetry in a different plane from that of the original axis. The first axis is filled with fixed vertices until the overlapping problem arises. An attempt is then made to place the remaining fixed vertices on the second axis. It is again possible that not all of the fixed vertices will go on this axis without causing edge overlapping. The process is repeated with the user creating a third axis in a different plane from the first two. This process is continued until all the fixed vertices have been accommodated. Thus, the final display may contain several axes of symmetry in different planes. This solution is acceptable only because the graph is considered as a three dimensional entity. Furthermore, the rotation facility of the GSYM system makes it possible for the user to view the graph from different sides.

The following is a detailed description of the procedure used to position the vertices on the screen.

Let A = "Axis Set" - the set of fixed vertices in the current automorphism mapping.

P = "Plot Set" - the set of vertices positioned so far. Initially, P = $\emptyset$, the empty set. P = VG, the vertex set of the graph, when the procedure has finished.

S = "Path Set" - a set of fixed vertices representing a path in the graph.

Display of Fixed Vertices

0) If $A = \emptyset$, then go to step 5. Otherwise, ask the user to select the z co-ordinate for the plane on which the next axis is to be located.[6]

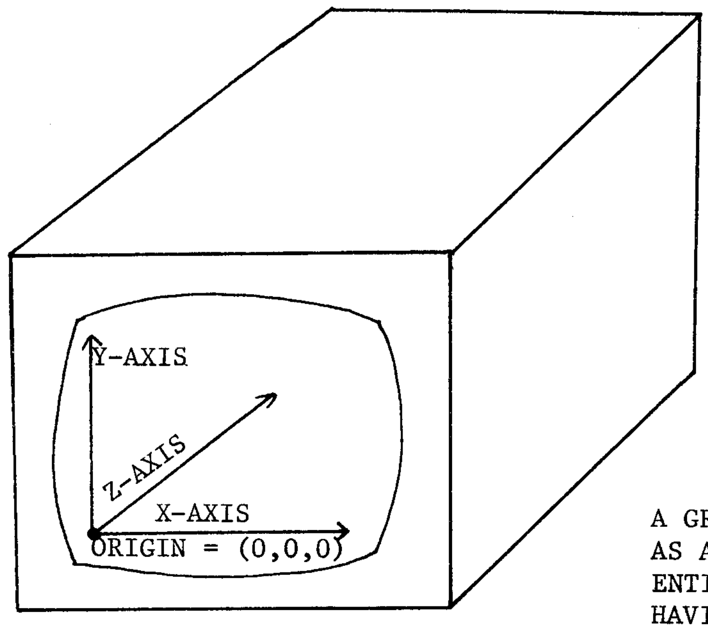
1) Let v be any vertex in A which is not adjacent to any vertex in P. If there is no such vertex, then go to step 3. If such a vertex exists, then let $S = \{v\}$ and $A = A - \{v\}$. In this step the routine attempts to find another fixed vertex which can be displayed on the current axis without causing any edge overlapping.

2) Find a vertex, w, in A which is adjacent to v, but which is not adjacent to any vertex in $P \cup S$ other than vertex v. Since the graph is simple there is no possibility of multiple edges between v and w when the graph is undirected. However, the above statement also implies that if the graph is directed, then there cannot be both a positive path from v to w and a negative path from v to w.

If there exists such a vertex then let $S = S \cup \{w\}$, redefine v to be the new vertex w and go to step 2. If there is no such vertex, then let $P = P \cup S$ and go to step 1.

In steps 1 and 2 the algorithm is attempting to place as many paths of fixed vertices as possible on the axis. Step 2 attempts to extend a path which was started in step 1. The path is extended as

6. The IBM 2250 graphic display as shown in Figure 5.6 is considered by the GSYM system to be a three dimensional cube with the screen representing xy co-ordinate values. All points on the screen have a z co-ordinate value of 0. Non-zero z co-ordinate values inside the cube are specified by having the system temporarily rotate the cube so that the screen represents yz co-ordinate values.



A GRAPH IS DEFINED AS A 3-DIMENSIONAL ENTITY WITHIN A CUBE HAVING THE APPROXIMATE SHAPE OF THE ACTUAL PHYSICAL DISPLAY UNIT.

FIGURE 5.6 : THE IBM 2250 GRAPHIC DISPLAY UNIT

far as possible without causing edge overlapping. Note that if a path is in a polygon, the path cannot be extended to complete the polygon due to the restriction that the new vertex, w, not be adjacent to any vertex already in $P \cup S$.

3) Let $p = |P|$, the number of vertices in P. Let d = the length of the axis of symmetry/(p+1). At present the length of the axis is taken as 4096. The user may control the size of the display by specifying the axis length.

4) Position the p vertices in P on the axis of symmetry at the points $(x_o, y_o+d)$, $(x_o, y_o+2d)$, ..., $(x_o, y_o+pd)$. The point $(x_o, y_o)$ is the lowest screen point on the axis. The vertices are positioned from $(x_o, y_o+d)$ towards the top of the screen in the same order as they were placed in P. Set $P = \emptyset$. Go to step 0.

5) If there are no fixed vertices in the automorphism, go to step 6. Otherwise, invoke the MOVE option[7] in order to allow the user to make any changes he wishes in the layout of the fixed vertices.

Actually at this point the user should not alter the x or z coordinates of any of the fixed vertices as this could interfere with the form of the rest of the representation. The user is being given the opportunity to move the fixed vertices up and down the axes. He is not expected to move vertices off the axis or out of the plane in which the axis is embedded.

7. The MOVE option is a GSYM feature which allows a user to move elements of a graph about the cube in which the graph is defined.

In order to allow the user to determine the location of the vertices along the various fixed vertex axes (assuming more than one axis was required for the fixed vertices), the graph is now rotated $90^{\circ}$ about the y axis. This, in effect, gives the user a side view of the axes.[8] The MOVE option is again invoked to allow the user to relocate vertices if he wishes. Again, he should only move vertices up and down axes. Finally, the graph is rotated back to its original position. Go to step 6.

<u>Addition of Mirror-image Vertex Pairs</u>

6) If there are no fixed vertices in the automorphism then go to step 7.

Examine each fixed vertex in order to determine if there exists a pair of mirror-image vertices which are adjacent to the fixed vertex and which have not yet been positioned. If there are no such vertex pairs to be plotted go to step 7.

Otherwise, for each fixed vertex for which there is an adjacent pair of vertices to be plotted, assign to the vertex pair the same y and z co-ordinates as the fixed vertex. Then assign the values 1648 and 2448[9] to the x co-ordinates of the first and second vertices, respectively, of the pair. Thus, these vertex pairs are placed on two imaginary lines parallel to the axis of symmetry. Only one such vertex pair is used with each fixed vertex even if there are several mirror-image pairs adjacent to the fixed vertex. Go to step 7.

8. The result is the same as would be achieved if the user were able to walk around the 2250 Display Unit and view the graph from the side of the display.

9. The values used in this procedure were chosen after several tests and are a function of the scheme used on the IBM 2250 Display Unit to address points on the screen.

7) Let m = the number of mirror-image pairs which have not been assigned their screen co-ordinates.

Let $d_1 = 2896/(m+1)$, $d_2 = 1448/([m/2] + 1)$.

The mirror-image pairs are all placed in an xy plane in the centre of the cube in which the graph is defined. The x and y co-ordinates of the first vertex of each pair are assigned values according to the scheme illustrated in Figure 5.7.

The second vertex of each pair is assigned x and y co-ordinates so that it is displayed as the mirror-image of the first. These pairs are positioned in this fashion in the hope that the "X" format will reduce if not eliminate the problem of overlapping edges. The values of $d_1$ and $d_2$ have been chosen to spread the display about the screen without getting too close to the border.

## Postediting the Graph Representation

8) If the graph contains only rotational vertices go to step 9.

After positioning the fixed and mirror-image vertices the user is presented with the display shown in Figure 5.8. The purpose in giving the user the chance to edit the graph now is to allow him to concentrate on the mirror-image symmetry before the rotational symmetries (if any) are added to the display.

If the user selects the free format edit option, then the GSYM MOVE option is invoked and any positional changes the user wishes to make can be carried out.[10] The mirror-image edit option makes it possible

10. Allowing the user access to the very powerful GSYM MOVE option at this point means the user could destroy the symmetry of the mirror-image vertices. In practice, however, the mirror-image edit option has proven more useful and would normally be the option chosen.

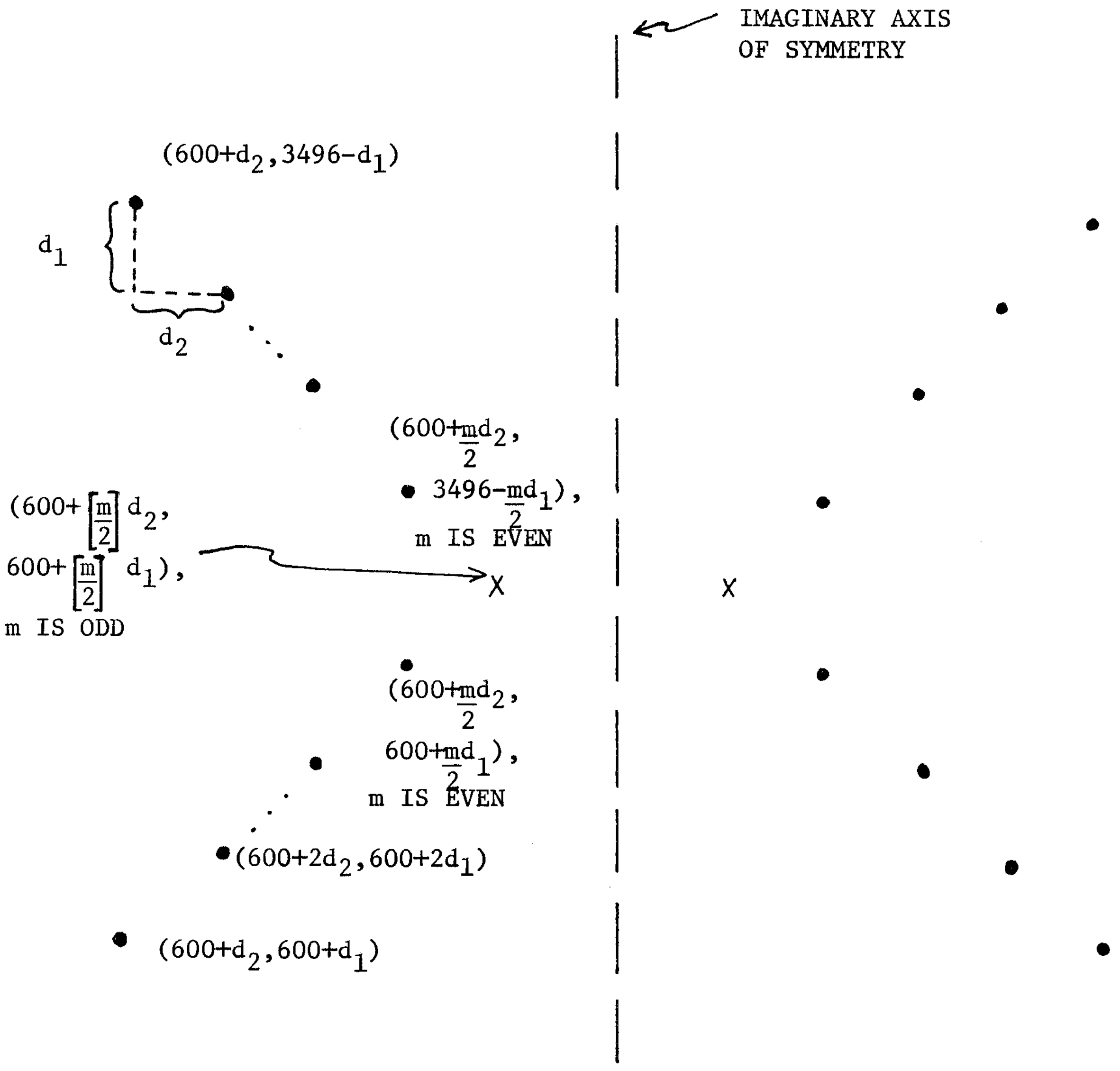


FIGURE 5.7 : MIRROR-IMAGE PAIR COORDINATES

```
SELECT EDIT OPTION:

     FREE FORMAT EDIT

     MIRROR-IMAGE EDIT

     HARD COPY

     CONTINUE

     STOP

     SHOW ORBITS

     ROTATE
```

FIGURE 5.8 : POSTEDITING SELECTION LIST

to handle the mirror-image pairs as single entities. Under this option if the user moves a vertex of such a pair, then its image (under the current automorphism) is moved at the same time in such a way as to preserve the mirror-image symmetry between the two vertices. This feature is especially useful when moving each vertex of a mirror-image pair to the opposite side of the axis reduces the number of edge intersections in the display.

The GSYM rotation option is available so that the user can view the graph from different sides as is desirable when there is more than one axis of symmetry. It is particularly useful to have the rotation feature available when there are a significant number of edges connecting vertices on one axis to vertices on the other axes. These edges cannot be properly viewed except from the side of the display unit, an effect which can be achieved (i.e. simulated) only by rotating the graph. It is also possible to request a hard copy of the display, even though the display is incomplete since the automorphism may contain rotational vertices.

The stop option was included as a natural consequence of a desire to allow the user to change his mind at almost any point and start fresh, even part way through the creation of a visual representation. The "SHOW ORBITS" option allows the user to view the orbits of the current automorphism again. This helps to relate the representation being generated to the automorphism from which it is generated. The continue option is invoked when the user is ready to have the rotational symmetries added to the display.

Rotational Symmetries

9) If all the rotational symmetries present in the automorphism have been plotted then the procedure is finished.

Have the user select the $\underline{z}$ co-ordinate for the plane in which the next rotation is to be embedded. Then ask the user to specify the screen distance between the vertices of the rotation and the screen co-ordinates of the first vertex in the rotations.

10) Use the method described in Chapter II for drawing polygons to position the vertices in the rotation. The vertices are assigned screen co-ordinates in the same order as they appear in the orbit representing the rotation, beginning with the user specified co-ordinates for the first vertex. Note that the vertices in the rotation may or may not be adjacent unlike the polygon the method was originally used to display (see Chapter II).

11) Enter the postediting routine. The user now has the same options described under step 8. Once the continue option is selected the procedure goes to step 9.

The above procedure uses no information about the graph and its structure other than the information available on the types of symmetries represented by each automorphism. It is completely general in nature and because of this generality one might expect that it would encounter difficulties. The possibility of successfully using one representation routine to handle all classes of graphs seemed remote even before the routine was tested. Some of the problems, the disadvantages and also the advantages in trying to apply the above procedure to all types of graphs are discussed below.

One of the biggest problems is how one judges the importance of the tradeoff between stressing the structure of the graph and presenting a 'pretty' picture of the graph. By its very nature the above procedure does a good job of displaying the different symmetries in the automorphism. However, there is little attention paid to the final picture one creates by isolating the individual symmetries and then rather artificially grouping them together on the screen. While the user is able to identify the various symmetries and the symmetry types, the overall view of the graph is often unnatural and less than satisfactory. Thus, one must rate this procedure a failure in the sense that the final picture is usually not very esthetic and frequently cannot be used by the user as it stands.

However, this very weakness does vindicate the earlier conclusion, based on the attempts at ring structure representations in Chapter III, that an interactive system is a necessary tool for attacking the visual representation problem. It is only because these representation routines are implemented on an interactive graph processing display system that it is possible to allow the user to perform postediting of the display. And it is the postediting facility which allows the user in most cases to transform an unsuitable visual representation into a picture that he finds pleasing and satisfactory.

Consider the following example of the representations produced by the General Representation Routine.

Figure 5.9 shows a graph as it might have been created by a user working with the GSYM system. Figure 5. 10 shows a visual representation of the same graph as created by the General Representation Routine using the specified automorphism. The automorphism is quite simple,

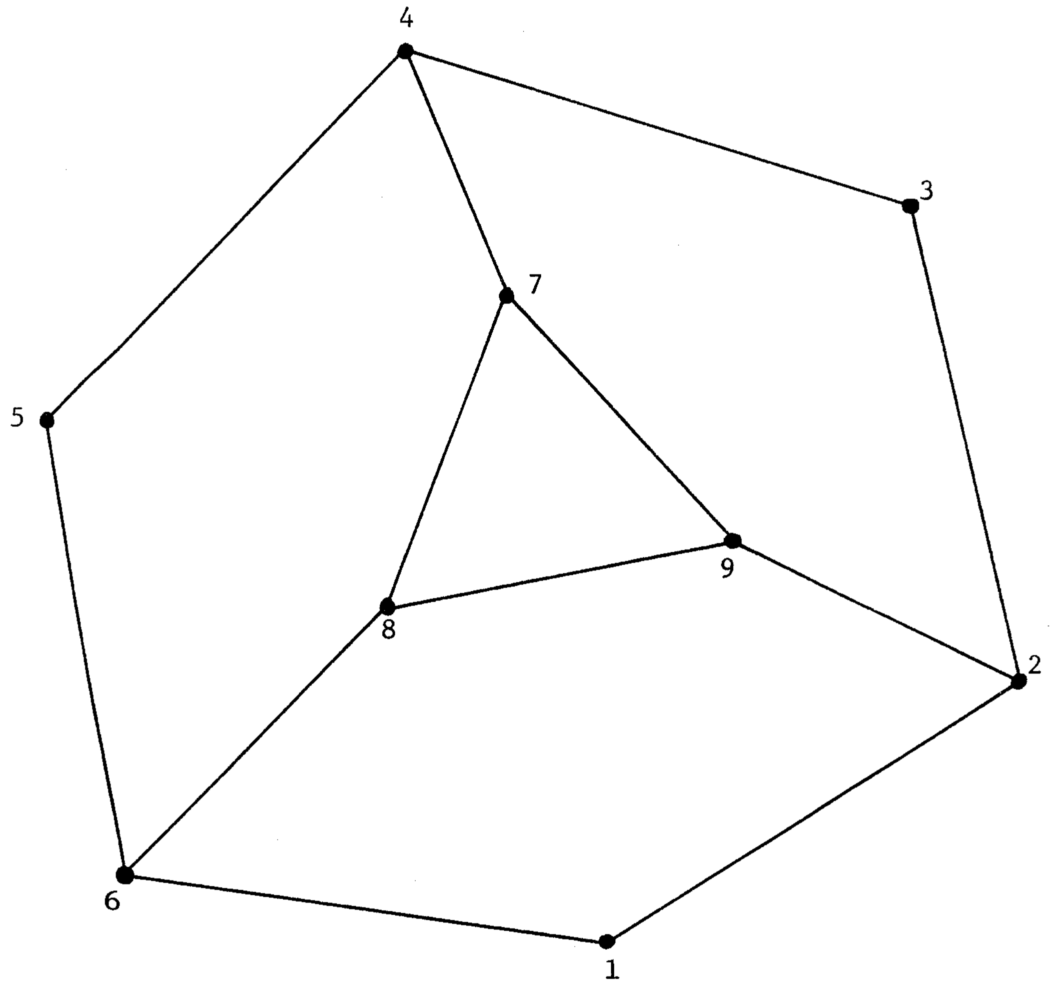
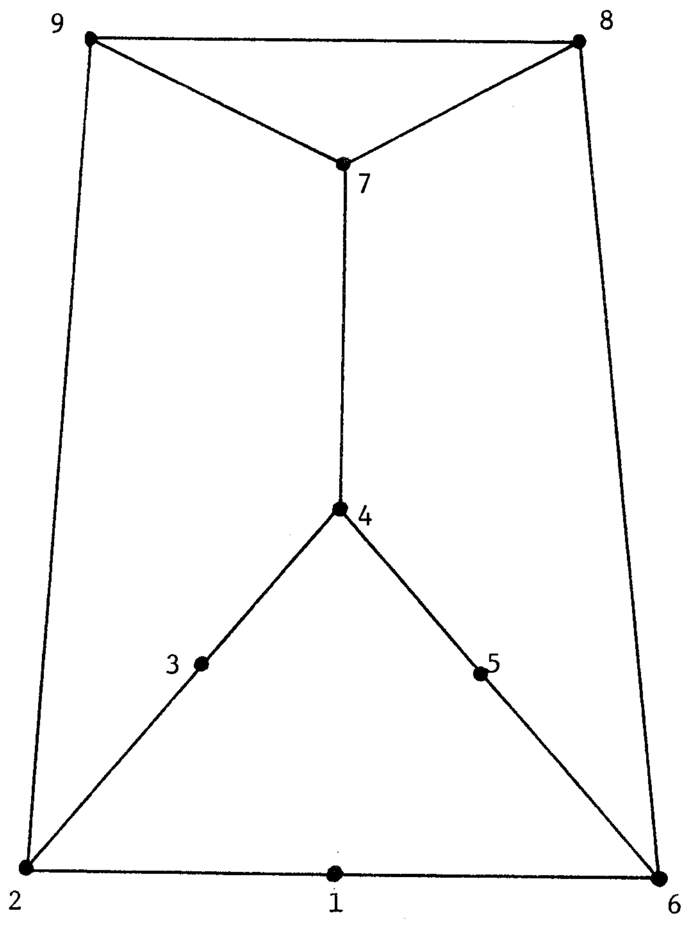


FIGURE 5.9 : ORIGINAL GRAPH



$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 6 & 5 & 4 & 3 & 2 & 7 & 9 & 8 \end{pmatrix}$$

ORBITS: {1}, {2,6}, {3,5}, {4}, {7}, {8,9}

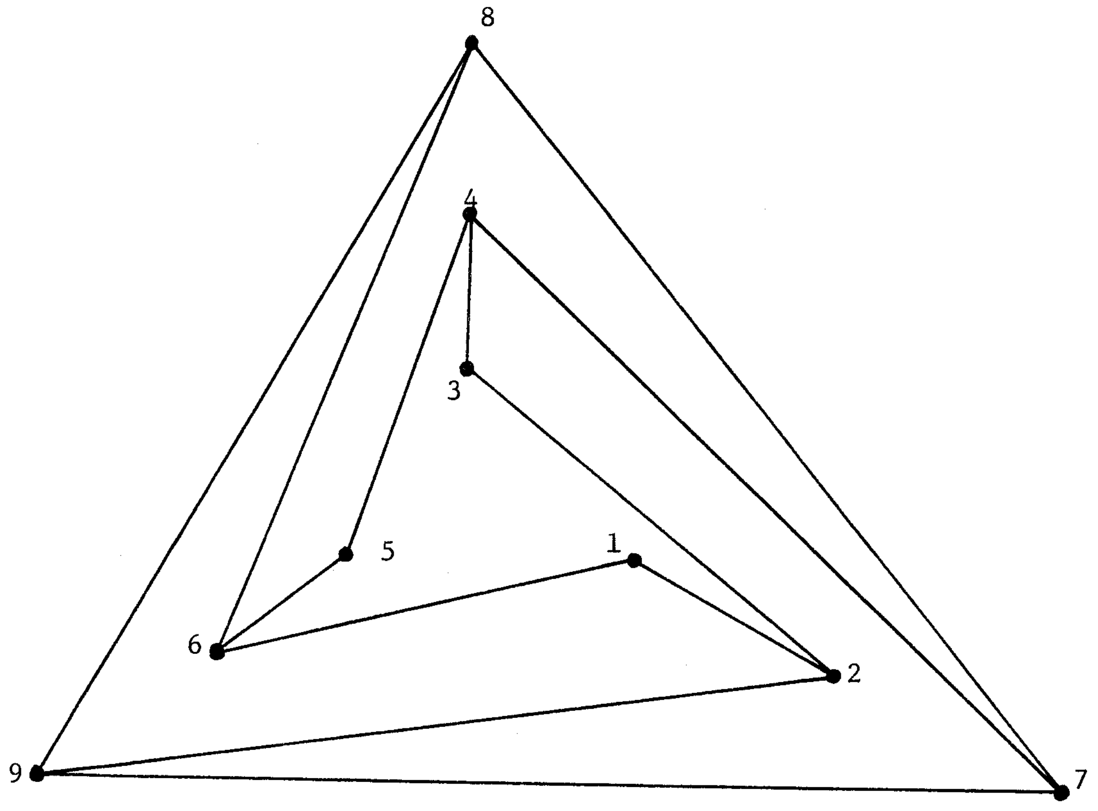
FIGURE 5.10 : GENERAL REPRESENTATION ROUTINE VERSION

resulting in a straightforward mirror-image display in only one plane. The diversity of the General Representation Routine is evidenced by the representation shown in Figure 5.11. This is the very same graph. The only difference is that now the representation is derived from an automorphism containing only rotational symmetries.

Simply describing the procedure for drawing these displays does not convey a feeling for its usefulness or indicate its weaknesses. However, Appendix C contains numerous displays produced using the General Representation Routine and the Tree Representation Routine. It is hoped that by examining this appendix the reader will obtain a clearer understanding of the points discussed in this chapter. An attempt is made, by means of suitable diagrams, to illustrate here some of the problems encountered in using these visual representation routines.

For example, it was indicated that the facility overcomes many problems by allowing the user to perform post-display operations on the representation. Figure 5.12 shows a representation created by the General Representation Routine. While the graph is actually quite simple, the nearly coincident edges shown in Figure 5.12 detract from the representation. Figure 5.13 shows the same graph after it has been transformed by postediting into a much more pleasing picture.

In Chapter III it was shown that representations derived from automorphisms which involved a high percentage of the vertices in non-trivial mappings (that is, the majority of the vertices were not fixed) would



$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 4 & 5 & 6 & 1 & 2 & 8 & 9 & 7 \end{pmatrix}$$

ORBITS: {1,3,5} {2,4,6} {7,8,9}

FIGURE 5.11 : VERSION STRESSING ROTATIONS

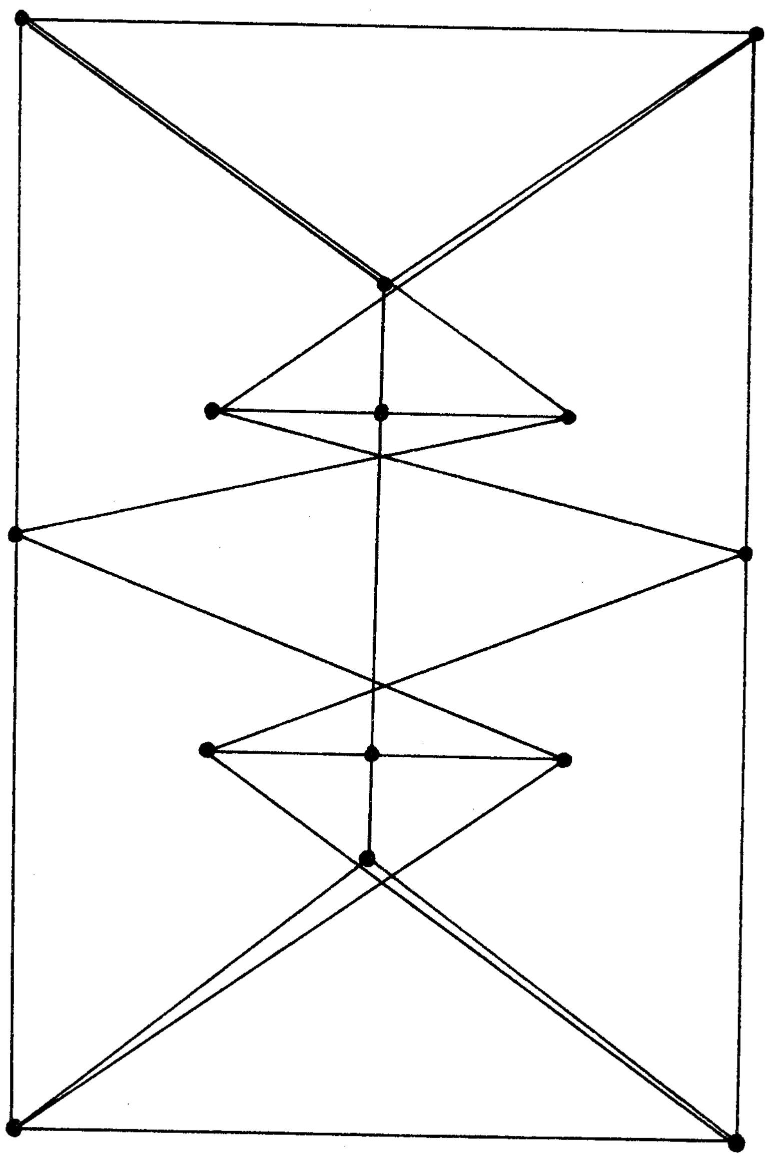


FIGURE 5.12 : CLUTTERED REPRESENTATION

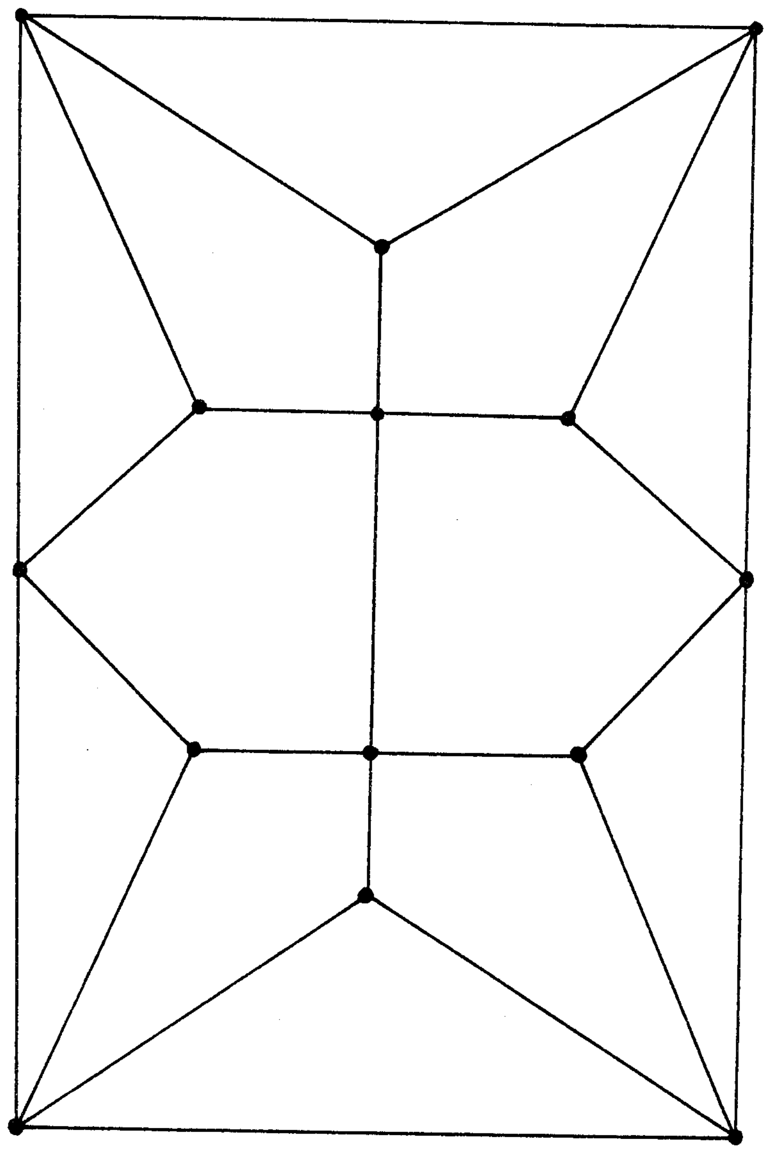


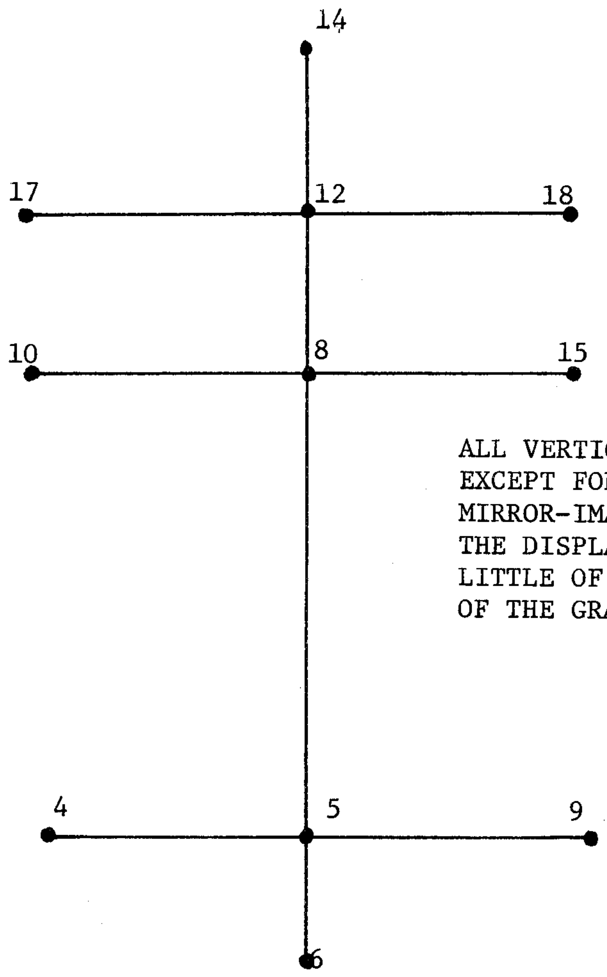
FIGURE 5.13 :  CLEANER REPRESENTATION

reveal more of the structure of the graph. This fact is stressed even more in the representations generated by the General Representation Routine since the representations depend solely on the symmetries in the automorphism. Figures 5.14 to 5.16 show several representations of the same graph obtained with the General Representation Routine for the specified automorphisms. The representations obtained from the automorphisms containing fewer fixed vertices are clearly more informative as well as more interesting. The second and third representations are more informative than the first because fewer fixed vertices are placed on axes of symmetry. The representation in Figure 5.16 uses only one axis of symmetry and as a result shows clearly all the edges in the graph.

In comparison the first two representations did not show all the edges connecting fixed vertices on the axes of symmetry. The last two representations are more interesting in a graph-theoretic sense because more and more of the tree structure of the graph is revealed, the representation in Figure 5.16 showing clearly that the graph is a tree.

Another aspect of the same problem is that the more fixed vertices in the automorphism, the larger the number of axes of symmetry that the General Representation Routine will use for the representation. For example, Figure 5.17 is a representation of a graph with a large number of fixed vertices. The same graph is shown in Figure 5.18 as it would appear from the side of the graphic display.[11] The large number of axes of symmetry results in a picture which is not nearly as pleasing

11. If it could be viewed through the side of the graphic display.

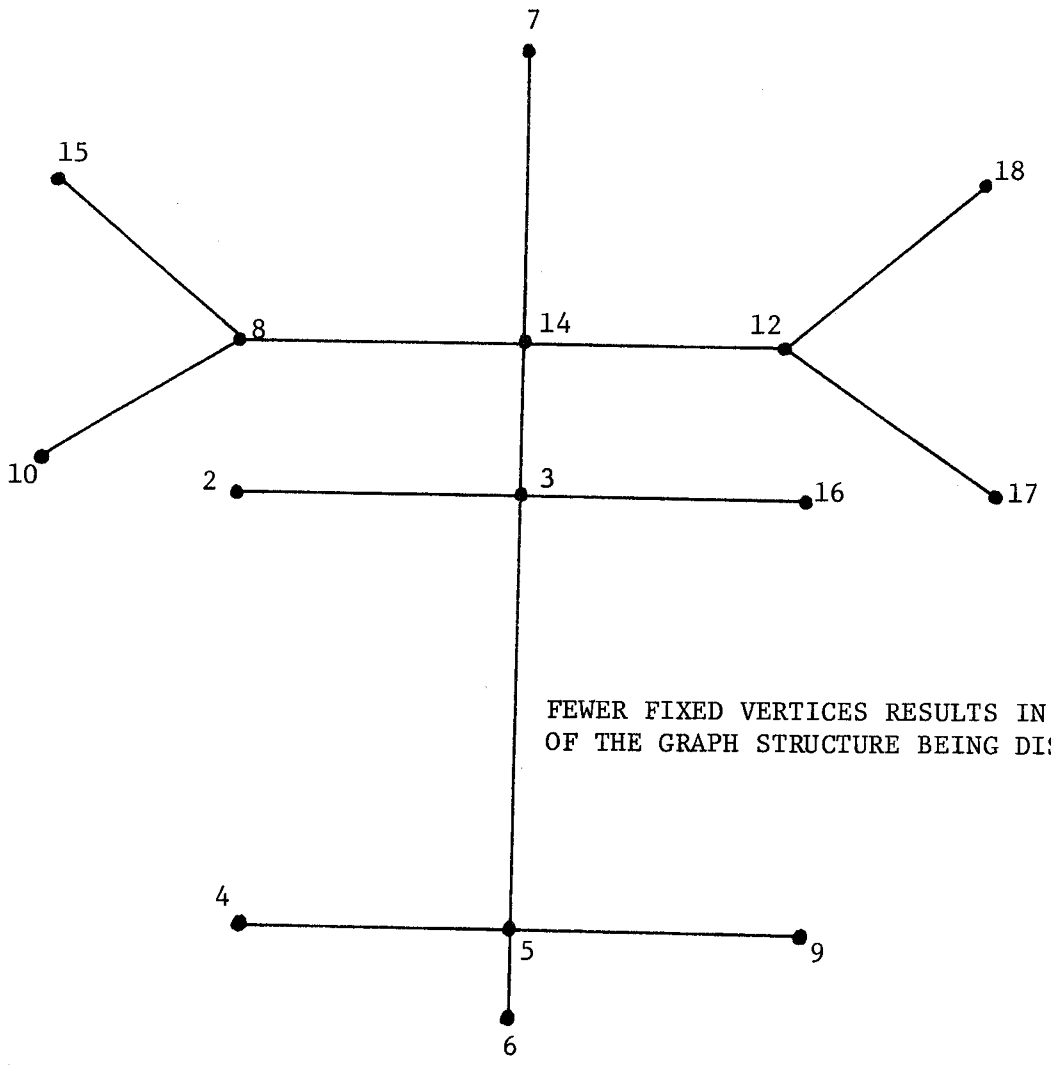


ALL VERTICES ARE FIXED EXCEPT FOR THE THREE MIRROR-IMAGE PAIRS SHOWN. THE DISPLAY SHOWS VERY LITTLE OF THE STRUCTURE OF THE GRAPH.

AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ 1 & 2 & 3 & 9 & 5 & 6 & 7 & 8 & 4 & 15 & 11 & 12 & 13 & 14 & 10 & 16 & 18 & 17 & 19 & 20 & 21 \end{pmatrix}$$

FIGURE 5.14

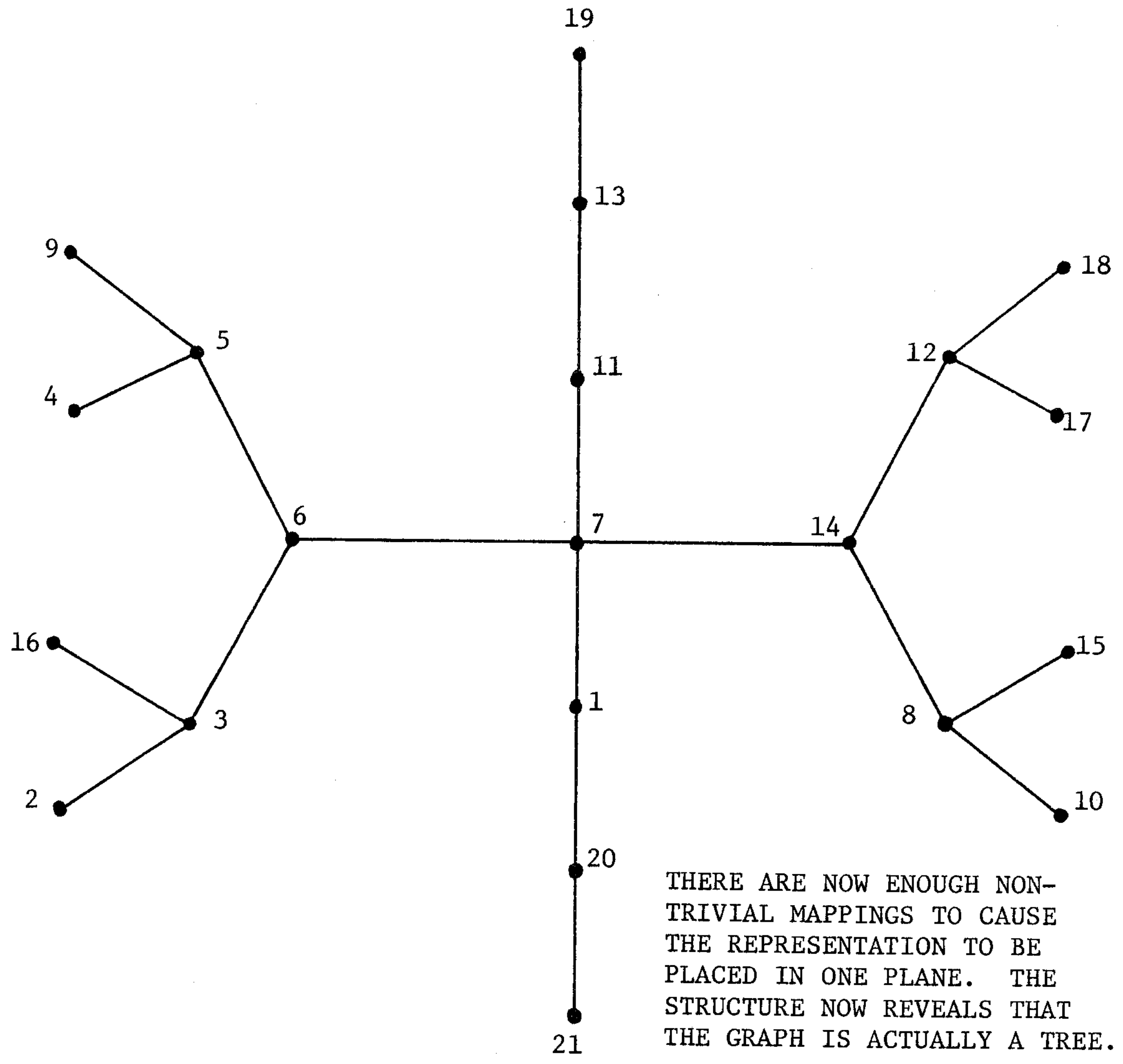


FEWER FIXED VERTICES RESULTS IN MORE OF THE GRAPH STRUCTURE BEING DISPLAYED.

AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ 1 & 16 & 3 & 9 & 5 & 6 & 7 & 12 & 4 & 17 & 11 & 8 & 13 & 14 & 18 & 2 & 10 & 15 & 19 & 20 & 21 \end{pmatrix}$$

FIGURE 5.15



THERE ARE NOW ENOUGH NON-TRIVIAL MAPPINGS TO CAUSE THE REPRESENTATION TO BE PLACED IN ONE PLANE. THE STRUCTURE NOW REVEALS THAT THE GRAPH IS ACTUALLY A TREE.

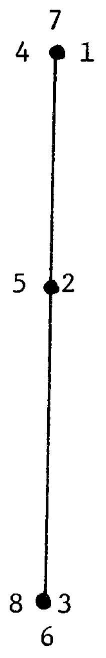
AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ 1 & 10 & 8 & 17 & 12 & 14 & 7 & 3 & 18 & 2 & 11 & 5 & 13 & 6 & 16 & 15 & 4 & 9 & 19 & 20 & 21 \end{pmatrix}$$

FIGURE 5.16

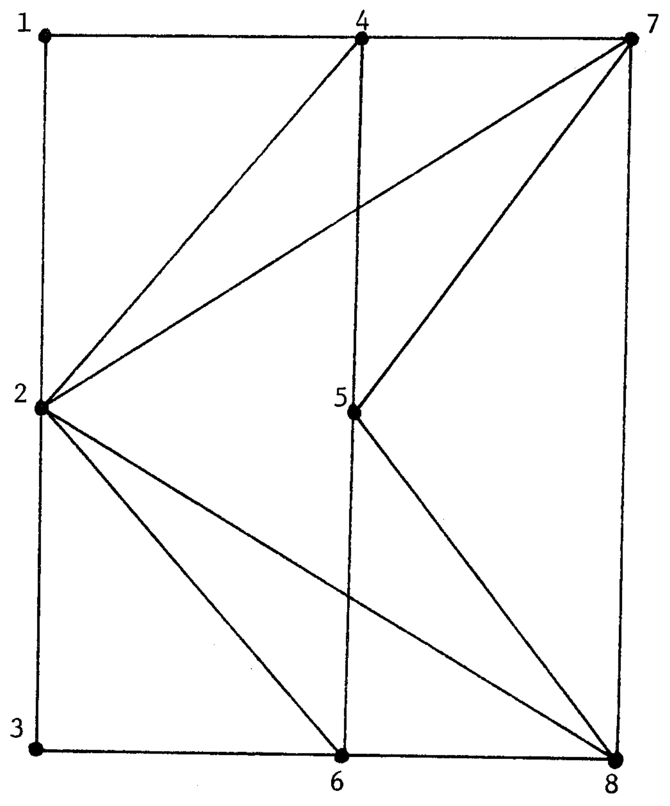
ADJACENCY MATRIX:

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



FRONT VIEW OF REPRESENTATION CREATED FOR IDENTITY AUTO-MORPHISM WHERE ALL VERTICES ARE FIXED.

FIGURE 5.17



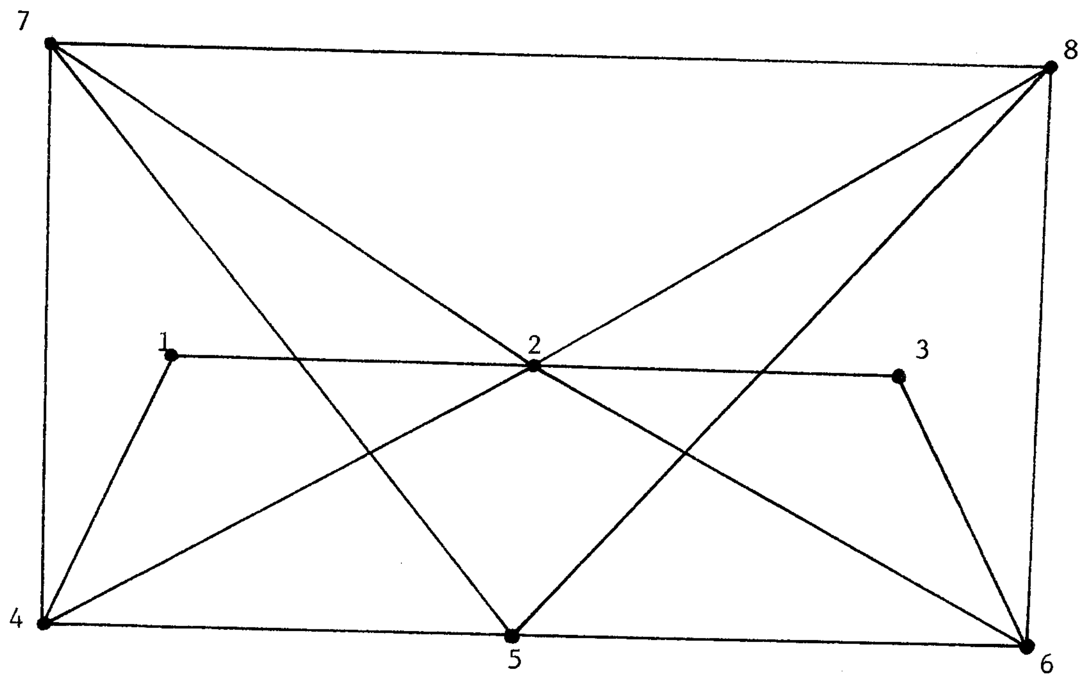
THE SIDE VIEW OF THE REPRESENTATION CREATED FOR THE IDENTITY MAPPING SHOWS THAT 3 AXES WERE NEEDED TO HANDLE THE FIXED VERTICES.

FIGURE 5.18

as that in Figure 5.19. Here the representation is derived from an automorphism with fewer fixed vertices.

It was necessary to rotate the graph in Figure 5.18 in order to get a clear picture of its shape as the representation routine created the display using several planes. While, as previously indicated, it may be necessary in certain cases to generate a three dimensional representation, it is also true that in many cases the General Representation Routine uses more planes than are really necessary. This does not imply that there is an error in the procedure. It simply means that of all the automorphisms in the automorphism group of a graph, some of the automorphisms will be better suited than others for generating visual representations. Thus, Figure 5.20 shows a picture of a graph where the General Representation Routine used more planes than necessary as evidenced by the representation of the same graph shown in Figure 5.21.

This problem of resorting to a three dimensional representation when two dimensions would suffice is closely related to the question of how well the General Representation Routine handles planar graphs. Unfortunately, the same problem may occur. Representations are created in which the vertices are placed in several planes when, because the graph is planar, one would hope that a single plane might have been used. It is difficult to fault the General Representation Routine for this as it has not been designed to worry about planar representation of graphs. In fact, the opposite approach has been used in that the procedure immediately seeks to use a non-planar representation in order to avoid problems with edge overlapping. In other words, the General Representation Routine is not as suitable for planar graphs because of the nature of its design.

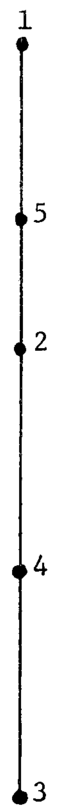


REPRESENTATION CREATED FOR THE AUTOMORPHISM BELOW SHOWS THAT ONLY ONE AXIS SUFFICES QUITE WELL FOR THIS GRAPH.

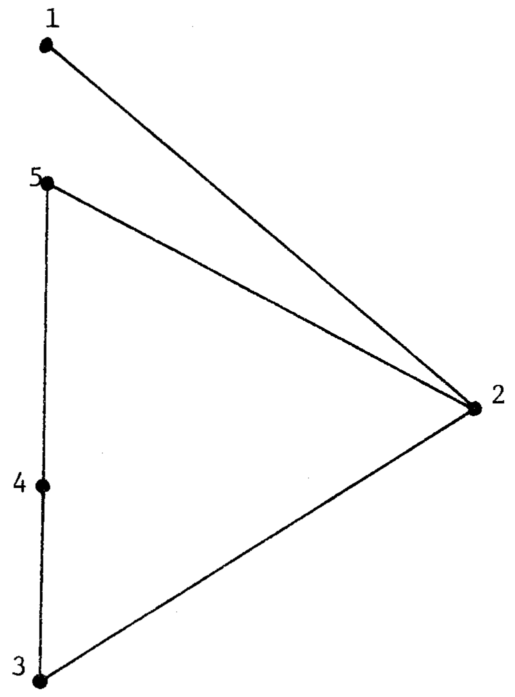
AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 6 & 5 & 4 & 8 & 7 \end{pmatrix}$$

FIGURE 5.19

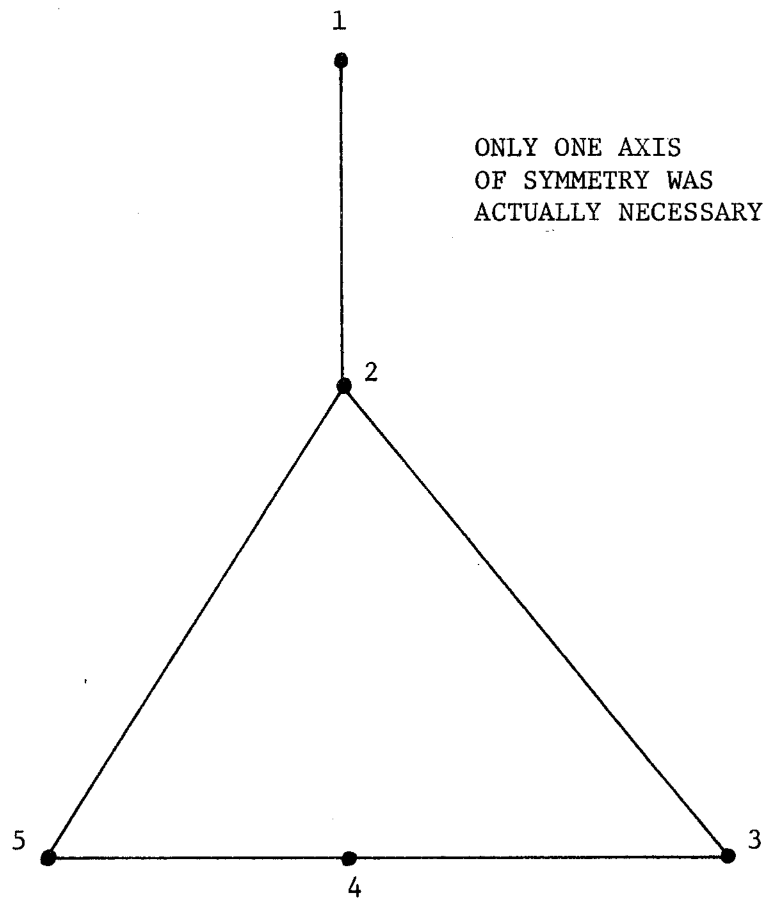


FRONT VIEW
- CONFUSING AND
  UNINFORMATIVE



SIDE VIEW
- TWO DIFFERENT AXES
  OF SYMMETRY USED

FIGURE 5.20



ONLY ONE AXIS
OF SYMMETRY WAS
ACTUALLY NECESSARY

AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 4 & 3 \end{pmatrix}$$

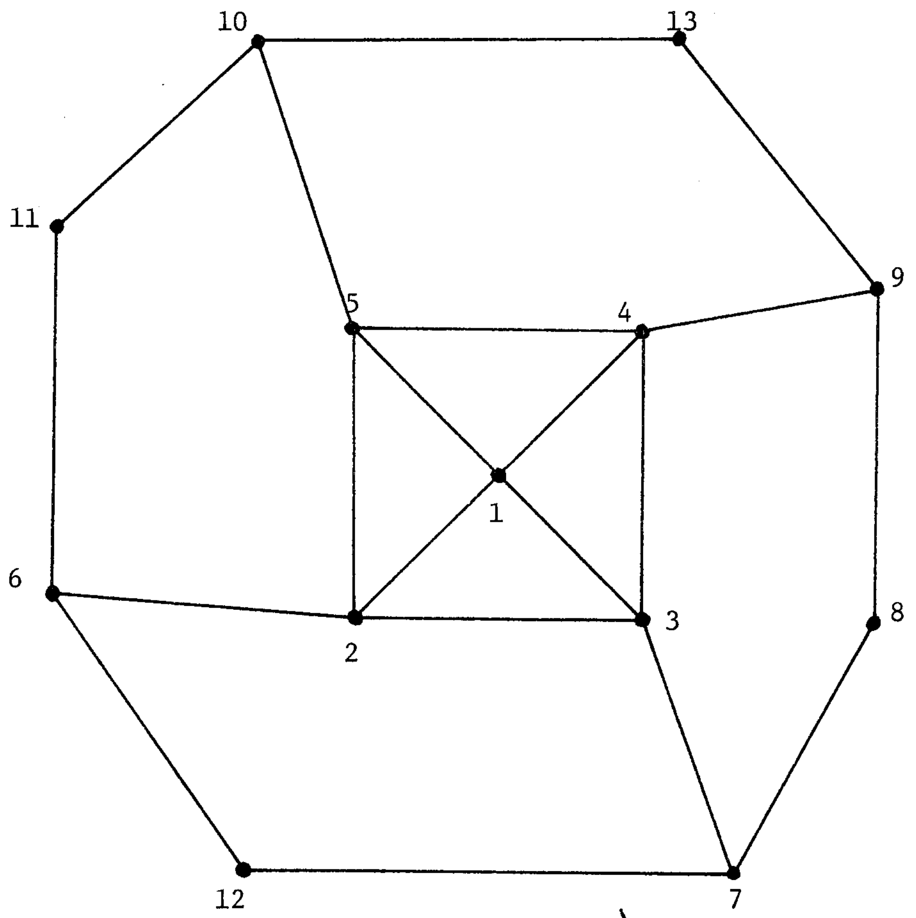
FIGURE 5.21

Figure 5.11 shows a representation of a graph which, according to the orbits of the automorphism mapping, should reveal the rotational nature of the symmetries. However, the actual rotations are not nearly as clearly displayed as they are in some representations. For example, the representation in Figure 5.22 appears to be more rotational in nature. The problem is that the mappings of the automorphism group are mappings of the vertices. The automorphism mappings, by definition, preserve adjacency. However, the mapping of the edges in EG induced by an automorphism mapping does not necessarily correspond to the vertex mapping.

For example, in a graph with one edge and two vertices, the end-vertices of the edge, an automorphism exchanging the two vertices as in a mirror-image symmetry induces a mapping of the edge onto itself. Thus, a mirror-image or rotational symmetry in an automorphism does not imply a corresponding mirror-image or rotational symmetry of the edges incident to the vertices in the vertex symmetry.

The result as indicated by the two graph representations in Figures 5.11 and 5.22 is that a representation created by the General Representation Routine may suffer because the display is based on an automorphism mapping of the vertices and does not, at least not directly, take into account what happens to the edges incident to these vertices under the mapping.

The above discussion shows clearly that the General Representation Routine does not succeed in handling all classes of graphs equally well. In fact, some classes of graphs it does not handle at all well. However, it should be just as evident that in the majority of cases it does succeed



$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 1 & 3 & 4 & 5 & 2 & 7 & 9 & 13 & 10 & 6 & 12 & 8 & 11 \end{pmatrix}$$

ORBITS: {1}, {2,3,4,5}, {6,7,9,10}, {8,13,11,12}

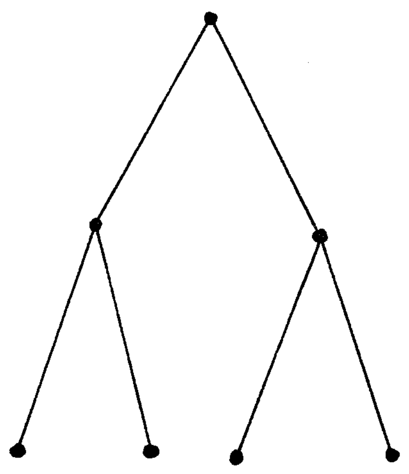
FIGURE 5.22 : ROTATIONAL GRAPH STRUCTURE

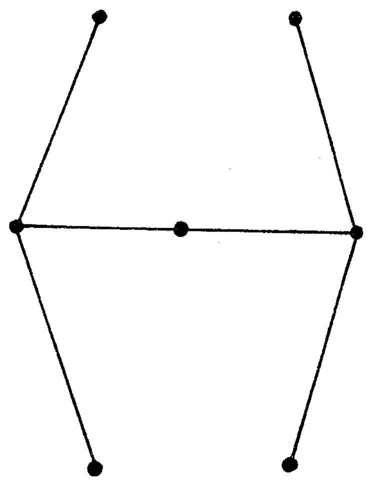
in providing a logical basis upon which the user can build his own picture of the graph. That is, if the routine generated picture is unsuitable, then at least the user has a good starting point from which to create his own picture.

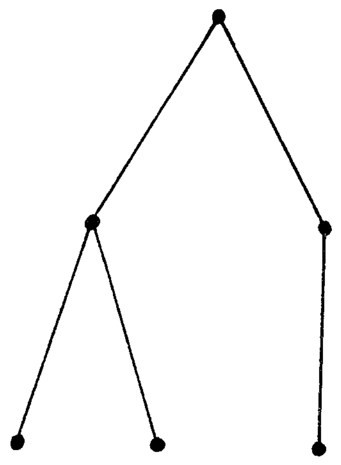
It is impossible to list all the classes of graphs which are 'good' graphs and those which are 'bad' graphs in terms of the success the General Representation Routine has in displaying them. However, there is one very large and important class of graphs for which the General Representation Routine appears almost totally unacceptable. This is the class of graphs which are trees. The General Representation Routine generates displays for trees which rarely, if ever, have a form suitable for portraying a tree. For example, Figures 5.23 and 5.24 show several representations created by the General Representation Routine for very simple trees. Owing to this rather serious failing of the General Representation Routine and because trees are important in many graph-theoretical applications the decision was made to investigate the possibility of implementing a separate tree representation procedure.

## Tree Representation Routine

Figure 5.3 showed that the user can select either the General Representation Routine or the Tree Representation Routine. The Tree Representation Routine is intended for the use of anyone working with trees. The graph must be a tree since the routine will not work otherwise. Moreover, the tree centre heuristic must have been invoked so that the identity of the tree centre or bicentre is known. The following is a description of a simple tree representation procedure. The procedure


ORIGINAL TREE


GENERATED REPRESENTATION


ORIGINAL TREE

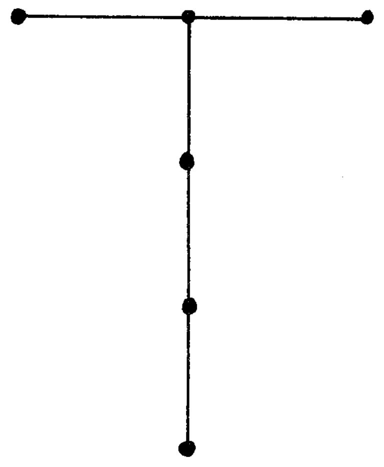
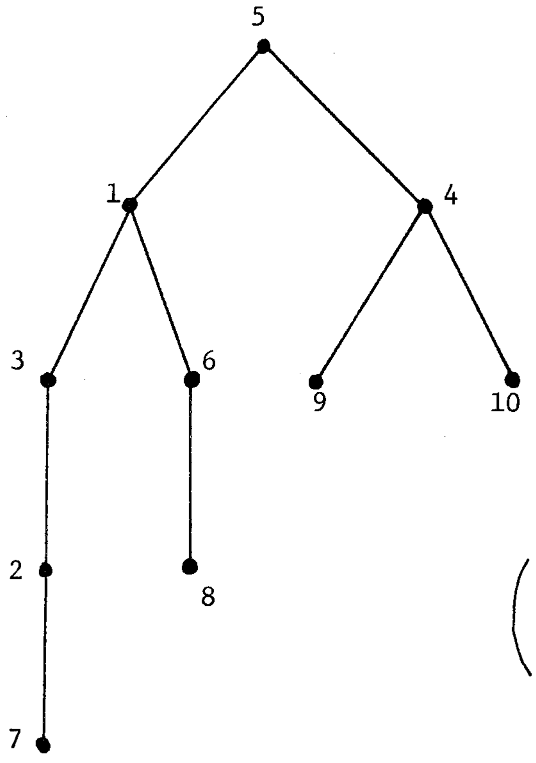

GENERATED REPRESENTATION

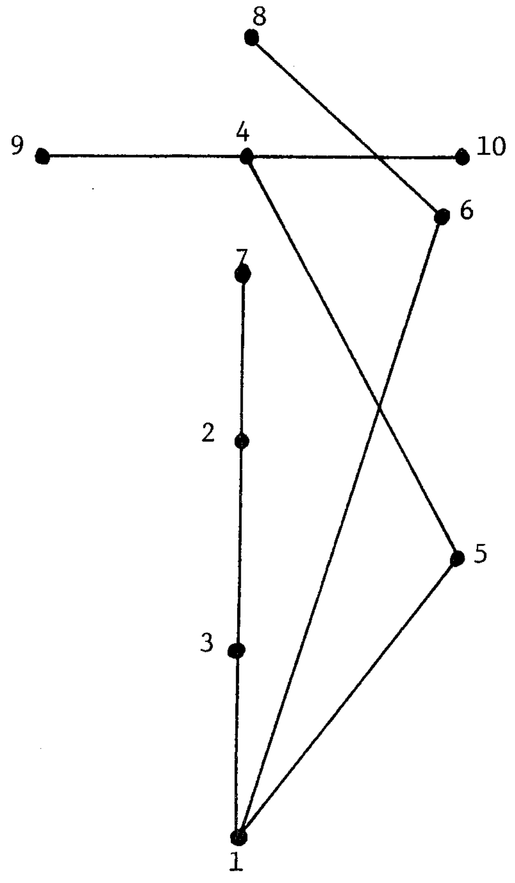
FIGURE 5.23



ORIGINAL TREE

ONLY NON-TRIVIAL AUTOMORPHISM:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 10 & 9 \end{pmatrix}$$



GENERATED REPRESENTATION

FIGURE 5.24

will be complicated later by the necessity of avoiding intersecting edges in the picture of a tree.

Tree Representation Routine

1) Use the centre or bicentre of the tree as the root of the tree in the display.

2) Partition the vertex set of the graph into sets according to the distance of each vertex from the centre or bicentre. There will be $\ell+1$ "levels" in the tree representation, where $\ell$ = the farthest distance of any vertex from the centre or bicentre. Let $n_i$ = the number of vertices at level i.

3) Assign y co-ordinate values to each vertex according to the level in which the vertex is found. The z co-ordinate for each vertex is set at 2048 in order to place every vertex in the tree on a plane in the centre of the three dimensional cube in which graphs are defined. This is possible since a tree is planar. The plane is then divided into $\ell+1$ equidistant levels along the y axis in order to spread the tree across the whole screen.

4) Let e = the number of end-vertices in the tree + c, where c = 1 if the tree has a centre and c = 2 if the tree has a bicentre. Divide the plane into e+1 equidistant increments along the x axis. The method being used here is similar to that described in Chapter II for Wolfberg's Tree Layout Routine. The final x co-ordinate values are assigned to the vertices of the tree by spacing the end-vertices and the centre or bicentre equidistantly across the screen and working from

the lowest level of the tree to the top, locating each vertex, v, at level i midway between all the successors of v at level i+1.

So far this procedure seems quite straightforward and just about what one might expect. In fact, Wolfberg's routine is essentially the above procedure with one difference. Wolfberg's routine assumes that the vertices are to be placed in the same order at each level as they originally were in when the graph was created on the screen by a user. The Tree Representation Routine cannot make this assumption. It must determine the ordering of the vertices in the graph without any prior information on how the user would like the vertices displayed. In addition, the vertices must be ordered so that the tree representation does not contain any intersecting edges. Finally, the visual representation of the tree must reveal clearly the mirror-image and rotational symmetries in the tree structure.

In order to isolate the various symmetries in the current tree automorphism the representation routine has been designed so that it moves mirror-image symmetries to the outside of the tree display. The rotational symmetries and fixed vertices are placed on the inside of the representation. This process begins with the ordering of the vertices on the first level. It is necessary to remember in the discussions below that the object is to order the vertices at each level. Once the vertex ordering has been determined the actual layout of the tree is handled quite simply by a routine similar to Wolfberg's.

## First level of the Tree

5) Let I be a location counter which is initially set to 1. Let O(v) be the order or position of vertex v at whatever level of the tree

v is in.[12] For every mirror-image pair of vertices (v,w) in the automorphism which are in the first level of the tree, order the vertices as follows:

$$O(v) = I, \quad O(w) = n_1 + 1 - I.$$

Increment I by 1 and repeat the process for the next mirror-image pair.

In this manner the mirror-image vertices are ordered so that when the tree is finally displayed the mirror-image symmetries are on the outside of the tree. The idea behind this is that were there no rotational vertices or fixed vertices (other than the root vertex if the tree has a centre vertex) in the automorphism, then the display would show the mirror-image symmetries just as if there was an axis of symmetry passing through the centre of the tree. The representation in Figure 5.25 is an example of such mirror-image symmetry.

For each rotational symmetry at level 1 order the vertices in the following manner:

Let S = the number of vertices in the rotation.

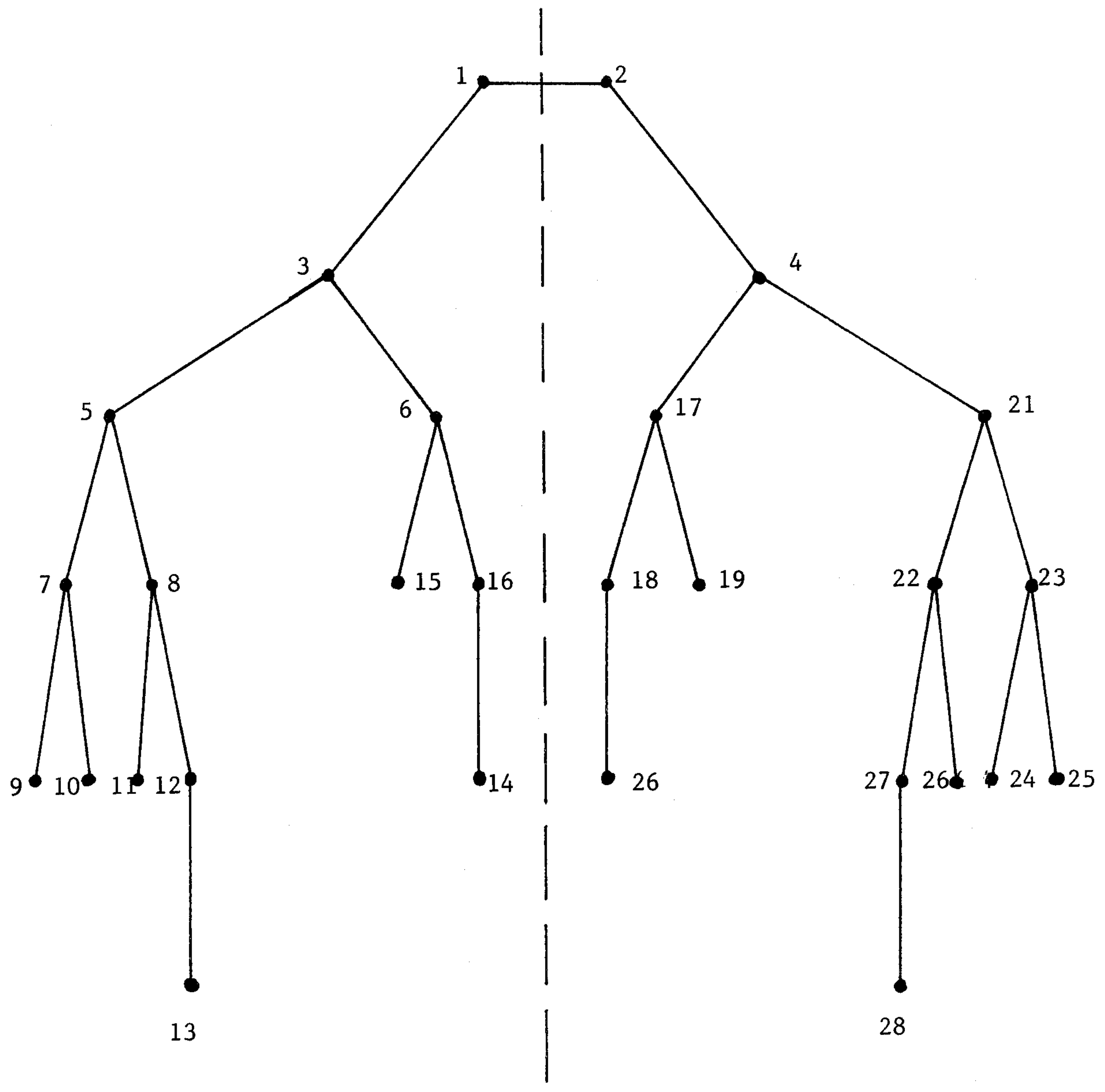
Let $(v_1, v_2, \ldots, v_S)$ be the set of vertices in the rotation.

Set $O(v_i) = I + i - 1$.

Reset $I = I + S$.

This process is repeated for every rotation at the first level. Thus, the rotational symmetries are placed from left to right after

12. That is, the vertex set of the graph has been partitioned into sets of vertices representing each level of the tree. The problem now is to change these sets into ordered lists by ordering the vertices at each level according to the left to right order in which they will be displayed on the screen.



ORBITS OF AUTOMORPHISM : {1,2}, {3,4}, {5,21}, {6,17}, {7,23}, {8,22}, {9,25}, {10,24}, {11,26}, {12,27}, {13,28}, {14,20}, {15,19}, {16,18}.

FIGURE 5.25 : MIRROR-IMAGE SYMMETRY IN A TREE REPESENTATION

the last mirror-image symmetry on the left side of the tree representation. The reason for this is that rotational symmetries are not as naturally related to trees as are mirror-image symmetries. That is, when one thinks of the structure of a tree one usually does not visualize it in terms of rotations.

One possible alternative to the method just used would be to space the rotational vertices equidistantly on the inside of the tree display. This would involve considerably more calculation, especially when it was necessary to order the vertices of several rotations. The primary consideration, however, was that the rotational symmetries in the final display are such that in another automorphism the vertices in the rotational symmetries are in mirror-image symmetries. What this means in terms of the tree display is that normally the user can, by postediting the display, picture the rotational symmetries as mirror-image symmetries. Keeping the rotational symmetries together in one area makes it easier for the user to locate the rotational symmetries and then do something about them if he wishes.

Let F = the number of fixed vertices at level 1. The fixed vertices are assigned the orders I+1, I+2, ..., I+F-1. This means that the fixed vertices are placed from left to right after the rotational symmetries. Unfortunately, owing to the nature of a tree it is not possible in most cases to place all the fixed vertices on an axis of symmetry as was done in the General Representation Routine.

It now appears as if a representation has been derived as a way has been found to use the symmetries in the automorphism to order the vertices in the graph. Now all that remains is to apply the above scheme to each level of the tree and then use the resultant orderings to create the tree display. Unfortunately, when such an attempt is made the disastrous results illustrated in Figure 5.26 can be generated.

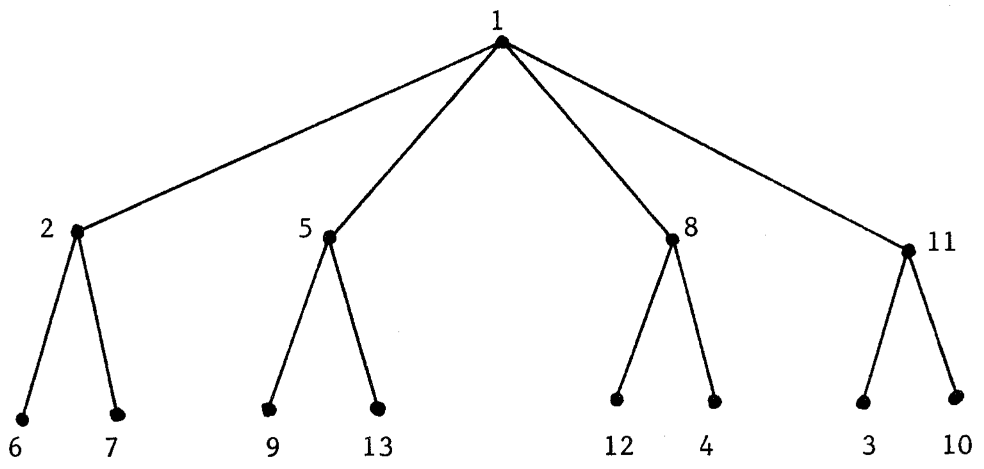
What happened, of course, was that the ordering at the first level was ignored when the vertices at the second level were ordered. Thus, it is necessary to remember what has happened in the upper level orderings as the vertices at the lower levels of the tree are ordered. The same type of ordering as outlined in step 5, should be applied but with the new restriction that what happens in level i+1 is dependent upon the ordering in level i.

Ordering of Vertices in Level i, i>1

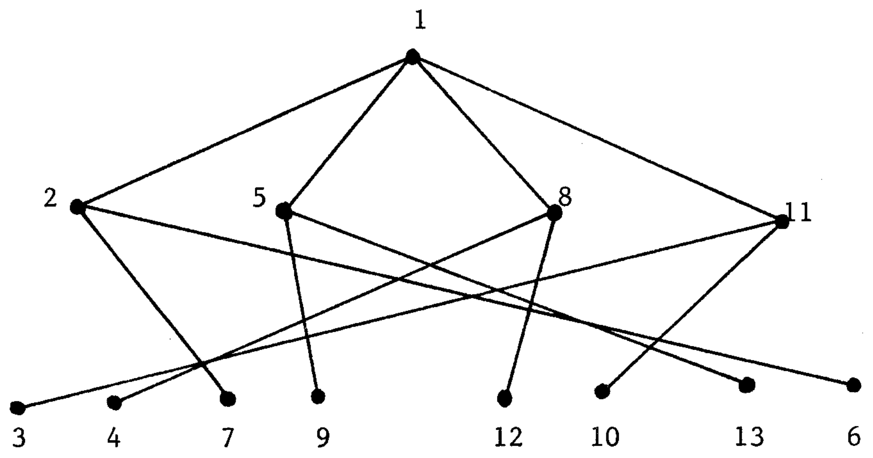
6) Reset the location counter I to 1. Examine the vertices at level i in the order determined by the ordering of the vertices at level i-1. That is, first examine those vertices at level i which are adjacent to the first vertex at level i-1. Continue with the vertices at level i adjacent to the second vertex at level i-1 and so on until the ordering for level i is complete. In this way it is insured that the vertices are ordered so as to prevent the intersecting edge problem (see Figure 5.26) caused by ignoring the order of the vertices at level i-1.

The problem now is to order whatever set, V, of vertices at level i (adjacent to a vertex v at level i-1) is currently under examination. This problem can be subdivided into two cases:

i) The vertex v at level i-1 is a fixed vertex.



ORIGINAL GRAPH



RESULTANT GRAPH

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 1 & 11 & 6 & 13 & 8 & 3 & 10 & 5 & 12 & 7 & 2 & 9 & 4 \end{pmatrix}$$

FIGURE 5.26 : POOR TREE DISPLAY

If v is fixed then there is no problem about preserving the correspondence between vertex v and its image under the automorphism mapping.[13] Preserving the correspondence between a vertex, v, and its image, w, under an automorphism mapping simply means that the layout imposed on the subtree having v as its root vertex must be repeated for the subtree having w as its root vertex. In other words, given some particular ordering of the successors of v, the successors of w (which are, of course, the image vertices under the automorphism mapping of the successors of v) must have the same relative ordering. For fixed vertices the same procedure that was used for level 1 vertices is used to add this set of vertices to the ordering of vertices at level i. The only difference now is that instead of using $n_1$ in the ordering formula the number of vertices in the set V must be used. Once the ordering of V is determined the vertices are added in this order to the list of ordered vertices at level i.

ii) v is not a fixed vertex:

In this case vertex v is involved in either a mirror-image mapping or a rotational mapping at level i-1. But if a vertex v is mapped onto a vertex w, then the successors of v (at level i) must map onto the successors of w(at level i). Thus, the vertices in V are involved in a mirror-image or rotational mapping.[14] Moreover, if the preimages of these vertices have already been ordered then the ordering of the vertices

13. Being fixed v is, of course, mapped onto itself by the automorphism.

14. Because, if the root vertex of a subtree is in a mirror-image or rotational mapping, the whole subtree is in the same type of mapping.

in V must correspond to this ordering. However, if the preimages of the vertices in V have not been ordered then the vertices in V can be given any ordering. That is, within the interval in which these vertices are to be added to the existing ordering of level i, their actual orderin is irrelevant.

Thus, examine the vertices in V and if their preimages have not been ordered assign an arbitrary ordering to the elements of V. If, however, the preimages have been ordered then the following scheme for ordering the vertices must be used:

Let $V = \{v_1, v_2, \ldots, v_{|V|}\}$.

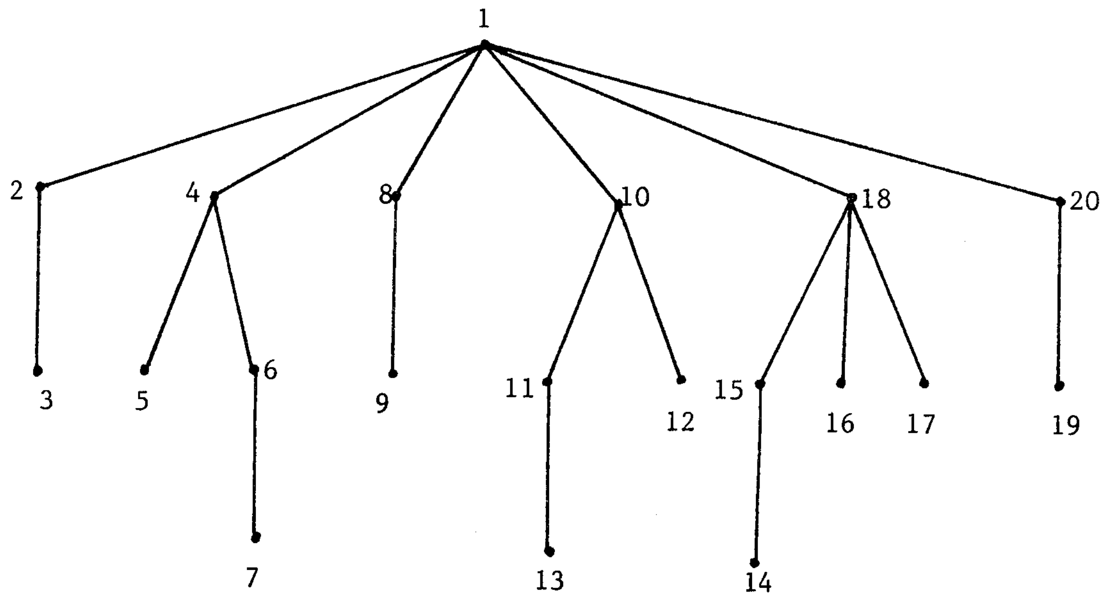
Create a $|V| \times 3$ array where row i contains the elements $v_i$, the preimage of $v_i$ and whatever numerical order has been assigned to the preimage of $v_i$. Reorder the rows in the array so that the elements in the third column are in increasing order. The order of the elements in the first column is now the order in which the vertices in V are to be added to the end of the current ordering of level i. This reordering makes the vertices in V correspond to the same relative order assigned to their preimages.

Once all the vertices at level i have been ordered, go to step 7.

7) If $i<\ell$, then set $i = i+1$ and go to step 6. That is, order the next level. If $i = \ell$, then go to step 8.

8) The graph has been ordered and the same type of layout routine Wolfberg used to create tree representations is used to display the graph.

Example 5.1



Automorphism:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ 1 & 8 & 9 & 10 & 12 & 11 & 13 & 20 & 19 & 4 & 6 & 5 & 7 & 14 & 15 & 17 & 16 & 18 & 3 & 2 \end{pmatrix}$$

Orbits:

(1), (14), (15), (18) - Fixed vertices

(4,10), (5,12), (6,11), (7,13), (16,17) - Mirror-image

(2,8,20), (3,9,19) - Rotations

In order to illustrate how this procedure works and to help the reader interpret the tree representations shown in Appendix C, the operation of the procedure on the automorphism in Example 5.1 will be described in detail.

Root vertex = 1. (found by the centre test)

<u>Level 1</u>:

$n_1 = 6$, $S = \{2, 4, 8, 10, 18, 20\}$. The only mirror-image pair is (4,10) so $O(4) = 1$ and $O(10) = 6$. The only rotation is $\{2, 8, 10\}$ so $O(2) = 2$, $O(8) = 3$ and $O(20) = 4$. The only remaining vertex is fixed so $O(18) = 5$. Thus $O(\text{level } 1) = (4, 2, 8, 20, 18, 10)$, rearranging the set of vertices at level 1 to obtain an ordered list using the orders just assigned to each vertex at level 1.

<u>Level 2</u>:

The successors of vertex 4 are the vertices 5 and 6. But the preimages of 5 and 6 [15] are not yet ordered so that an arbitrary ordering can be assigned, say $O(5) = 1$ and $O(6) = 2$.

The successor of vertex 2 is vertex 3. Since vertex 2 has only one successor at level 2, $O(3) = 3$ can be set without having to consider whether or not vertex 2 is fixed. This is true whenever a vertex has only one successor at the next level.

Similarly, vertex 9 is the only successor of vertex 8, so $O(9) = 4$ and, again, 19 is the only successor of 20 so $O(19) = 5$.

15. The preimages of 5 and 6 are 12 and 11 respectively since the automorphism maps 12 onto 5 and 11 onto 6.

The successors of 18 are vertices 15, 16 and 17. Vertex 18 is fixed so it is necessary to examine the types of mappings vertices 15, 16 and 17 are involved in. 15 is fixed, but 16 and 17 are a mirror-image pair so that O(16) = 6, O(17) = 8 and O(15) = 7.

The successors of 10 are 11 and 12. Vertex 10 is not fixed. Moreover, the preimages of 11 and 12 are already ordered. Hence, build the array:

| | | |
|---|---|---|
| 11 | 6 | O(6) = 2 |
| 12 | 5 | O(5) = 1 |

The array is sorted to produce:

| | | |
|---|---|---|
| 12 | 5 | O(5) = 1 |
| 11 | 6 | O(6) = 2 |

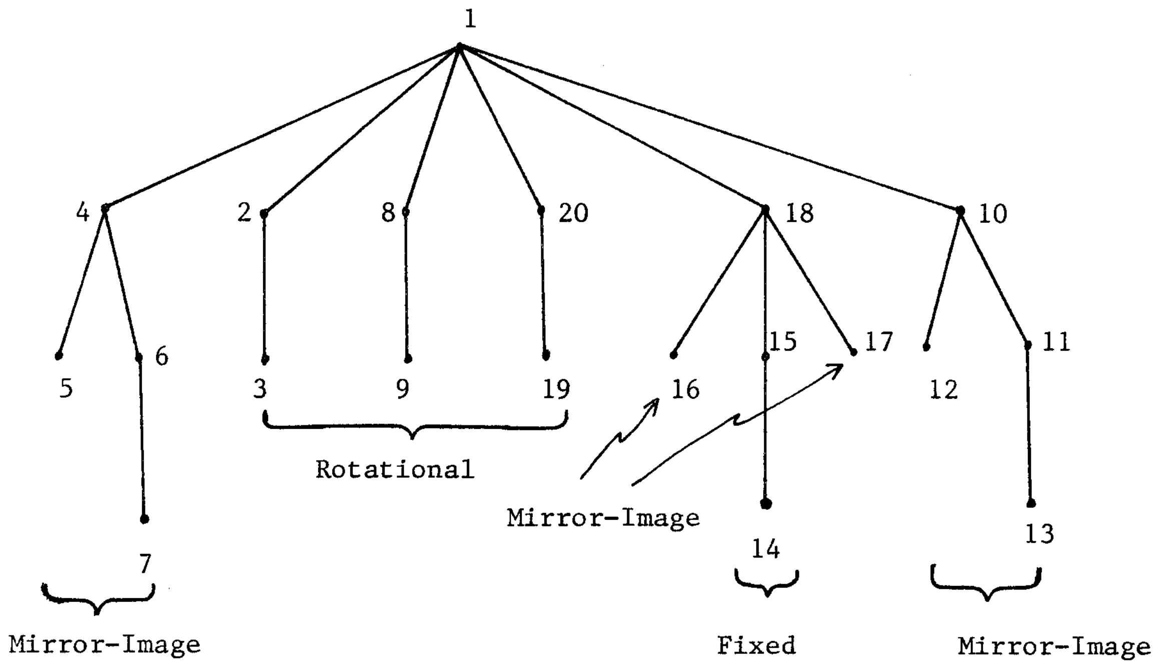
Therefore, O(12) = 9 and O(11) = 10.

Thus, O(level 2) = (5, 6, 3, 9, 19, 16, 15, 17, 12, 11).

<u>Level 3</u>:

Only three vertices at level 2 have successors at level 3 and all three vertices have only one successor. The successors of vertices 6, 15 and 11 are 7, 14 and 13 respectively. Hence, O(7) = 1, O(14) = 2 and O(13) = 3.

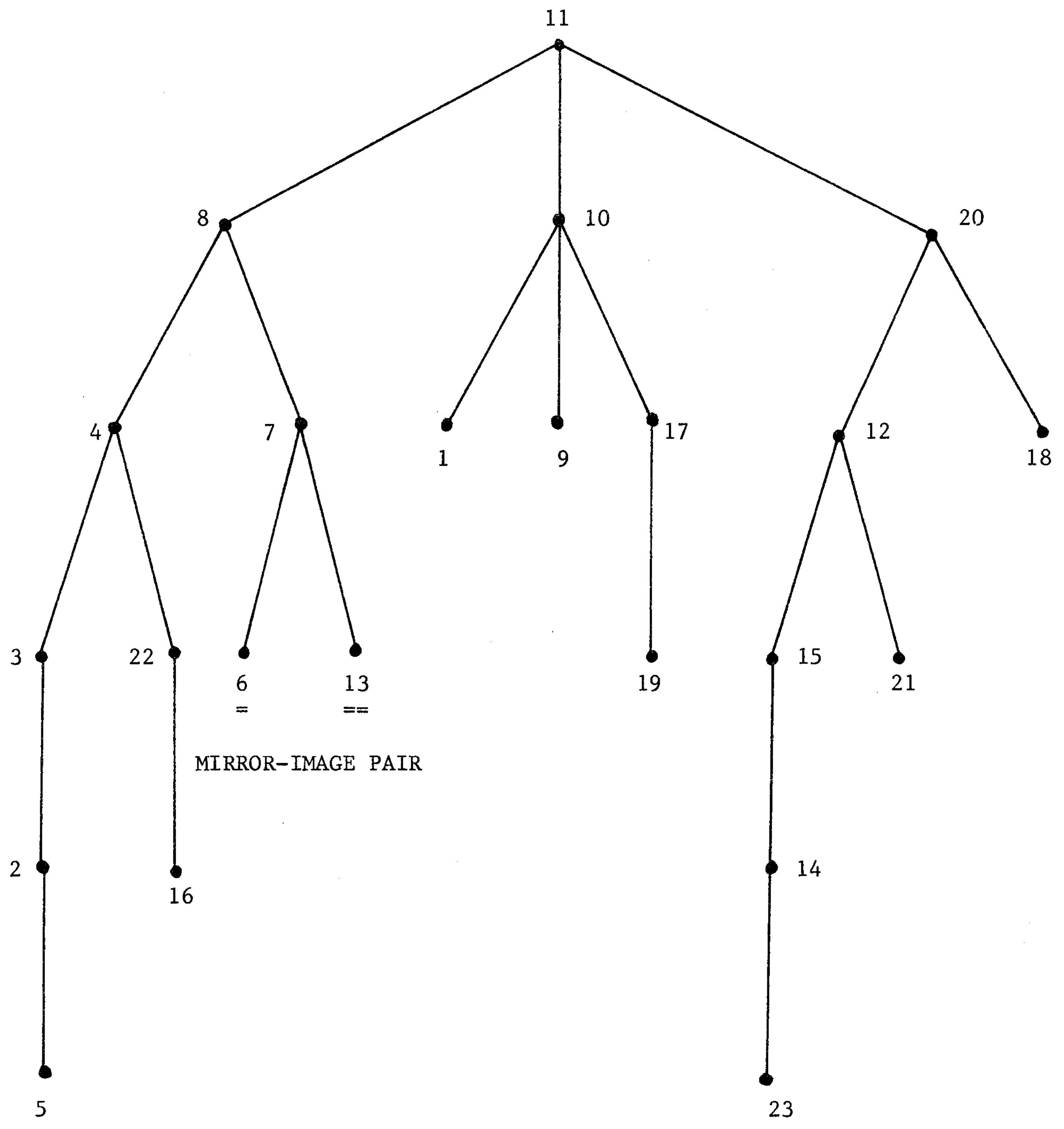
The complete graph is ordered so that after applying the layout routine the following representation is obtained:



Note that the above representation shows a mirror-image mapping at level 2. The two vertices in this mirror-image pair are adjacent to a fixed vertex at level 1. This type of situation occurs frequently in the representations generated by the Tree Representation Routine.

Once the tree representation has been completed the user is again allowed to perform postediting. For example, he might wish to move the branches of the rotation in the above example in order to display them as mirror-image symmetries. However, the user should not invoke the mirror-image edit option to move mirror-image pairs which do not have the centre of the screen as their axis of symmetry since this routine assumes the axis to be in the centre. The actual postediting facilities for trees are rather limited. For example, there is no way to relocate all the vertices in a branch except by moving each vertex individually. Special postediting options for trees were not felt to be necessary because it was found in general the Tree Representation Routine produced excellent displays.

The reader can judge for himself the degree of success of the Tree Representation Routine by examining the tree representations included in Appendix C. The only real problem one might encounter in using the tree representation routine is the problem previously discussed where an automorphism contains a large number of fixed vertices. In such cases the representation routine still creates a very good display of the tree as evidenced by the representation shown in Figure 5.27. The problem is that the symmetries that would be revealed by another automorphism of the tree are not available to use to balance the branches of the tree. However, this problem is easily avoided by using only automorphisms with a small percentage of fixed vertices to generate the representations.



$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 1 & 2 & 3 & 4 & 5 & 13 & 7 & 8 & 9 & 10 & 11 & 12 & 6 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \end{pmatrix}$$

FIGURE 5.27 : AUTOMORPHISM FIXES ALL VERTICES EXCEPT PAIR (6, 13)

## Alternative Visual Representation Schemes

The two visual representation schemes that have actually been implemented using the GSYM system are radically different in nature. As has been seen, the General Representation Routine frequently fails to produce a satisfactory representation in the sense that it is either unwieldy or is unsuited to the user's application. On the other hand, the Tree Representation Routine generates extremely good displays for any graph which is a tree. Both routines attempt to portray the structure of the graph in terms of mirror-image and rotational symmetry. The difference in the degree of success thus stems from the fact that the Tree Representation Routine knows a good deal more about the graph than does the General Representation Routine. Since the graph is a tree it knows that the graph has a planar representation, that it has a centre or bicentre that can be used as the root of the tree representation, that the vertices can be placed at equidistant levels from the root, that there are no polygons in the graph and so on. All of this information helps to determine the best way of displaying the symmetries in the graph.

One then questions whether there should be a concentration on designing visual representation routines for special classes of graphs or on attempting to improve the General Representation Routine. This is not an easy question to answer. One must not be misled by the above discussions as the Tree Representation Routine and the General Representation Routine are both extreme examples of representation routines.

One is almost totally naive about the nature of the graph while the other routine knows everything. There are very few classes of graphs such as trees which lend themselves so well to a visual portrayal. Instead of attempting to answer this complex question with a simple yes or no, some special representation routines and some improvements to the General Representation Routine will be considered. By doing this some of the problems involved in choosing a visual representation scheme will be illustrated.

None of the schemes about to be described have been implemented owing to the considerable amount of time required to prepare and program a visual representation routine. This, then, is the first tradeoff in the debate between general versus specific representation routines. With a general representation routine one hopes, at best, to be able to handle many if not all classes of graphs moderately well. A special representation routine, on the other hand, is designed to handle one class of graphs extremely well. The special routine may not be able to cope with any other class of graphs. Assuming that both types of routines take about the same amount of effort to implement, one must decide how to allocate programming resources. A user interested in a special application involving only one class of graphs would be foolish to implement a general representation routine. His time is better spent creating a routine to handle the type of graph he is using. If, however, one may encounter several types of classes of graphs, it would seem a better approach to implement a general representation routine or at least fashion a compromise routine based on any special properties that may be common to the several classes of graphs involved.

In general, improvements and changes to the General Representation Routine will take the form of display layout improvements, since any consideration of specific graph-theoretic properties cannot be added to the procedure without compromising the plan to have a completely general representation routine. These improvements in display methods for the graph may be very simple and some are not at all obvious. For example, in the course of deciding how to treat the rotational symmetries in the General Representation Routine it became necessary to decide whether each rotation should be on a separate plane or whether several rotations could be placed on the same plane. The final decision to use separate planes was based primarily on a desire to isolate the individual symmetries. In using separate planes to place the rotations one often creates a display where many edges connect the different planes. In order to see these edges in the proper perspective it is usually necessary to rotate the graph.

In the case where several rotations are placed in the same plane one is attempting to embed as many edges as possible in each rotation plane and thus reduce the number of connecting edges. A very simple example of this type of embedding is shown in Figures 5.28A and 5.28B. In Figure 5.28A it is seen how a pair of concentric triangles might look when they are embedded in the same plane. Using separate planes for these triangles would force the user to experiment with various rotations of the graph before obtaining a display such as that shown in Figure 5.28B. This is a very simple example and does not indicate the problems that could arise when several rotations were placed in the same plane. For example, Figure 5.29 illustrates how a slightly more complicated pair of rotationstreated in this manner results in a rather confusing display.

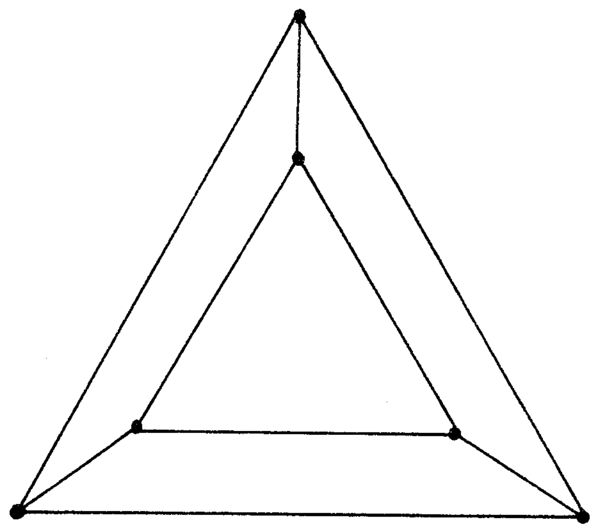


FIGURE 5.28A

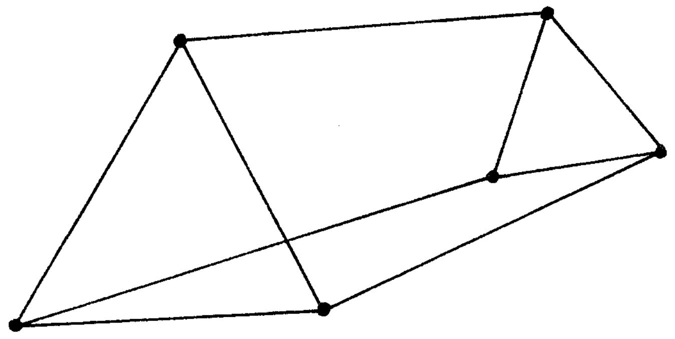


FIGURE 5.28B

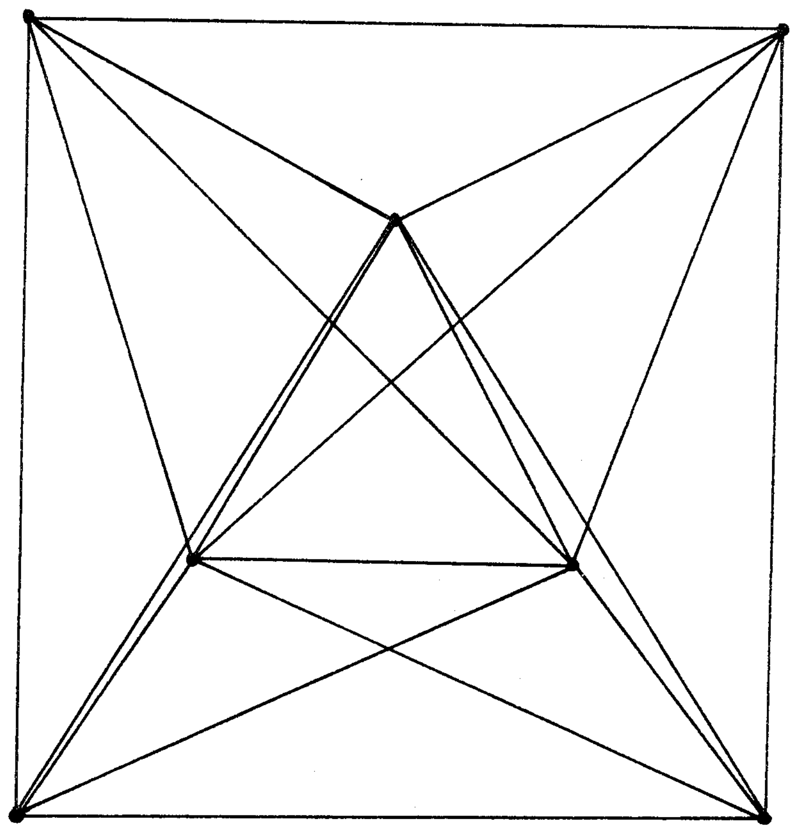


FIGURE 5.29 : CONFUSING ROTATIONAL REPRESENTATION

The reader can imagine just how cluttered a single plane could get with several such rotations on it. This is not to say that using individual planes provides a magical solution to the problem, but at least the user is able to separate the rotations from the interconnecting edges by rotating the graph.

Probably the most important improvement one could make to the General Representation Routine would be to devise a method whereby the elements of the graph could be distributed more evenly about the screen. This means that one would try to balance the distance between the elements and also the area enclosed by polygons of the graph. A small step in this direction has been made in the General Representation Routine where the user is required to specify the distance between the vertices of each rotational symmetry and the location of the first vertex in the rotation. The idea is that the user will thus be able to distribute the rotations equally about the screen and also be able to treat them as if they were concentric polygons. This is the method used to generate most of the General Representation Routine displays in Appendix C. As the reader can see from these representations this mode of attack works quite well in many cases.

However, this is strictly an ad hoc method. What would be really useful would be a method whereby the polygon areas would automatically be balanced or a means whereby the user could request that the screen area used to display specified parts of the graph be equalized or reduced or enlarged. Such a routine, in our opinion, would be more useful and have a better chance of success if it involved the user at some point in the process. Such a routine would be both simpler and more effective if guided by a user viewing the display and judging how well the routine is working.

The creation of special purpose representation routines takes the opposite approach to the same problem of obtaining a 'good' display of the graph. The idea is to find some special property of a class of graphs that can be interpreted visually in such a way as to dictate how the graph must be displayed on the screen. This was especially evident with trees where the properties of a tree indicated that the graph must have a planar representation with a root vertex and several levels of vertices below the root vertex. In essence, the special property sought is one which will impose a unifying approach over the steps that are taken to display the various symmetries in the automorphism group.

The most obvious special property that a graph might have is that it be planar, since all graphs can be partitioned into planar and non-planar graphs. The representation system could then be restricted to displaying only planar graphs. It is possible to embed such graphs in a single plane without having any intersecting edges and Tutte gives an algorithm for drawing such a graph in the plane.(xxiii) Unfortunately Tutte's algorithm tends to produce pictures where a large number of edges are drawn within a very small area. As a result, for most practical purposes, the algorithm proves to be unsatisfactory. Moreover, the algorithm does not attempt to represent the structure of the graph as given by the symmetries of the automorphism group. This in itself is sufficient reason for not using the algorithm.

The problem with the property of planarity is that it is too general since it does not indicate how one might represent the symmetries of the graph in a planar representation. When the display of mirror-image vertex pairs by the General Representation Routine was discussed it was shown how the procedure attempted to reduce the occurrence of intersecting edges. This, again, was an ad hoc treatment. A procedure for creating planar representations would probably use the regions of the graph to generate a display without intersecting edges. The display could be generated region by region, thusly preventing any possibility of intersecting edges. However, as there is no direct relationship between the vertex mappings of the automorphism group and the regions of the graph, it is not obvious how the symmetries of the graph could be represented using the regions of the graph to create the display.

The type of special property that is being sought is best illustrated by the special class of cubic, cyclically four-connected graphs considered in Chapter III. There it was shown how the special properties of this class of graphs could, in most cases, be used to impose a special type of overall structure on the displays of graphs in the class. In this case it was a ring structure form that was the basis for the representations. Curiously enough, this same type of ring structure seems intuitively well suited for representing both mirror-image and rotational symmetries. Unfortunately, there has not been time to pursue this possibility and perhaps generalize the ring structure to handle similar classes of graphs.

The types of considerations one must make before choosing or implementing a visual representation algorithm have been discussed.

There is no definitive answer as to which of the previous methods should be chosen; the answer must depend on the particular application or need. The two approaches discussed previously are not the only ways to tackle the problem of visual representation of graphs. The above methods have been discussed because both attempt to interpret directly the symmetries found in the automorphism group mapping. An alternative method which was considered does not directly utilize the symmetries of the automorphism. Instead it determined if the graph had a spanning tree. If such a spanning tree could be found then the Tree Representation Routine and an automorphism of the tree could be used to obtain a good display of the tree. The remaining edges of the original graph would be added to the display to create the graph representation. Since the Tree Representation Routine is capable of producing a good representation of a tree then it would be expected that the final representation of the whole graph should also be quite good.

Another rather novel idea (suggested by Z. Jordanov) is that one might treat the edges as springs with equal physical properties. One would choose several vertices of the graph as fixed points and then release the springs (edges) to see what form the graph took when the springs reached a state of equilibrium. The intuitive idea behind this is that the forces exerted by the springs would depend on the structure of the graph and the final equilibrium state should, therefore, reflect this structure. Even if this were true, the problems involved in choosing suitable fixed points and in calculating the forces exerted on each vertex appear to make this method impractical. However, it is a good example of the room that exists for experimentation and research into the question of how to obtain the 'best' visual representation of a graph.

# CHAPTER VI : IMPROVEMENTS, APPLICATIONS AND CONCLUSIONS

This chapter concludes the thesis by suggesting several improvements that might be made in the work that was done. Some possible applications of the concept of automated visual representation of graphs and, in particular, the representation of symmetry in graphs are also considered briefly. Finally, we shall attempt to evaluate the degree of success of our efforts to create an automated generation process for visual representation of graphs in light of our original goals.

## Improvements

The two main areas of the thesis research which are most susceptible to improvement are those of the calculation of the automorphism group and the generation of visual representations. In Chapter IV it was indicated that the set of heuristics used in the automorphism group calculation had proven very efficient and practical. However, as indicated then the chosen collection of heuristics is not definitive. There may be additional heuristics which are especially suited for certain classes of graphs and worth implementing. For example, a worthwhile addition would be the inclusion of the algorithm of Corneil and Gotlieb. Until the conjecture about the automorphism partitioning is proven the results could be treated just as those of any other heuristic.

The reordering of the partition sets option was implemented with the idea of possibly reducing the execution time required by the final automorphism group algorithm. Another very simple such option would be one where the user can determine the ratio of edges to vertices in the graph. If by taking the complement of the graph it is possible to obtain a graph with fewer edges it may be worthwhile to use the complement to obtain the automorphism group. It is possible to use either the graph or its complement since the automorphism group of a graph is the same as the automorphism group of its complement. The possible saving in execution time comes from the fact that the final automorphism group algorithm calculates the automorphism group using the fact that the successors of a vertex map onto the successors of the object of the vertex under the mapping. Thus, by reducing the number of edges the number of successors that must be checked is reduced and hence there is a saving in execution time. The difference in the number of edges between the graph and its complement would have to be large enough to make it profitable to use the complement, taking into consideration the time required to calculate the complement of the graph.

Again, in connection with the final automorphism group algorithm, it might prove useful when checking the vertex mappings to use the fact that the predecessors of a vertex must map onto the predecessors of the image of the vertex. This fact, of course, is applicable only to digraphs. The tradeoff involved here is whether the increase in execution time required to check a mapping would be offset by possible reductions in the number of mappings that will need to be checked. That is, by checking both facts at the same time, it should be possible on the average to discover sooner that a permutation is not an automorphism.

The area of the thesis in which improvements are most important is in connection with the process of generating visual representations. One of the minor problems currently encountered in using the visual representation routines is to obtain a representation based on a mapping containing few fixed vertices. This problem can be avoided by examining the orbits of the automorphism and determining whether the percentage of fixed vertices indicates that a 'good' representation may be forthcoming. This, however, is somewhat tedious and it might be useful to add a feature whereby the representation routine would generate representations only for those automorphisms having a user specified percentage of non-trivial mappings.

Another useful improvement would be to extend the postediting facilities which are currently oriented primarily towards manipulating mirror-image symmetries. There are no facilities available for treating rotational symmetries in the same manner. Another possibility would be to improve the editing facilities for tree representations. In particular, it would be very useful to be able to manipulate whole branches of a tree as single elements.

With the current representation routines the user is given the choice of having a representation created for each automorphism after the automorphism is generated. It is not possible to create the automorphism group first and then use the representation routines to generate the representations. In cases, however, where the user already knows the automorphism group, it would be useful to be able to do this.

Another consideration is whether it is necessary to use the whole automorphism group in order to obtain a satisfactory representation. For example, the user might be able to obtain the representation he requires from one of the first few automorphisms. Thus, one possibility would be to generate several automorphisms and then use these to generate one or more cyclic subgroups of the automorphism group. It is impossible to determine if any such cyclic subgroup is actually the complete automorphism group without calculating the automorphism group itself. However, the representations derived from these automorphisms may be sufficient for the user's purposes and, of course, it is much quicker to create such subgroups than it is to calculate the whole automorphism group.

Another problem related to the generation of the representations is that of the uniqueness of the representations. As indicated in Chapter III the mappings of the automorphism group may give rise to representations which are similar if not identical. In many cases the only difference is in the labelling of the vertices. One possible way to reduce, if not solve, this problem would be to examine the orbits of each mapping of the automorphism group. Mappings whose orbits are the same, except for the order of the vertices in the orbits, give rise to very similar representations. Thus, it might be worthwhile to generate only one representation for each set of mappings whose orbits contain the same elements. The problem here is that it would be necessary to calculate the whole automorphism group in order to be able to examine the orbits of all the mappings.

When the reasons for using the automorphism group of a graph as a basis for generating visual representations of the graph were discussed it was indicated that one advantage of this method was that it made it possible to create representations in a systematic fashion. This should mean that the same representations will always be generated for a graph regardless of how the graph is labelled. Closer inspection of the General Representation Routine procedure reveals that this is not necessarily true. The problem is that the fixed vertices are selected for the various axes of symmetry (assuming that more than one axis is used) in the order in which the vertices are labelled. This means that by relabelling the graph one could conceivably obtain two different representations for the same automorphism of the graph. The allocation of the fixed vertices to different axes should be made in a deterministic fashion. For example, the longest path of fixed vertices could be used first and so on.

The same problem appears again when the routine adds to the display mirror-image pairs adjacent to a fixed vertex. There may be more than one such pair adjacent to a fixed vertex, yet only one pair at most is used with each fixed vertex. Moreover, the choice as to the pair added to the display is quite arbitrary. The procedure should be changed so that all pairs adjacent to the fixed vertex are used. Later, when adding mirror-image pairs which are not adjacent to any fixed vertex to the display, the pairs are added according to the order of their vertex labelling. This should be changed so that they are added according to their distance from the first fixed vertex on the axis or according to some other deterministic criteria.

In actual practice the above problems are not too serious as they will only affect someone using the General Representation Routine to examine and compare the members of a particular class of graphs. However, as indicated in Chapter V someone in this situation is quite likely to use a special representation routine especially suited for use with the particular class of graphs.

## Applications

The GSYM system was designed as a tool for graph-theoretical research into the question of visual representation of graphs. Thus, it might be expected to prove useful to anyone interested in continued research in this area. It is also a relatively general graph processing tool and could be used for implementing and testing general graph-theoretical algorithms. It could conceivably be used for other non-graph-theoretical problems requiring a display facility. However, this guise would be an ill fit because the operations and data structures of the system are designed and implemented using graph-theoretical terminology.

The GSYM system in combination with the representation routines is obviously best suited for use as a tool for graph theorists. It provides a means whereby a graph theorist can easily investigate the structure of a graph or class of graphs in which he is interested. The system and the representation routines were designed to be easy to use and also to be as fail safe as possible so that a person with little or no previous experience with computers or the graphic display could start using the system immediately. A tool is of little use if the effort required to use it is greater than that required to tackle the problem without the tool. Thus, we are suggesting that the most immediate and probably the most useful application of the thesis work is to replace the trial and error, paper and pencil method of drawing graphs.

We also hope to show that this facility is ideal for communicating graph-theoretical information about the structure of graphs. Representation routines should be very useful for communication between graph theorists in situations where someone needs to obtain a catalogue of a class of graphs for his own research. At present it would only be possible to provide him with a list of the adjacency matrices of the graphs plus possibly some other written data on the structure of the graphs. However, by using a representation routine for the class of graphs it would be possible to provide one or more pictures of each graph as well.

There are other possible applications for the thesis work which are not so immediate. For example, some of the original heuristics used for automorphism group calculation were based on heuristics used by Unger to tackle the graph ismorphism problem. The fact that the interactive versions of these heuristics as well as the others implemented have proven much faster and more practical than the non-interactive versions indicates that these interactive routines could, in turn, be applied to the graph isomorphism problem. Due to the similarity of the two applications one would expect the same type of improvement in execution time requirements. It might also be possible in some cases to settle the question of whether two graphs are isomorphic by simply examining the representations created for the two graphs by one of the representation routines. This would be practical only if the representations already existed. It is unlikely to be cheaper in terms of execution time to calculate the automorphism groups of both graphs solely in order to obtain visual representations of the graphs which may or may not settle the isomorphism problem.

During the discussion of the ring structure representation routine in Chapter III it was mentioned that since the representation is not tied to any automorphism of the graph, it is possible for the routine to create a representation of a graph having only the trivial automorphism group. After that this question was tactfully avoided. The representation routines were based on the automorphism group of the graph and the representations were derived almost exclusively from non-trivial automorphisms. While it is possible to use the representation routines to obtain a representation from the identity mapping, such representations are of little practical value as the identity mapping provides no information about the structure of the graph. Thus, the question that was avoided. What happens if the graph has only the trivial identity mapping in its automorphism group? There is usually little point in generating a representation from the identity mapping.

This problem is related to one which has been investigated previously without much success. What is the minimum number of changes necessary to a graph with a trivial automorphism group in order to create a graph with a non-trivial automorphism group? This problem was originally investigated by Erdos and Renyi (xxxvi) and more recently by Quintas (xxxvii). An asymmetric graph is a graph with a trivial automorphism group. The degree of asymmetry of a graph is the minimum number of changes (edge additions and/or deletions) required to create a graph having a non-trivial automorphism group. The results obtained so far have been restricted primarily to existence proofs. There has not yet been found a construction which determines what edges must be added and/or deleted to create the desired graph. However, it would seem intuitively feasible that one way to tackle this problem would be to use the partition created by the heuristics in the calculation of the automorphism group. The partition indicates which mappings are most likely to be automorphisms although a later check of these mappings may prove that they are not. However, when these checks fail it may be possible to use the failure points as starting points to determine what changes are most likely to produce a non-trivial automorphism.

The immediate relevance of this question to the visual representation routines is that it may be possible by making the required changes to obtain an automorphism which can be used to create a good representation of the new graph. This representation may then be altered (reverse the edge additions and/or deletions) to produce a representation of the original graph. Moreover, if the required changes were not too extensive the final representation should also be a good representation in that it will reveal the "almost symmetric" structure of the original graph.

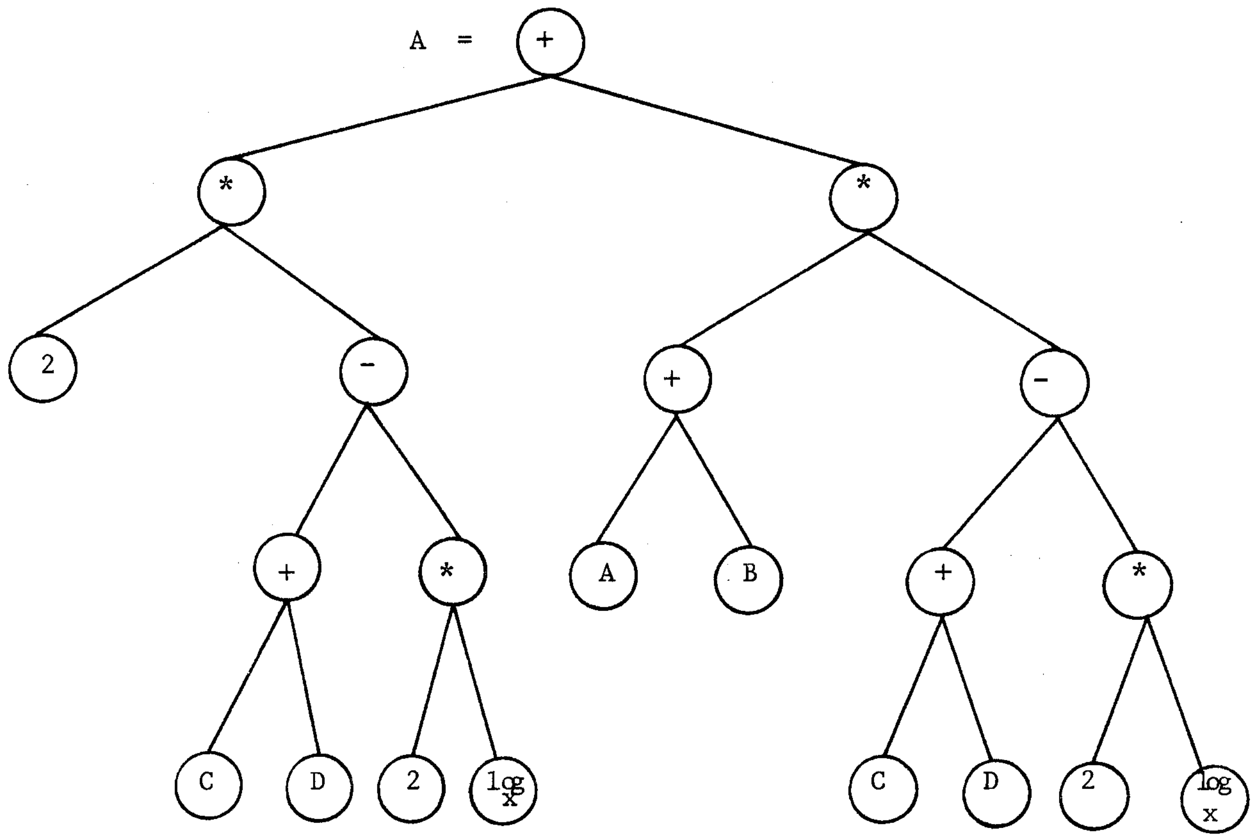
The most apparent application for visual representation routines is to problems where the structural symmetry of an object is important. The representation routines are intended to portray visually the structural symmetries in a graph. However, a graph can be used to model many different relationships from chemical bonding to program structure. Thus, it might prove useful to apply the visual representation routines to such areas. For example, in the field of fault diagnosis one of the more recent techniques for digital fault simulation is that of functional simulation.(xxxv) With this method a digital system composed of modules $M_1$, $M_2$, ..., $M_n$ is to be simulated. Overall system behaviour is to be determined using the assumption that there are faults in only certain of the modules. Thus the remaining modules need not be simulated at the gate level. It is only necessary to compute the output of these modules as a function of their input. If such a digital system were defined as a graph, then one could use a representation routine to obtain pictures of the symmetries in the system. These pictures would indicate which modules should or should not be simulated in detail. For example, if modules $M_i$ and $M_j$ form a mirror-image pair, then if module $M_i$ is not being simulated at the gate level then the mirror-image symmetry suggests that neither should module $M_j$ be simulated at the gate level. The idea is that the symmetry in the system would help determine how to set up the simulation tests.

Another area where symmetry is important is in the optimization of computer programs. This can take place at two levels. On the higher level there is the question of program structure and the best segmentation of programs in such applications as time-sharing. Graph-theoretic techniques are currently being used in such investigations. The structure of a program, interpreted as a graph, could be displayed with its symmetries using the representation routines. Such displays might then indicate the best program segmentation points or reveal portions of the program which are repeated. On a lower level there is the problem of formula simplification. This is a subject which has received considerable attention in connection with the optimization of code generated by compilers. For example, in a statement or series of statements where a common subexpression appears it may be less costly in terms of execution time to replace the expression by a variable and set the variable equal the value of the expression before executing the statement(s).

Example 6.1:

```
A = 2*(C+D-2*LOG(X) )   +   (A+B)*(C+D-2*LOG(X) )
```

Formula Tree Representation:



In the above simple example the subexpression C+D-2*LOG(X) could be replaced by a variable set equal to the subexpression. It would then be necessary to generate the code for the subexpression only once. Moreover, the value of the subexpression would be calculated only once at execution time. If such a formula were treated as a tree and then displayed on the screen using the Tree Representation Routine the user could easily identify the symmetric subexpression from the representation. He could then determine for himself the best way to simplify the formula.

The above discussions were intended to indicate some of the possible applications and benefits of research into the visual representation of symmetries in graphs. It has not been possible to consider these applications in detail and assess their practical merits as each application appears to be a major research problem in its own right.

## Conclusions

Apart from such considerations as the efficiency of the coding for the representation routines, the generality of the GSYM system, the selection of automorphism group heuristics and so on, the question to consider is what has been achieved in light of the original goals for the thesis. The original aim was to create an automated process for generating visual representations of graphs. On the whole, this goal has been achieved. The representations, being derived from the automorphism group of the graph, are obtained in a systematic fashion. Moreover, it is possible to generate one or more representations (usually more) for each graph. In addition, thanks to the interactive GSYM system, it was possible to successfully handle the automorphism group problem. The use of interactive heuristics in the calculation of the automorphism group made it not only possible, but even practical, to base the visual representations of a graph on its automorphism group.

The stated goals indicated that the process of generating visual representations was to be automated. The user should be able to request a representation of a graph and have the system return with the representation on the screen. As we know, we have had to allow for postediting of the representations that are generated. Thus, the system is not fully automated. However, the need for postediting is reduced when special representation routines are used and this is the direction we expect further research in this area will follow. That is, just as with the development of programming languages we would expect a parallel development in the creation of visual representation routines -- a proliferation of special representation routines intended for specific applications.

As to the question of whether the representations are "good" representations, this is difficult to answer in general. The representations obtained from the Tree Representation Routine are very good. The representations generated by the General Representation Routine can usually, thanks to the postediting facilities, be transformed into quite satisfactory if not good representations. Unfortunately, it is not easy to determine what constitutes a "good" representation. However, we feel that we have shown that it is possible to create representation routines which are suitable for most users and for most applications. Moreover, we believe that this work has shown that this facility should be present in all graph processing languages, especially those which make use of a graphic display medium.

# A P P E N D I X   A

# APPENDIX A : HARDWARE DESCRIPTIONS AND DEFINITIONS

## Hardware Descriptions

The IBM S/360-2250 Model 1 Graphic Display Unit is the graphic device on which the GSYM system was implemented. The 2250 Display Unit has been designed to operate in combination with an S/360 computer; in this case the 2250 was connected to an IBM 2911 switching unit which, in turn, was linked to an S/360 Model 75 and an S/360 Model 50. Thus, the 2250 Display Unit and the GSYM system could be used with either machine. The following description briefly describes some of the features of the 2250. It will not discuss the S/360 machines except in their relation to the 2250.

## IBM 2250 Display Unit

The IBM 2250 Display Unit is a programmable cathode ray tube (CRT). Although the 2250 is a computer with its own control unit and memory, it is normally expected to be operated as an I/O device attached to a channel of an S/360 computer. The channel is usually a multiplexor channel as the 2250 is basically a low speed device.

The CRT consists of an electron gun and a phosphor-coated screen. The electron gun focuses an electron beam on the screen which causes the phosphor to glow briefly at the point of contact. The movement of the electron gun is controlled by graphic orders placed in the 2250 memory.

Thus, the user programs the 2250 to trace images on the screen, placing graphic orders in the 2250 buffer to create a display. The CRT screen is divided into a 1024 by 1024 grid, with each intersecting point being addressed as an ordered pair (x,y) where $0 \leq x,y \leq 1023$.

The 2250 at the University of Waterloo contains most of the optional features which are available to the Model 1 2250 Display Unit. These include the following:

1) Buffer Memory

The buffer for the 2250 consists of between 4K and 32K bytes of storage in 4K increments. The University of Waterloo 2250 has an 8K buffer. This buffer can be accessed by either the 2250 or the main computer with an access time of 4.2 microseconds per byte. The 2250 uses the buffer as a memory for storing the instructions and data which control the movement of the electron gun. The main computer uses the buffer to pass data and instructions to and from the 2250. While the 2250 executes the graphic orders in the buffer the main computer can continue processing. Thus, the screen may contain a display while the main machine is preparing the next.

2) Absolute Vectors

Without this feature the 2250 could only draw straight lines horizontally, vertically or at a $45^{\circ}$ angle. Any other straight line would be drawn as a combination of such lines. With this option, however, the 2250 can draw straight lines between any two points on the screen grid.

3) Character Generator

This option is a hardware device which draws alphanumeric characters on the screen. The characters are stored as one byte data items in the 2250 buffer and when they are to be displayed the character generator directs the movement of the electron gun so as to trace the character. Without this option characters must be generated by software using a sequence of vectors.

4) Function Keyboard

The function keyboard consists of a box with 32 pushbuttons or keys on it. The keyboard is used primarily for interrupting the main computer. An interrupt is sent to the main computer which then requests data from the 2250 to identify the key used. In the GSYM system the keyboard is used mainly to control light-pen operations.

5) Alphanumeric Keyboard

This keyboard is very similar to a typewriter keyboard. Its primary purpose is to type in alphanumeric data which is stored in the 2250 buffer. This data is displayed on the screen as it is typed. A special underscore character is displayed on the screen beneath the position where the next character typed will be placed.

6) Light-Pen

The light-pen is another user control device. It is a pen-like stylus which is attached via a flexible cord to the 2250.

The pen can detect the beam of the electron gun as it passes over the screen. If the pen is enabled when it detects this beam it halts the display and sends an interrupt to the main computer. Information is then sent to the main computer specifying the type of data detected (line or character), the location of the byte accessed in the 2250 buffer and the x,y co-ordinates of the beam. The pen is enabled and disabled under program control. When it is enabled an interrupt occurs as soon as the beam is detected. Otherwise, the pen switch must be activated by pushing the tip of the pen gently against the screen.

Further details on the use and programming of the 2250 Display Unit may be obtained from the IBM manuals listed in the Bibliography (xxv, xxvi).

## Graph-Theoretic Definitions

Set - A set is a collection of distinct elements (objects). The set {a,b,c,d,e} contains the elements a, b, c, d, and e.

Empty Set - the empty set or null set, denoted by $\emptyset$, is a set that contains no elements.

Subset - a set T is a subset of another set S if every element in T is also in S.

Group - a group is a nonempty set G together with a binary operation (denoted by uv for u, v in G) which satisfies the following:

1) Closure - For all u, v in G, uv is also an element of G.

2) Associativity - For all u,v,w, in G u(vw) = (uv)w.

3) Identity - There is an element i in G such that iv = vi for all v in G.

4) Inversion - For each v in G there is an element $v^{-1}$ in G such that $vv^{-1} = v^{-1}v = i$.

Subgroup - if S is a subset of a group G and S is a group under the same binary operation as in G then S is a subgroup of G.

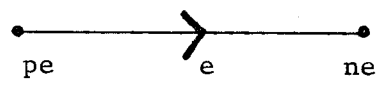
Cyclic Group - a group which can be generated by one element is called a cyclic group. Thus, if g is in a group G, then the set of powers <g> = {$g^n$ : n is an interger} forms a subgroup of G, the cyclic group generated by g. The number of elements in a group is called the order of the group. Thus, if g is the generator of a cyclic group of order n, then the elements of this group can be given as {$g, g^2, g^3, \ldots, g^n$}.

Permutation - a permutation of a set of elements G is a 1 - 1 mapping of the set G onto itself.

Cycle - a cycle in a permutation is a subset of elements that are cyclically permuted. For example, in the permutation $\begin{pmatrix} abcdef \\ cedabf \end{pmatrix}$, {a,c,d} is a cycle since a is permuted into c, c is permuted into d and d is permuted into a. We shall also refer to a cycle as an orbit. Thus, the orbits in the above permutation are {a,c,d}, {b,e} and {f}.

Graph - a graph G consists of a vertex set VG and an edge set EG. Each edge is incident upon two vertices called its end-vertices. The end vertices need not be distinct, in which case the edge may be called a loop. The number of vertices in G is denoted by |VG| and the number of edges in G is denoted by |EG|.

Directed Graph[1] - a directed graph is a graph in which for each edge e one of the ends is specified as the positive end pe and the other as the negative end ne. We follow the notation:



The direction goes from pe to ne. e is an outdirected (indirected) edge incident to pe (ne). If vertex v = pe and vertex w = ne then e is a directed edge from v to w.

Incidence - if e is an edge with ends v and w, then vertex v and edge e are incident to each other as are w and e.

Adjacency - if e is an edge with ends v and w, then v and w are adjacent to each other.

1. The term digraph may be used in place of directed graph.

Linear Graph - a graph is linear if it has no two edges with the same pair of ends. That is, there is at most one edge connecting any two vertices. A linear digraph is a directed graph such that there are no two directed edges with positive end v and negative end w.

Simple Graph - a linear graph is simple if it contains no loops. A simple digraph is a linear digraph which does not contain any directed edge whose positive end is the same as its negative end.

Subgraph - a subgraph H of G is a graph for which $VH \subseteq VG$, $EH \subseteq EG$. H is a proper subgraph of G if $H \neq G$. For any $Y \subseteq VG$, the subgraph of G generated by Y, written $G[Y]$, is the subgraph with vertex set Y and the edge set those edges of G having both ends in Y.

Spanning Subgraph - a spanning subgraph H of a graph G is a subgraph H of G such that $VH = VG$.

Outvalence (Invalence) - the outvalence (invalence) of a vertex v is the number of edges with positive (negative) end v.

Valence - the valence of a vertex x is the sum of its invalence and outvalence. In an undirected graph the valence of a vertex x is the number of edges incident to x.

Regular Graph - a graph is regular of degree n if all its vertices have the same valence, that valence being n. For example, a cubic graph is a regular graph of degree 3.

Path - a path is a sequence of the form $P = (v_o, e_1, v_1, e_2, \ldots, e_n, vn)$, where $v_i$ are vertex terms, $e_i$ are edge terms and $e_{i+1}$ has ends in G of $v_i$ and $v_{i+1}$. VP = the vertex set of the path P. EP = the edge set of the path P. $v_o$ is the origin of the path and $v_n$ is the terminus of the path. A path containing only one vertex may be denoted as a set with only one element, for example, the path $\{v_o\}$.

Simple Path - if the vertex terms of a path P are all distinct then P is a simple path.

Closed Path - a path P is closed if its origin is the same as its terminus. Otherwise, it is open.

Length - the length of a path is the number of edge terms in the path.

Directed Path - let $E_+P$ = the set of forward edges of the path $P = \{e :$ there is an edge term $e_j$ of P such that $pe_j = v_{j-1}$ and $e = e_j\}$.

Let $E_-P$ = the set of reverse edges of the path $P = \{e :$ there is an edge term $e_j$ of P such that $ne_j = v_{j-1}$ and $e = e_j\}$.

A forward directed path is a path P for which $E_-P = \emptyset$.

A reverse directed path is a path P for which $E_+P = \emptyset$.

A directed path from v to w is synonymous with "there is a forward directed path with origin v and terminus w". The term 'positive' ('negative') path may be used to denote a forward directed (reverse directed) path.

Connected Graph - a graph G is connected if every pair of vertices in VG is joined by a path.

Distance - the distance between two vertices v and w is the length of the shortest path joining them if any.

Polygon - a polygon is a connected non-null graph each vertex of which has valence 2. An 'n-gon' is a polygon having n vertices.

Directed Polygon - a directed polygon is a polygon in which each vertex has invalence 1 and outvalence 1. A directed polygon may also be referred to as a positive or negative polygon.

(Directed) Circuit - a (directed) circuit is the edge set of a (directed) polygon.

Complement - the complement G' of a graph G has VG' = VG but two vertices are adjacent in G' if and only if they are not adjacent in G.

Bipartite Graph - a bipartite graph is a graph G whose vertex set VG can be partitioned into two subsets $V_1$ and $V_2$ such that every edge e in EG has one end in $V_1$ and the other in $V_2$.

Component - a component of a graph G is a maximal non-null connected subgraph of G.

Cut Vertex - a cut vertex of a graph is a vertex whose removal from the graph increases the number of components in the graph. If v is a cut vertex of a connected graph G, then G - v is not connected.

Nonseparable Graph - a nonseparable graph is a non-trivial connected graph which has no cut vertices.

Block - a block of a graph G is a maximal nonseparable subgraph of G.

Coboundary - let $X \subseteq VG$. The coboundary of X is the set of edges in EG each having one end in X and the other end in VG - X.

Cutset - a cutset is a minimal non-null coboundary. That is, a cutset is a coboundary which does not contain any other coboundaries.

Neighbourhood - the set of vertices adjacent to a vertex v is called the neighbourhood of v. The set of vertices at a distance N from a vertex v is called the Nth-neighbourhood of v.

Tree - a connected graph in which there are no polygons is called a tree. A subgraph of G (where G is any graph) which is a tree is a subtree of G.

Level in a Tree - the Nth-neighbourhood of the root vertex of a rooted tree is the Nth level of the tree.

Spanning Tree - a spanning subgraph of a graph G which is a tree is called a spanning tree or spanning subtree of G.

Rooted Tree - a rooted tree is a tree with one vertex, the root, distinguished from all the other vertices.

Endpoint - an endpoint of a graph G is a vertex whose valence = 1.

Branch - a branch at a vertex v of a tree T is a maximal subtree of T containing v as an endpoint. The number of branches at v is equal to the valence of v.

Centre - every tree has a centre consisting of either one vertex or two adjacent vertices. If the centre consists of two adjacent vertices it is also called a bicentre. The centre is obtained by repeatedly removing the endpoints of the tree until there is finally only one vertex or a pair of adjacent vertices left in the tree.

Embedding - a graph is said to be embedded in a surface S when it is drawn on S so that no two edges intersect.

Planar Graph - a graph is planar if it can be embedded in the plane. A plane graph has already been embedded in the plane. This may be called a planar embedding of the graph.

Regions - we refer to the regions defined by a plane graph as its faces, the unbounded region being called the exterior face.

Planar Map - a planar map is a connected plane graph together with all its faces.

Cyclically k-connected Graph - a graph G is cyclically k-connected if for any coboundary $\delta X$ such that $G[X]$ and $G[VG-X]$ each includes a polygon, $|\delta X| \geq k$.

<u>Adjacency Matrix</u> - the adjacency matrix $A = [a_{ij}]$ of a labelled graph G with n vertices is the nxn matrix in which $a_{ij} = 1$ if $v_i$ is adjacent to $v_j$ and $a_{ij} = 0$ otherwise. For a labelled digraph $a_{ij} = 1$ if there is a directed edge from $v_i$ to $v_j$ in EG, otherwise, $a_{ij} = 0$. The adjacency matrix of a digraph need not be symmetric.

<u>Incidence Matrix</u> - the incidence matrix $B = [b_{ij}]$ of a labelled graph G with n vertices and m edges is the nxm matrix in which $b_{ij} = 1$ if vertex $v_i$ is incident to edge $e_j$ and $b_{ij} = 0$ otherwise.

<u>Mapping</u> - let G and H be graphs. Let f be a correspondence associating with each edge or vertex x of G a unique edge or vertex fx of H and satisfying the following two conditions:

i) If x is in VG then fx is in VH.

ii) Let A be an edge in G with ends x and y. Then either fx = FA = fy or FA is an edge of H whose ends are fx and fy.

f is a mapping of G into H. If the mapping is 1 -1 then f is a mapping of G onto H. We will refer to fx as the image of x under the mapping f. Moreover, if the mapping f is 1 - 1 and v = fx, then x will be called the preimage of v under the mapping f in order to stress that something is being mapped onto v.

<u>Isomorphism</u> - a 1 - 1 mapping of a graph G onto a graph H is an isomorphism and if such a mapping exists G is isomorphic to H. If f is an isomorphism of G onto H then f relates vertices of G to vertices of H, and edges of G to edges of H. Moreover, if edge A in EG is incident to a vertex v in VG then fA is incident to fv in H.

Automorphism - an automorphism of a graph G is an isomorphism of G onto itself. The set of all automorphisms of a graph G is denoted by A(G). A(G) is a group, called the automorphism group of G. The identity mapping of a graph G is an automorphism of G, called the identity automorphism or the trivial automorphism of G. A graph with a non-trivial automorphism is a symmetrical graph, otherwise the graph is asymmetrical. A graph such that every permutation of the vertices is an automorphism is called a full symmetric graph.

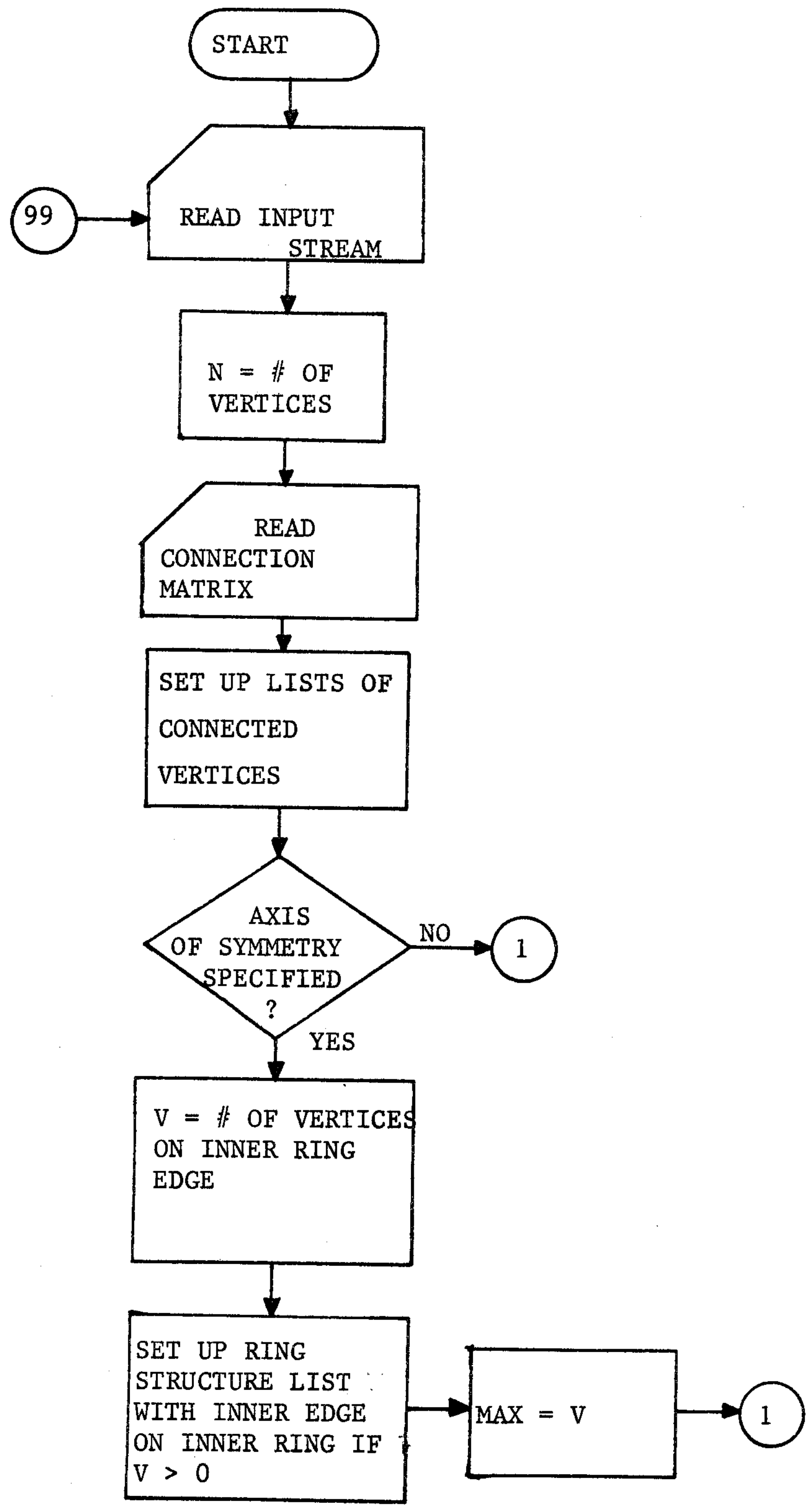
Partition - a partition on a set is a subdivision of all the elements in the set into disjoint subsets. That is, a partition on a set is a collection of subsets of the set such that every element in the set is placed in one and only one of the subsets.

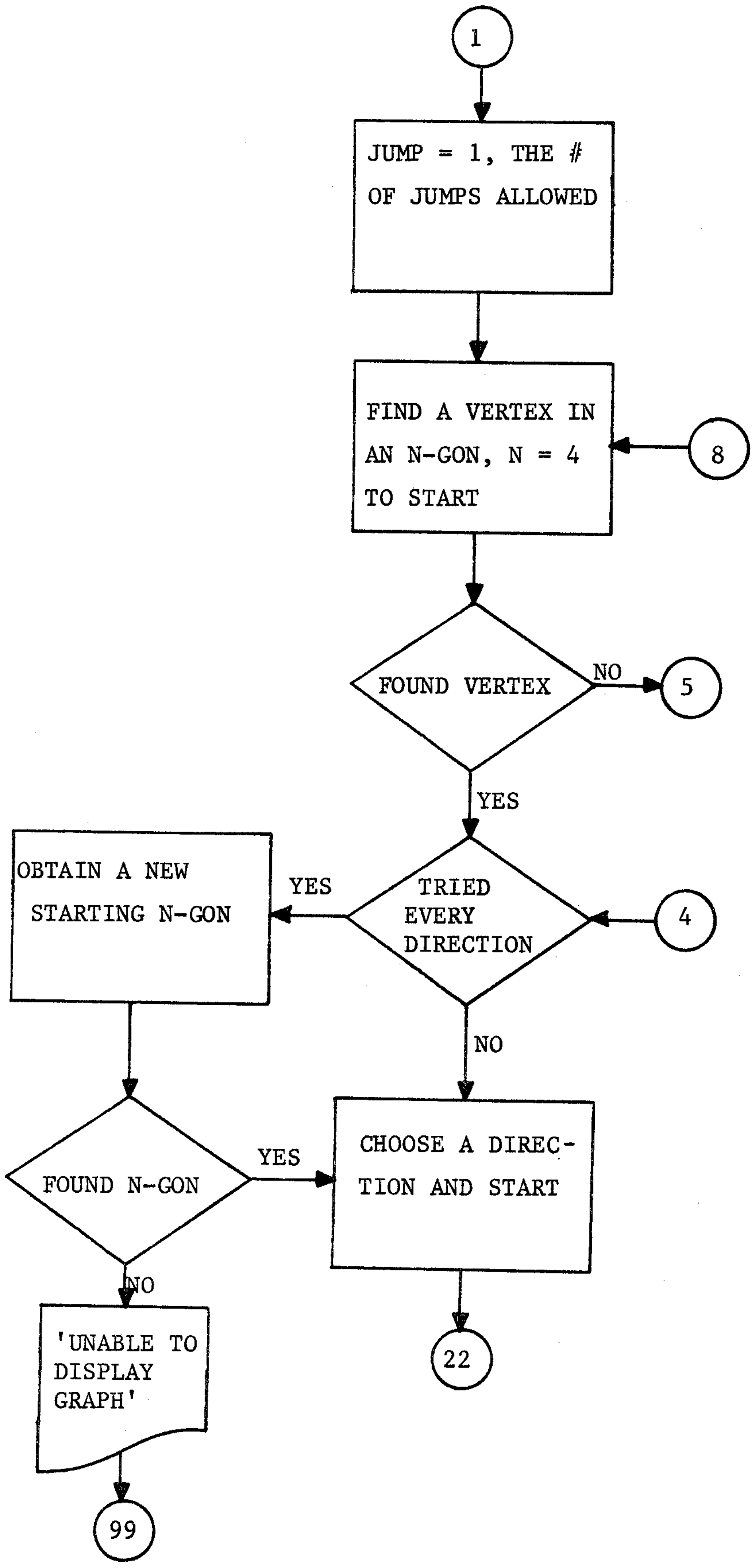
Automorphism Partitioning - the automorphism partitioning of the vertex set of a graph G is a partitioning of VG into sets $S_1$, $S_2$, ..., $S_n$ such that for all v and w in $S_i$ there exists an automorphism f in A(G) such that fv = w.

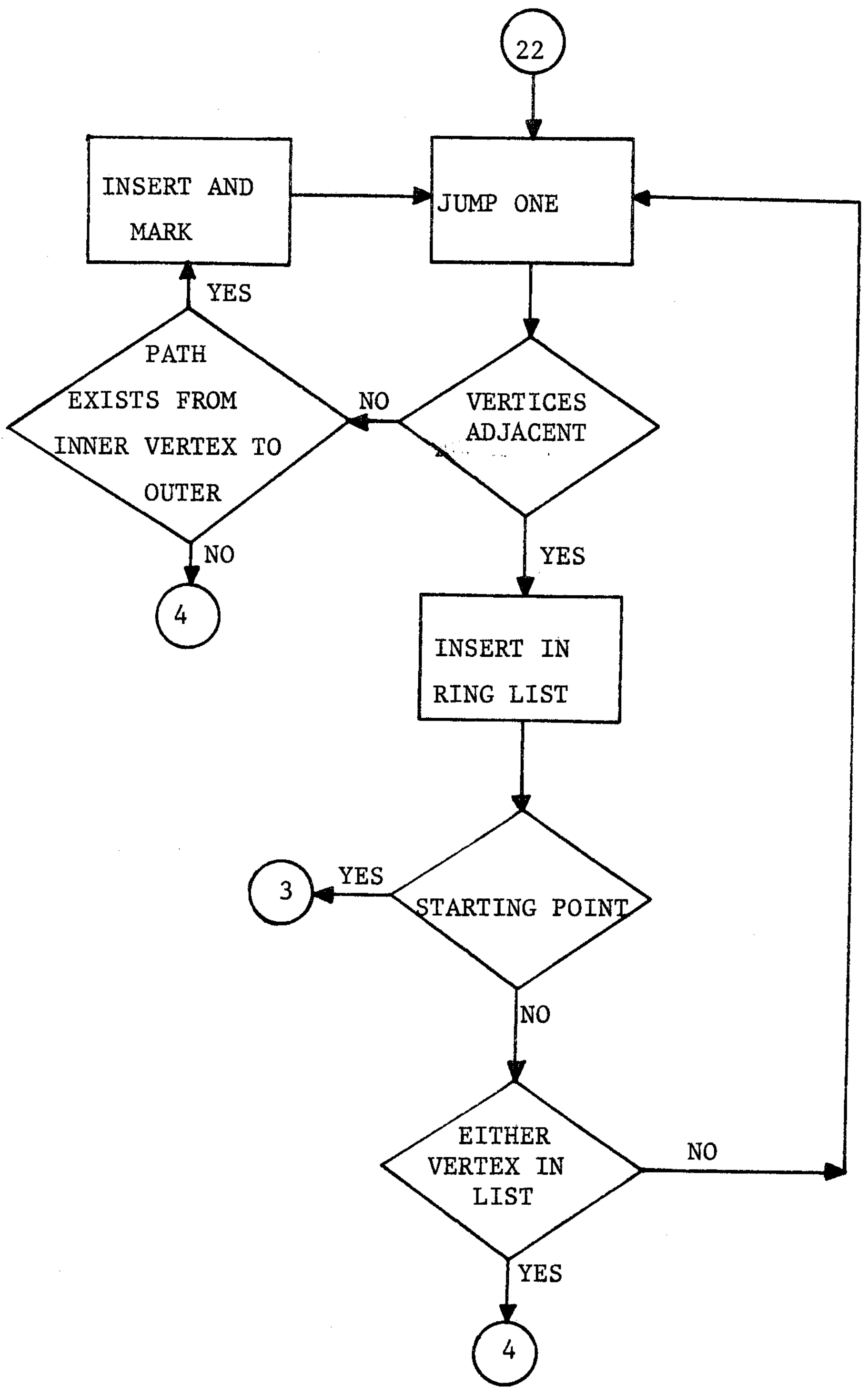
# A P P E N D I X  B

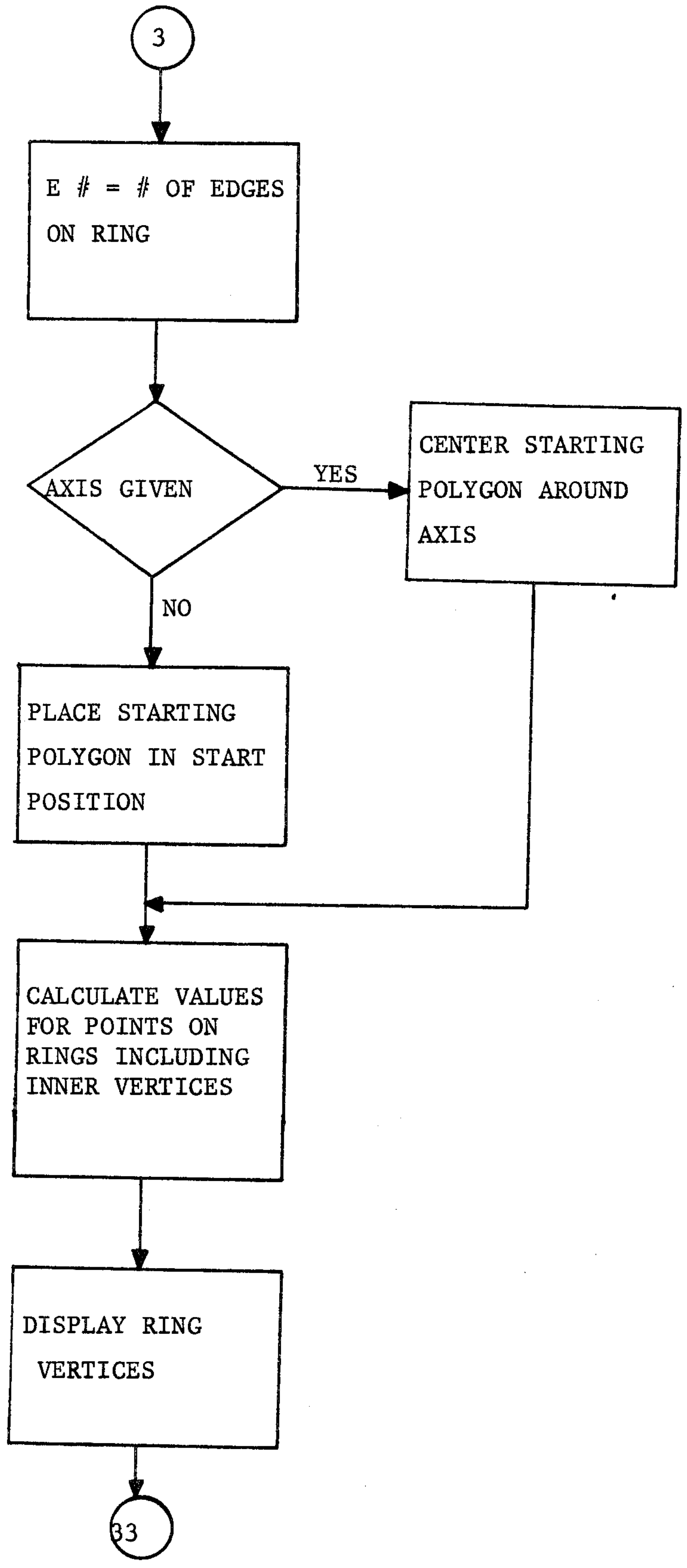
# APPENDIX B : REPRESENTATIONS OF CUBIC, CYCLICALLY 4-CONNECTED PLANAR GRAPHS

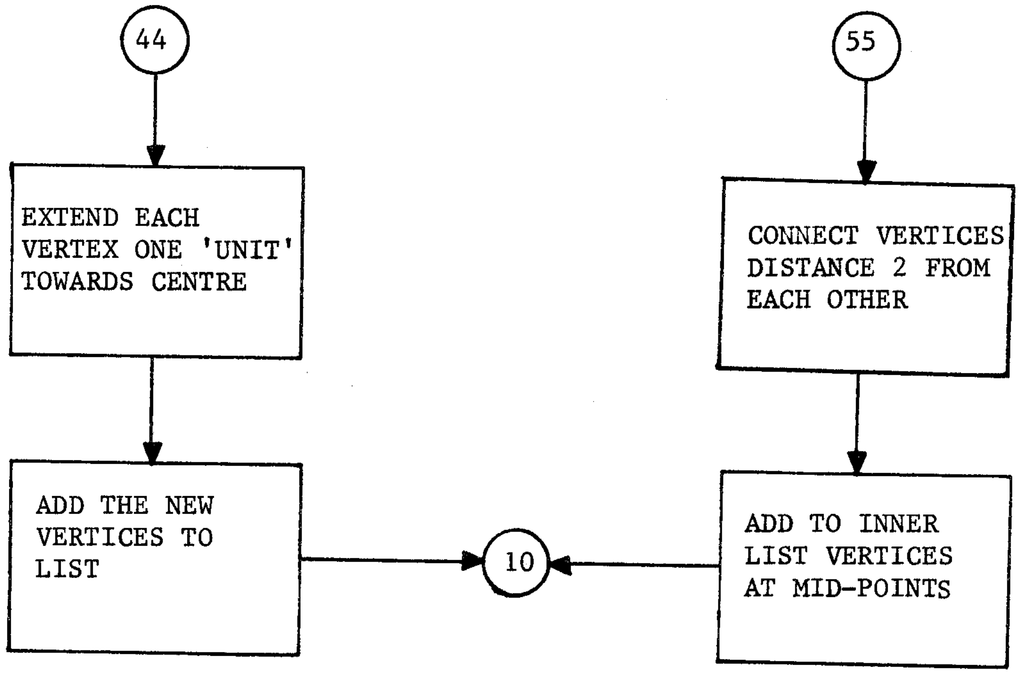
The representations contained in this Appendix have been generated using the representation routine described in Chapter III. The graphs are all cubic, cyclically 4-connected, planar graphs. Each graph is first shown as a hand-drawn planar map and is then shown in the form of a planar representation generated by the representation routine. The graphs are listed according to the number of regions in the planar map. A detailed flowchart of the representation routine is given at the start of the Appendix.

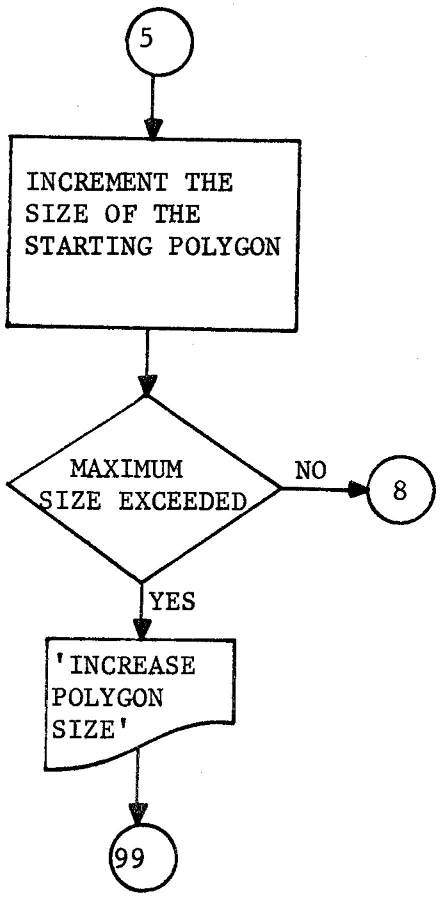

START
99
READ INPUT STREAM
N = # OF VERTICES
READ CONNECTION MATRIX
SET UP LISTS OF CONNECTED VERTICES
AXIS OF SYMMETRY SPECIFIED ?
NO
1
YES
V = # OF VERTICES ON INNER RING EDGE
SET UP RING STRUCTURE LIST WITH INNER EDGE ON INNER RING IF V > 0
MAX = V
1

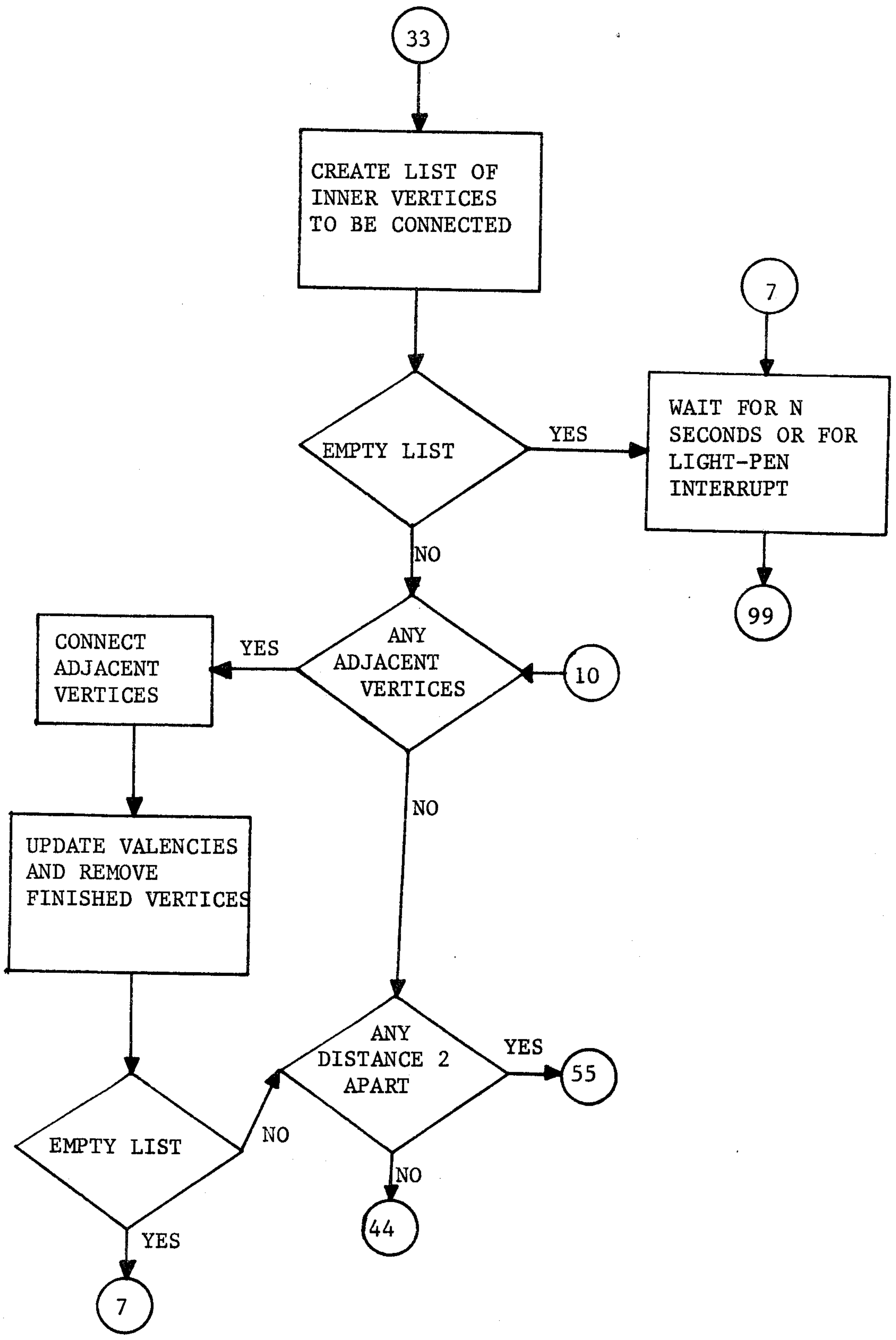

1
JUMP = 1, THE # OF JUMPS ALLOWED
FIND A VERTEX IN AN N-GON, N = 4 TO START
8
FOUND VERTEX
NO
5
YES
TRIED EVERY DIRECTION
4
YES
OBTAIN A NEW STARTING N-GON
NO
FOUND N-GON
YES
CHOOSE A DIRECTION AND START
NO
'UNABLE TO DISPLAY GRAPH'
22
99

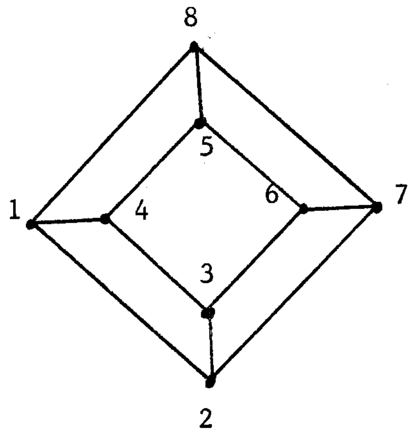

22
INSERT AND MARK
JUMP ONE
YES
PATH EXISTS FROM INNER VERTEX TO OUTER
NO
VERTICES ADJACENT
NO
4
YES
INSERT IN RING LIST
3
YES
STARTING POINT
NO
EITHER VERTEX IN LIST
NO
YES
4

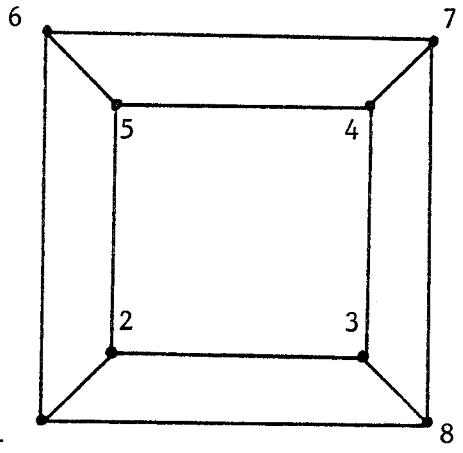

3
E # = # OF EDGES ON RING
AXIS GIVEN
YES
CENTER STARTING POLYGON AROUND AXIS
NO
PLACE STARTING POLYGON IN START POSITION
CALCULATE VALUES FOR POINTS ON RINGS INCLUDING INNER VERTICES
DISPLAY RING VERTICES
33


44
EXTEND EACH VERTEX ONE 'UNIT' TOWARDS CENTRE
ADD THE NEW VERTICES TO LIST
10
55
CONNECT VERTICES DISTANCE 2 FROM EACH OTHER
ADD TO INNER LIST VERTICES AT MID-POINTS

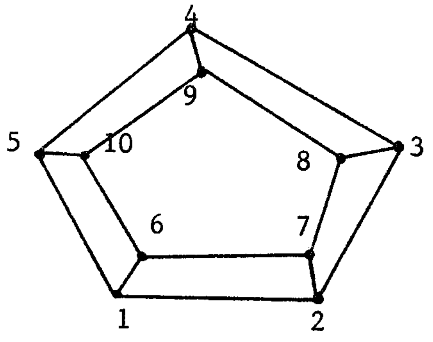

5
INCREMENT THE SIZE OF THE STARTING POLYGON
MAXIMUM SIZE EXCEEDED
NO
8
YES
'INCREASE POLYGON SIZE'
99

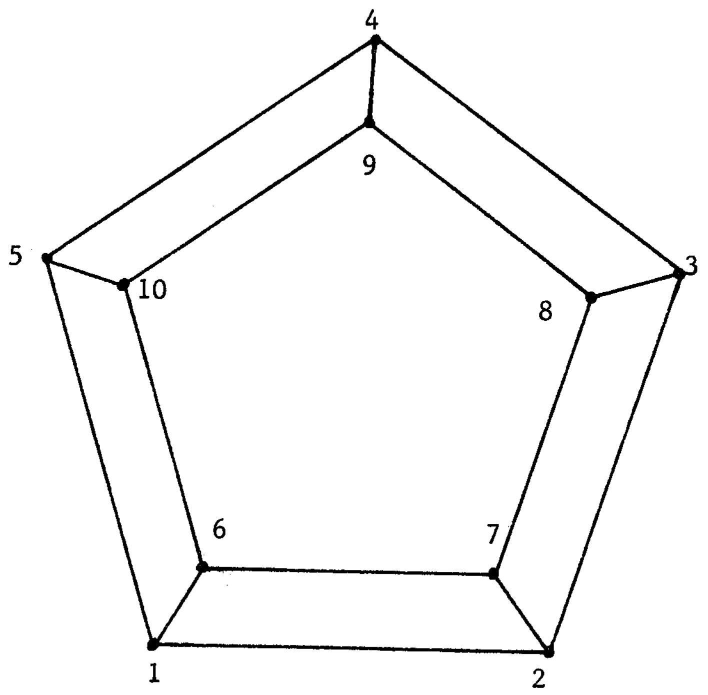

33
CREATE LIST OF INNER VERTICES TO BE CONNECTED
EMPTY LIST
YES
NO
7
WAIT FOR N SECONDS OR FOR LIGHT-PEN INTERRUPT
99
CONNECT ADJACENT VERTICES
YES
ANY ADJACENT VERTICES
10
NO
UPDATE VALENCIES AND REMOVE FINISHED VERTICES
ANY DISTANCE 2 APART
YES
55
EMPTY LIST
NO
NO
44
YES
7



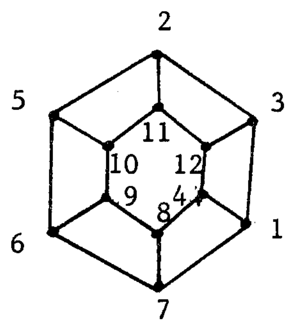


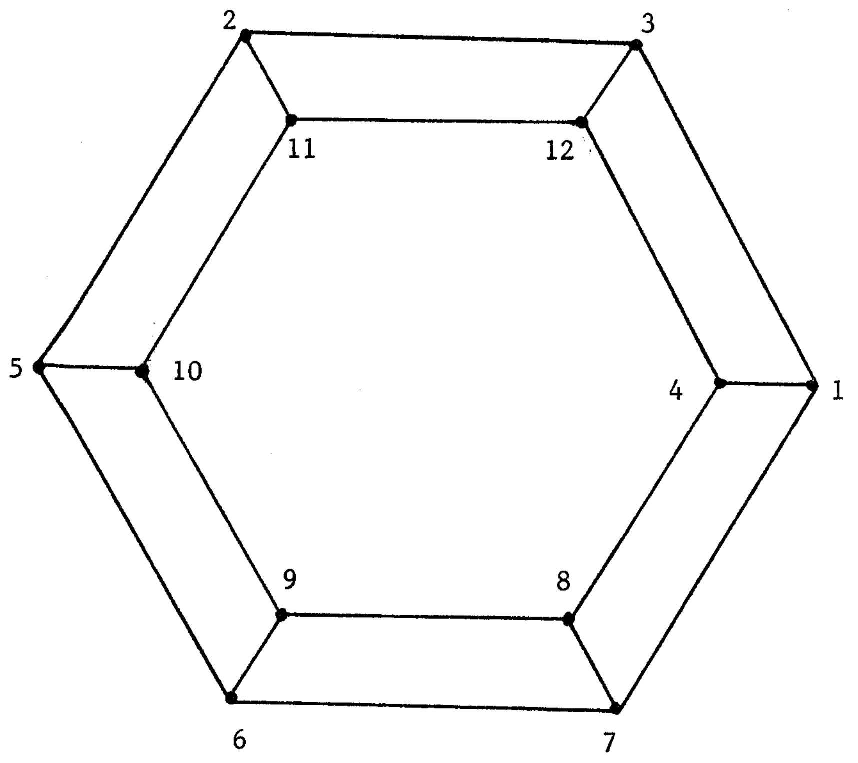
6 REGIONS : MAP 1 OF 1



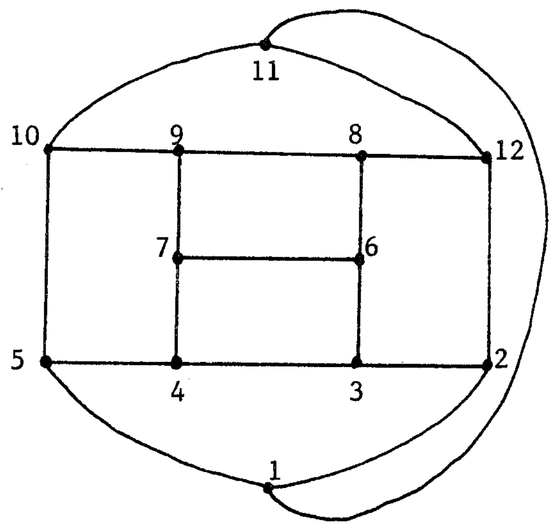


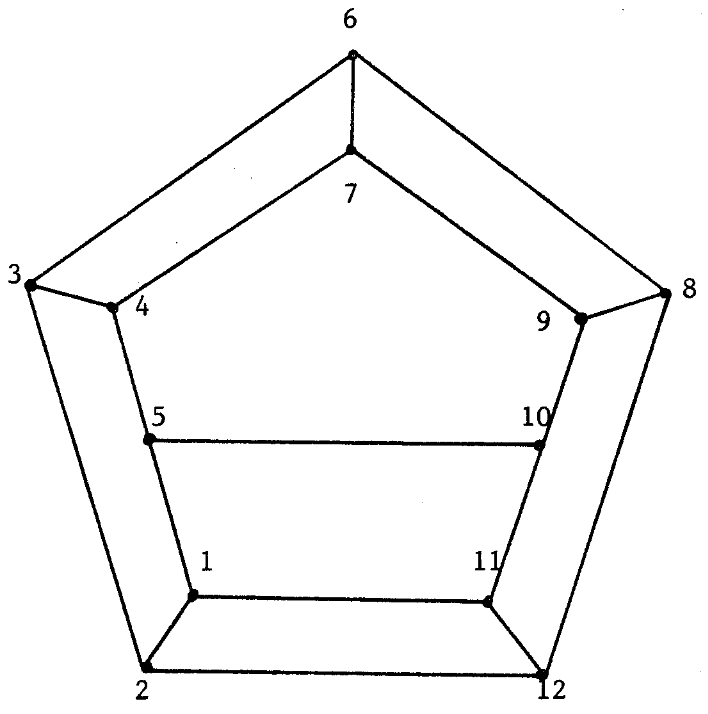
7 REGIONS : MAP 1 OF 1



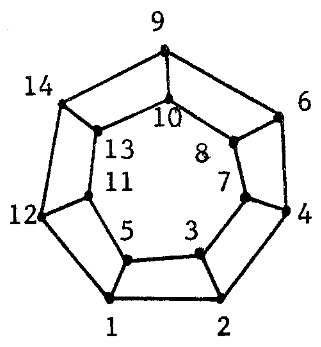


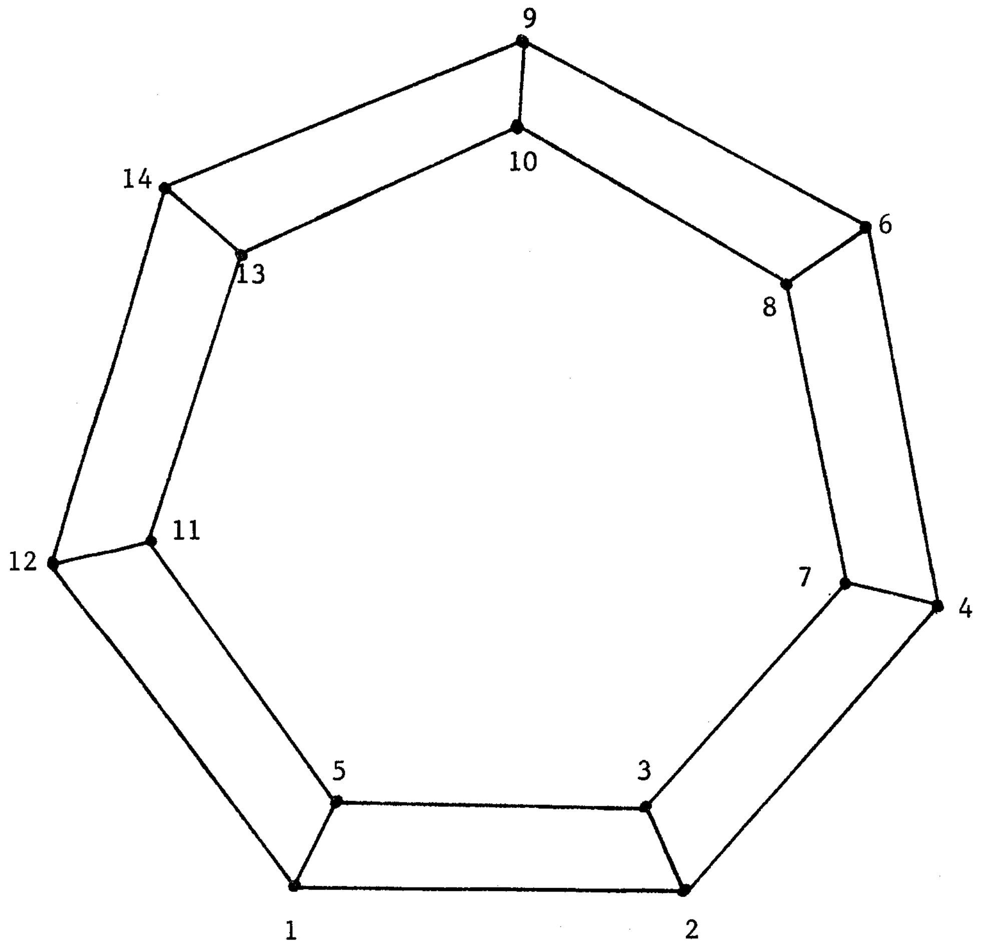
8 REGIONS : MAP 1 OF 2



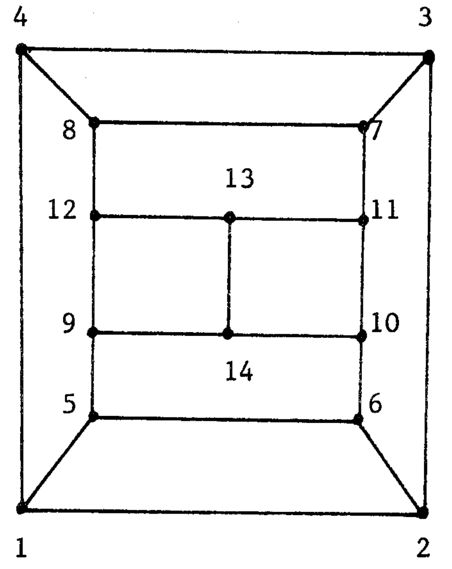


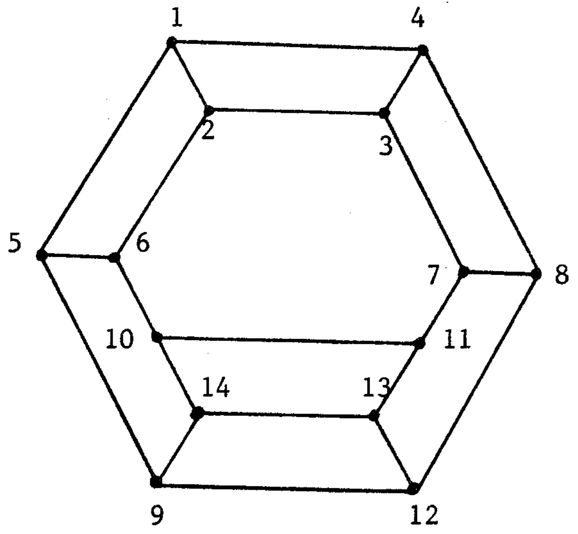
8 REGIONS : MAP 2 OF 2



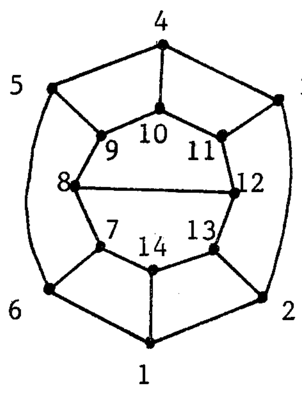


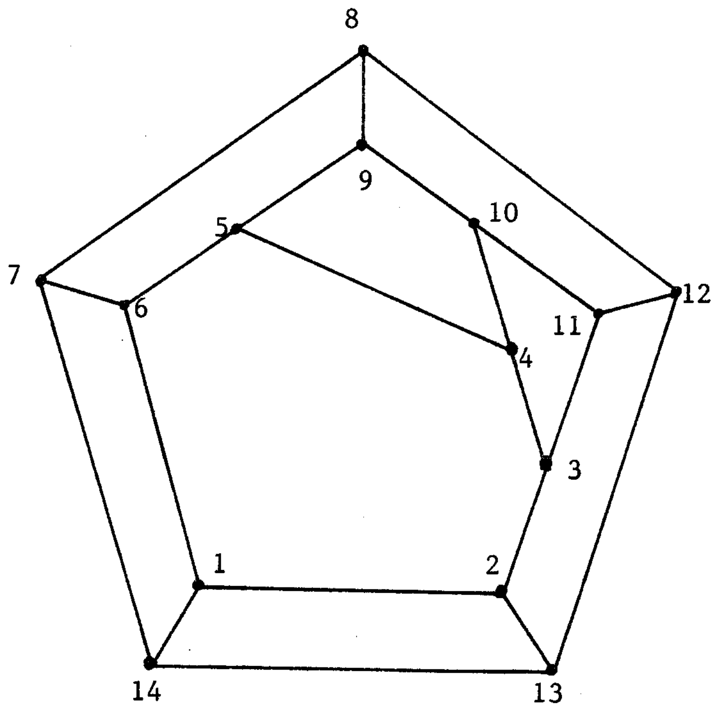
9 REGIONS : MAP 1 OF 4



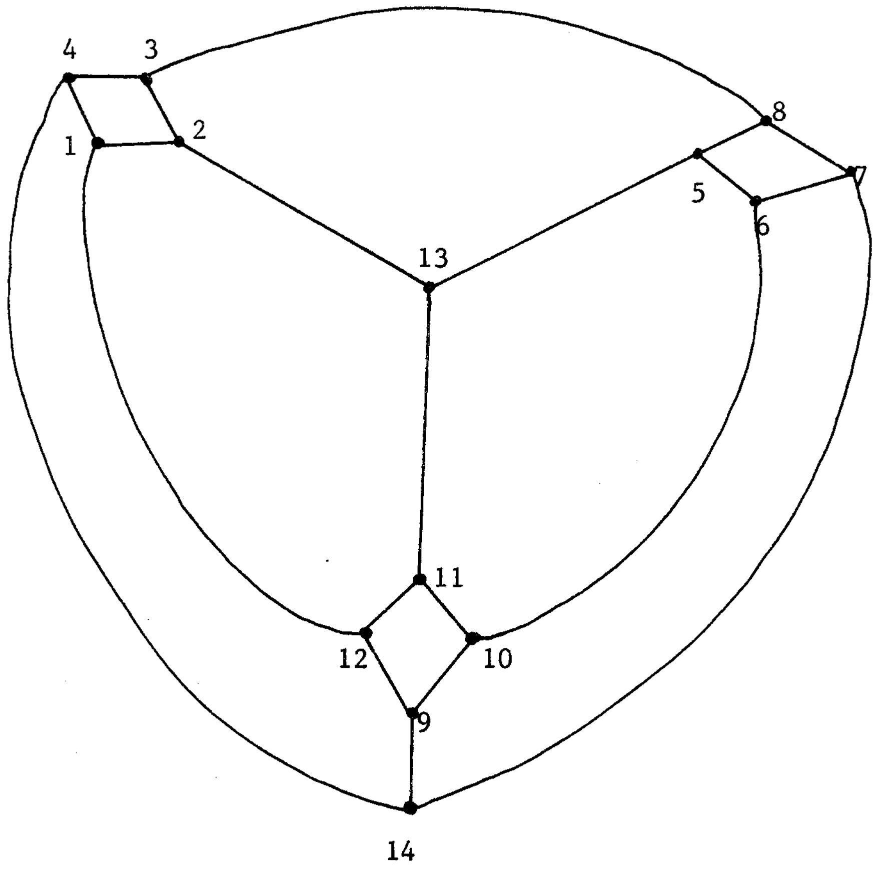


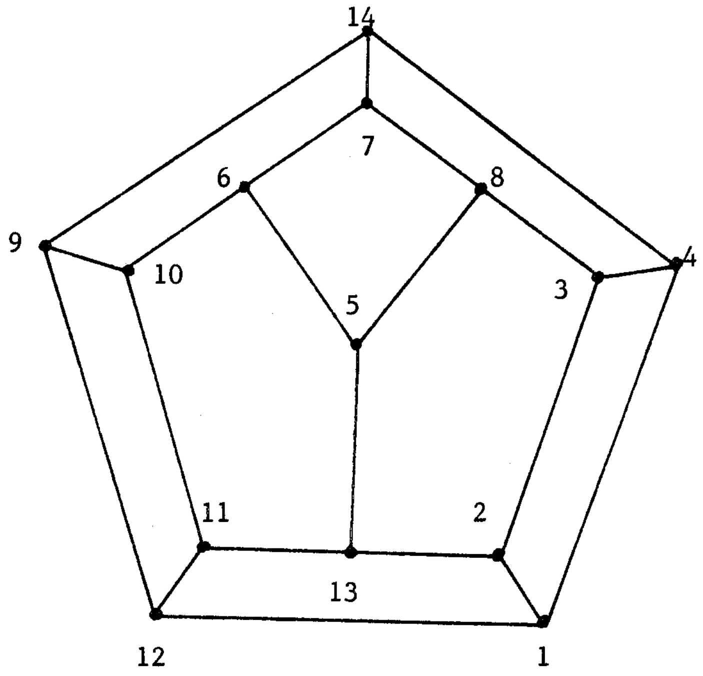
9 REGIONS : MAP 2 OF 4



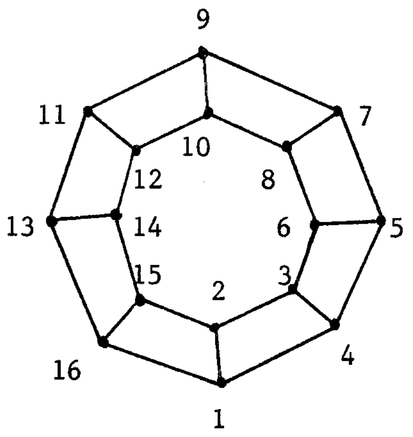


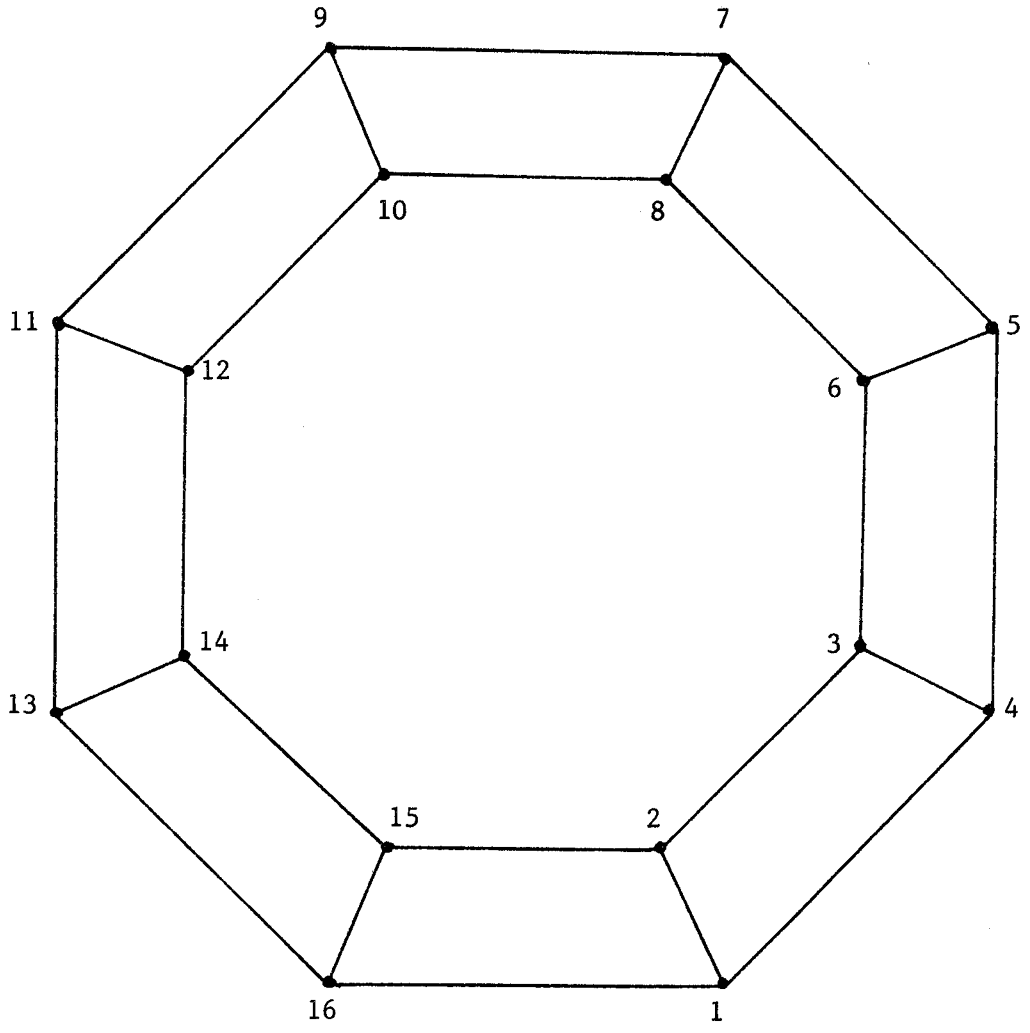
9 REGIONS : MAP 3 OF 4



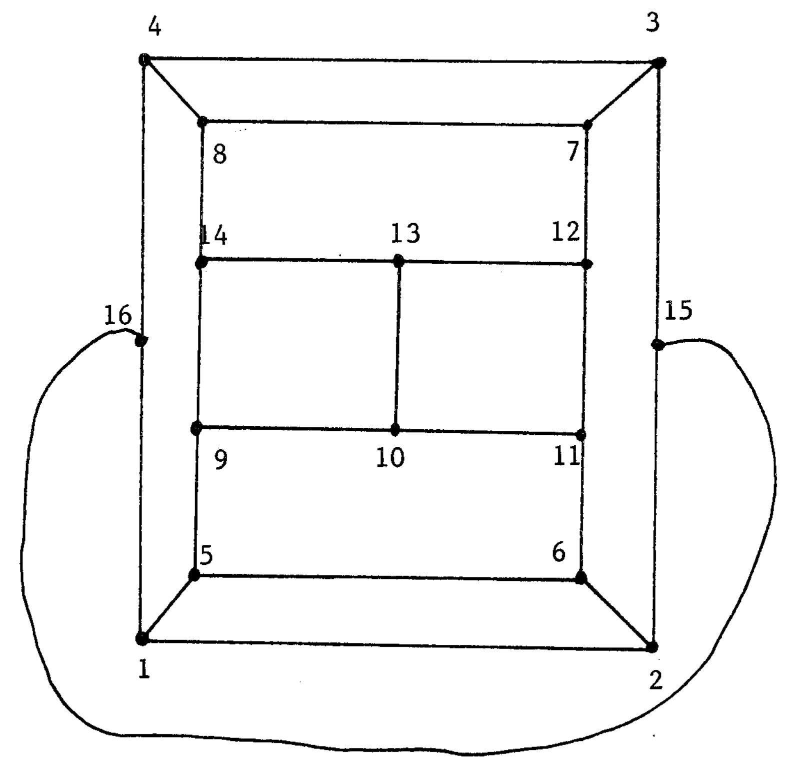


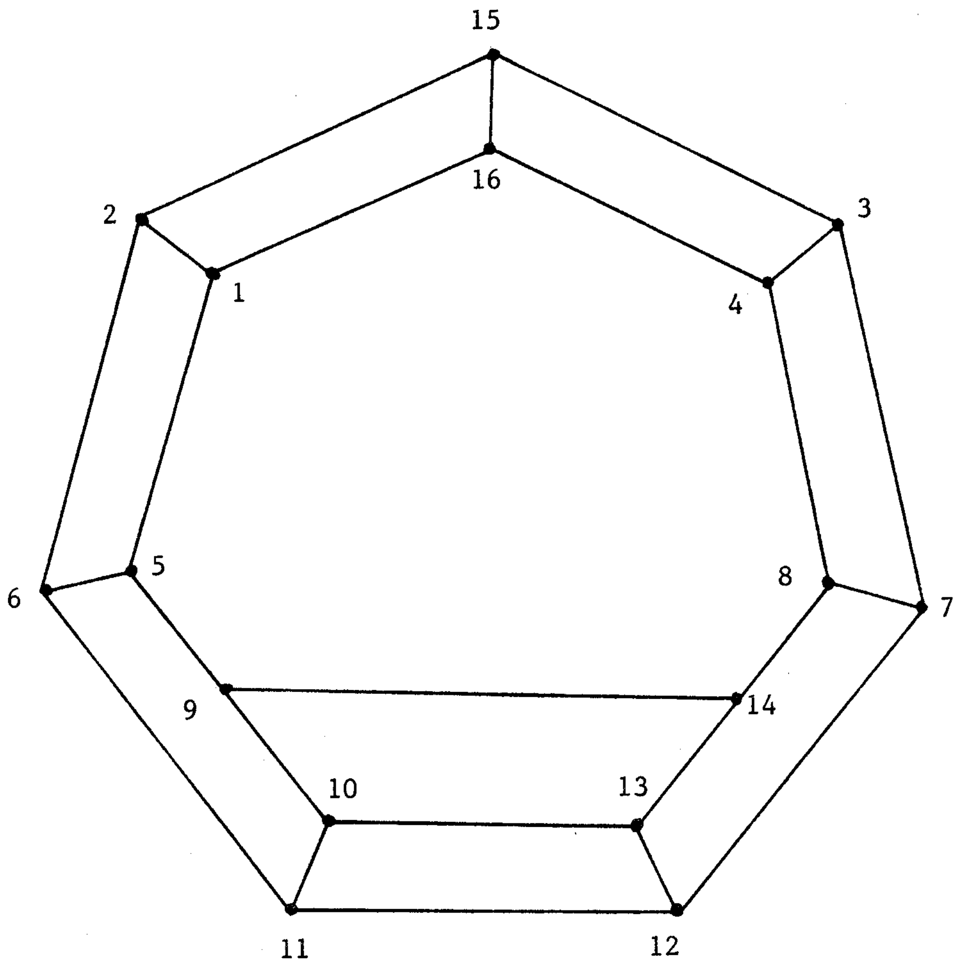
9 REGIONS : MAP 4 OF 4



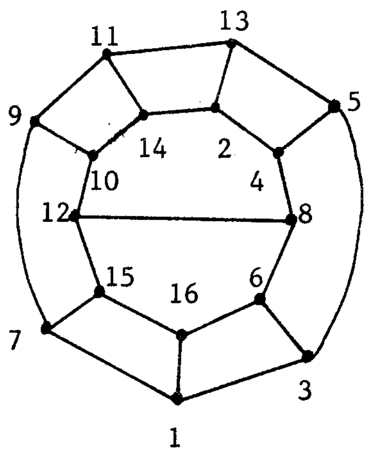


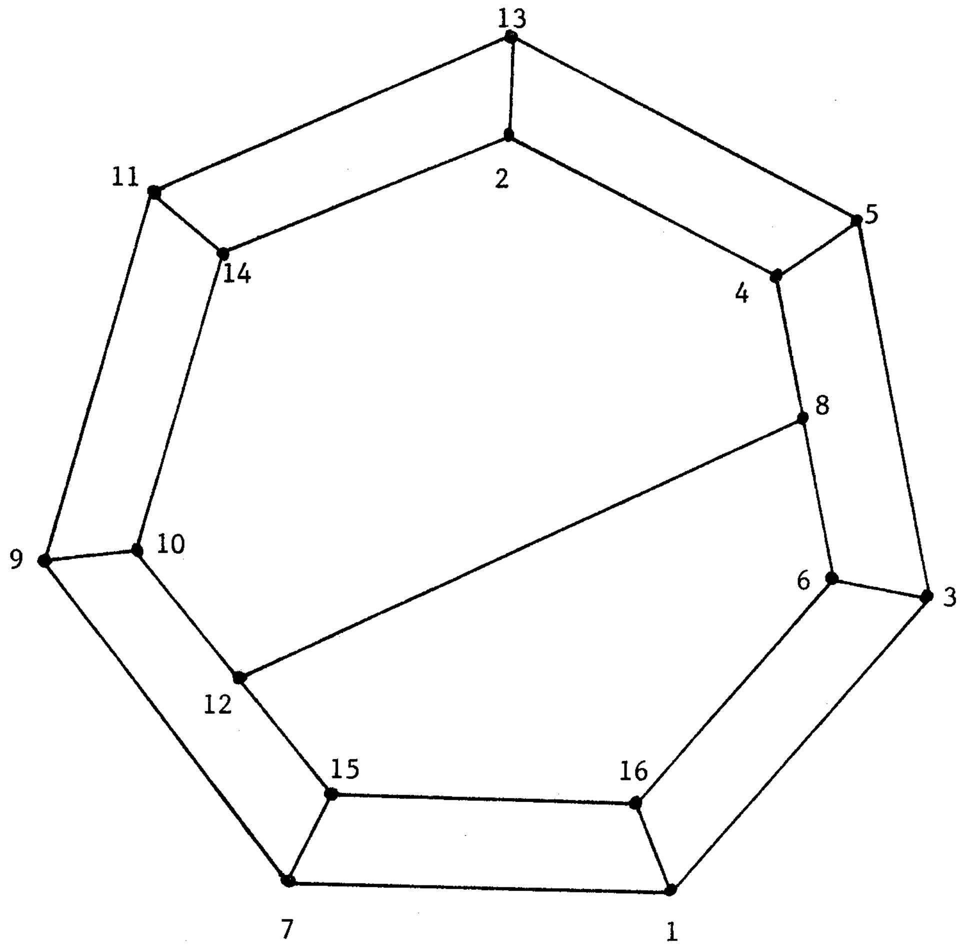
10 REGIONS : MAP 1 OF 10



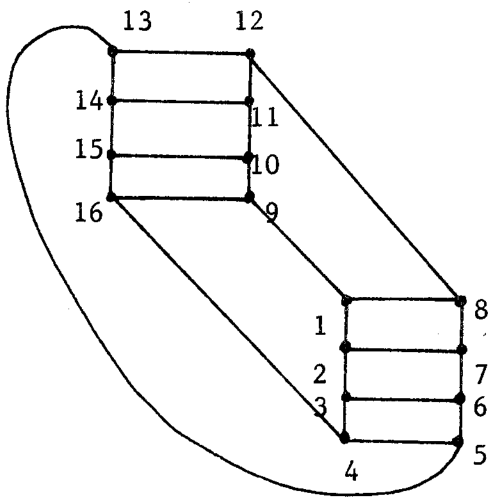


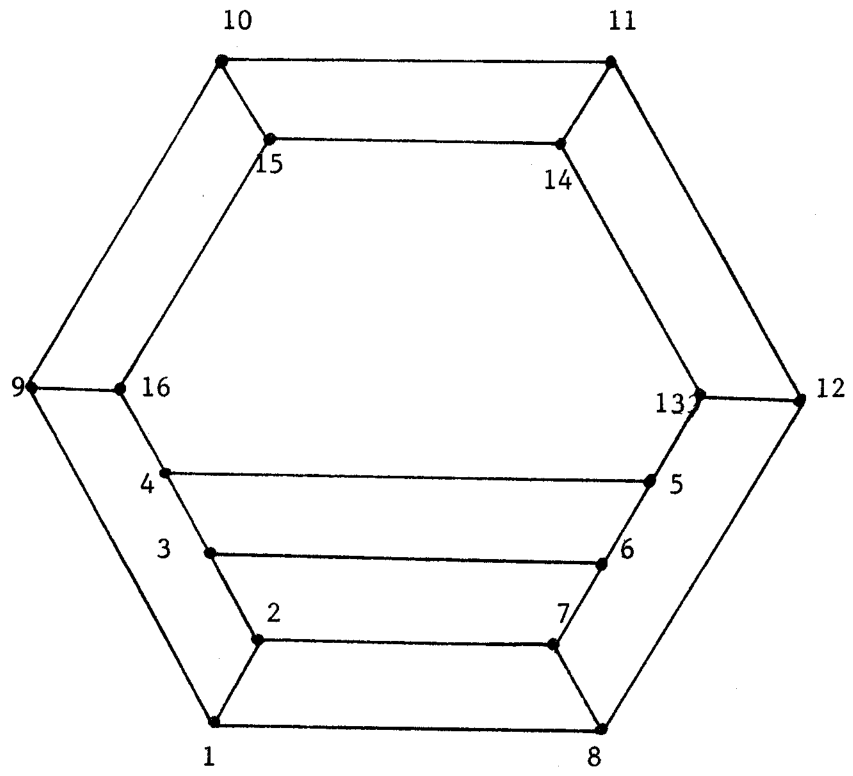
10 REGIONS : MAP 2 OF 10



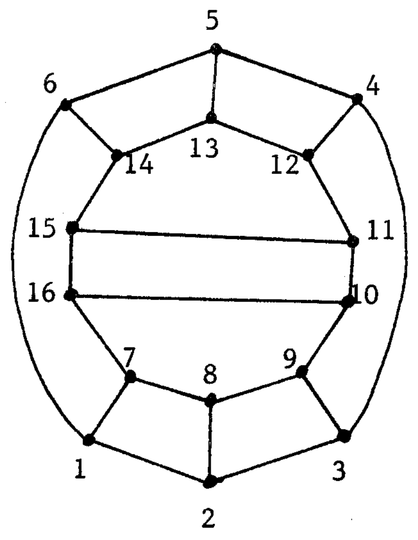


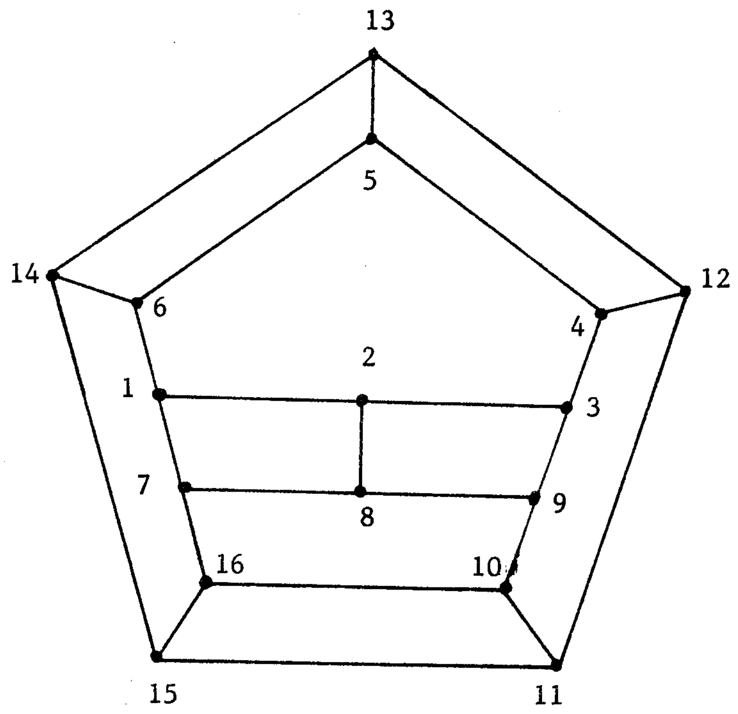
10 REGIONS : MAP 3 OF 10



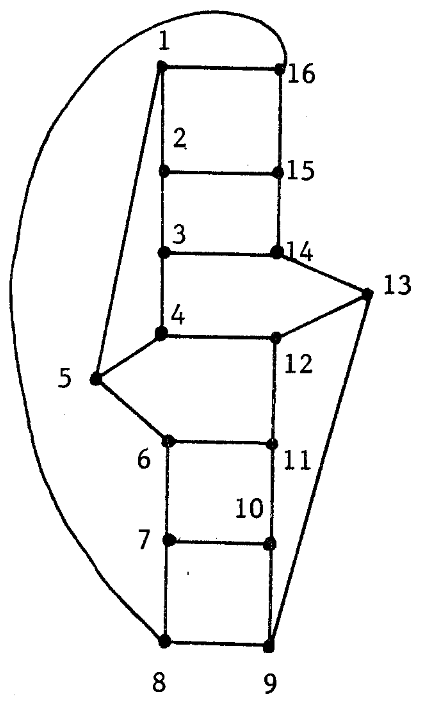


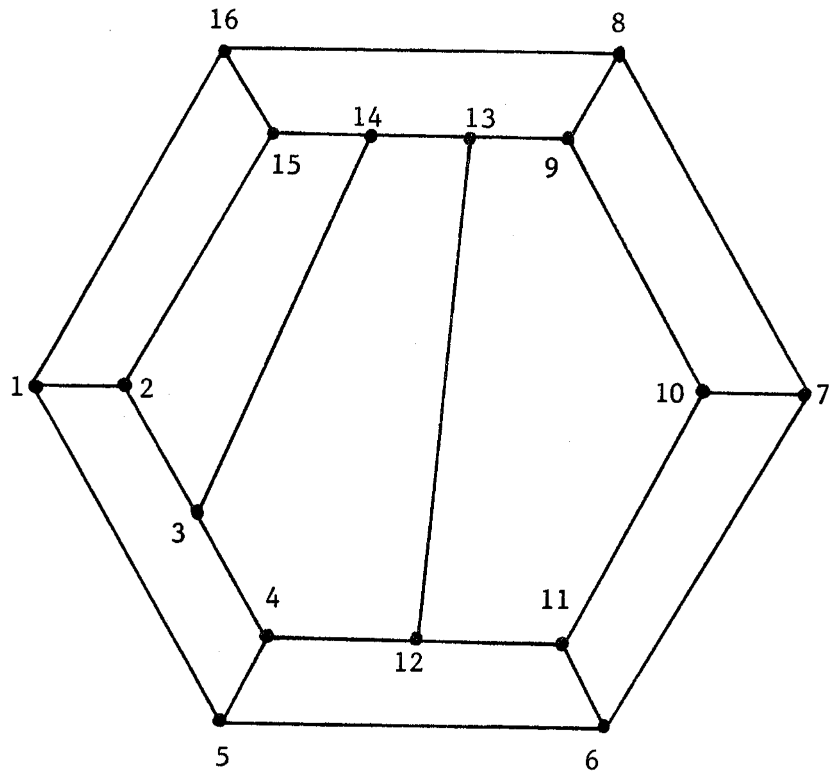
10 REGIONS : MAP 4 OF 10



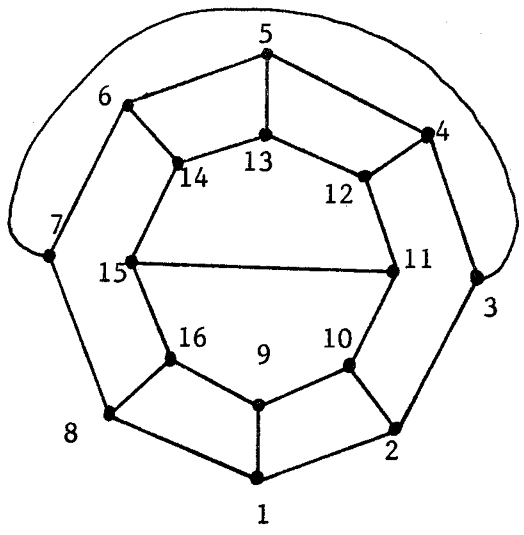


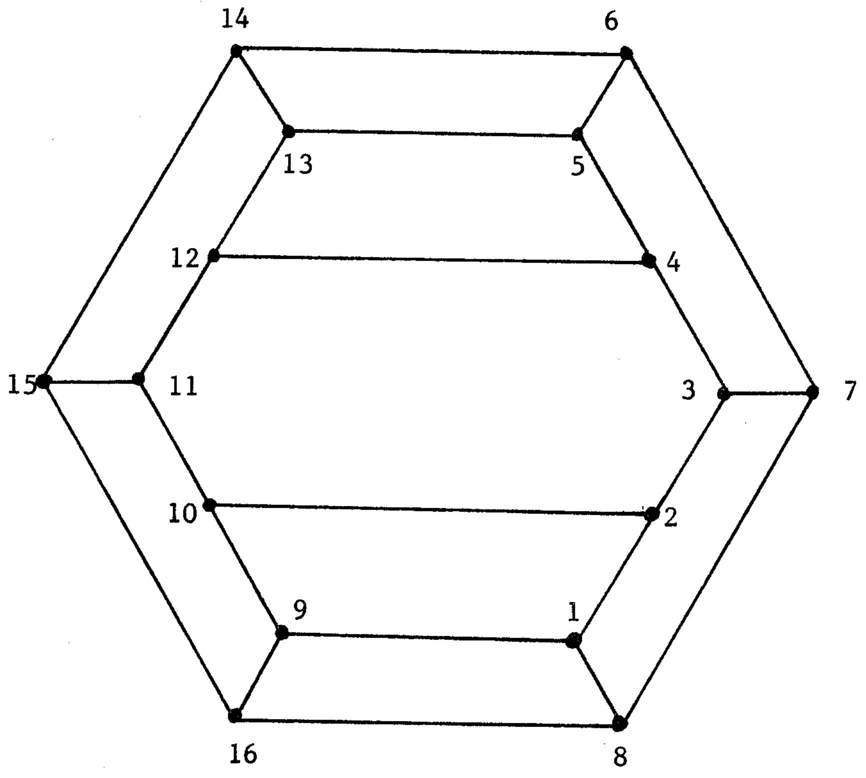
10 REGIONS : MAP 5 OF 10



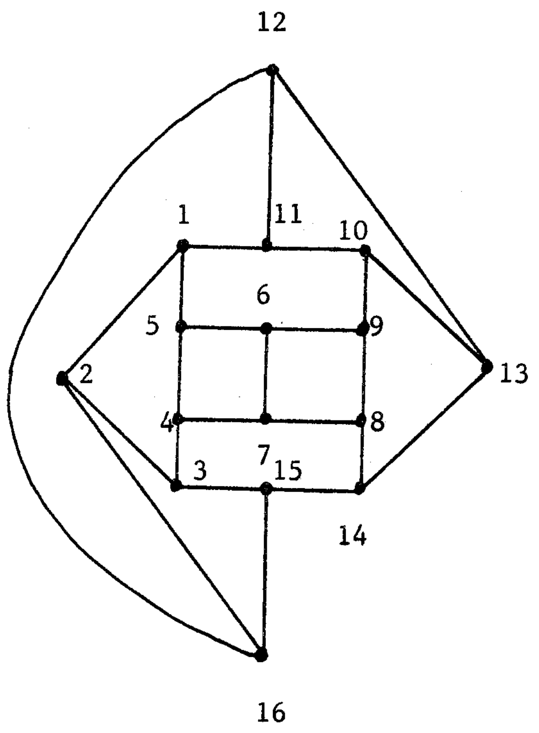


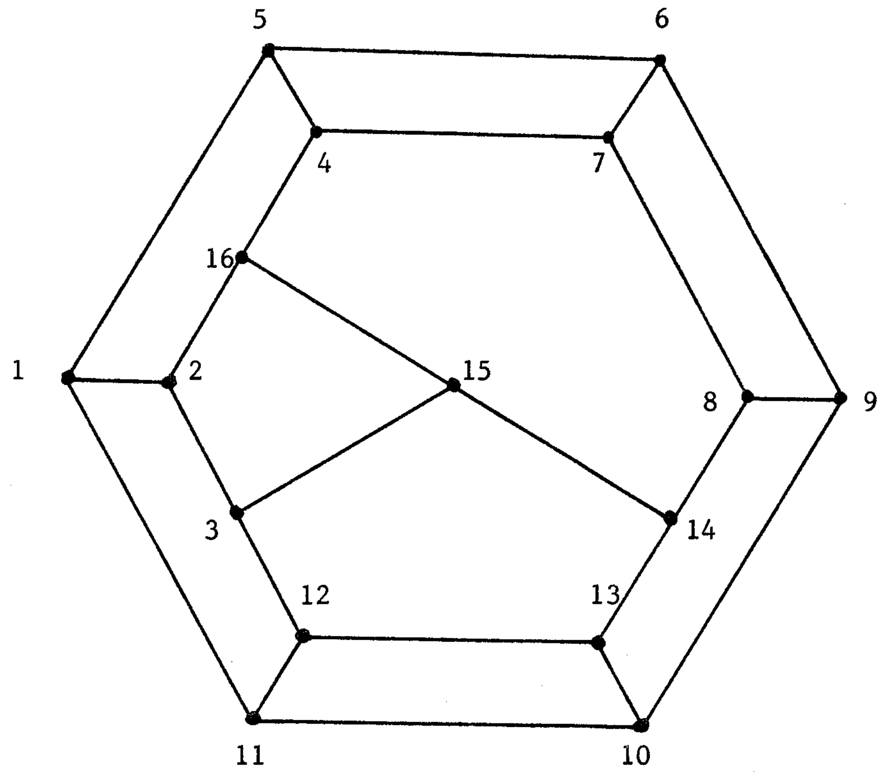
10 REGIONS : MAP 6 OF 10



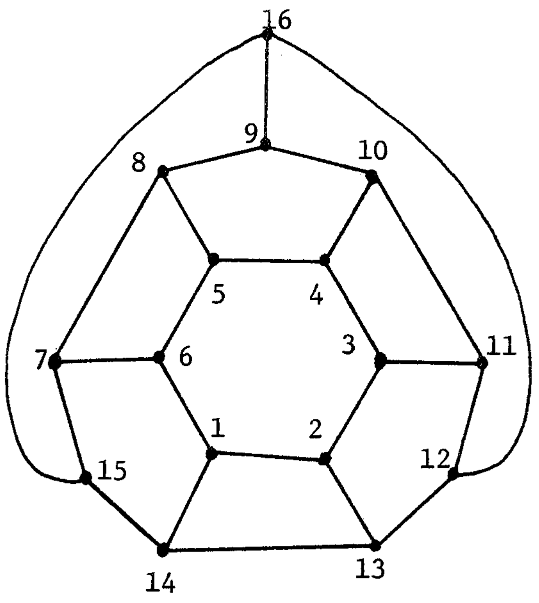


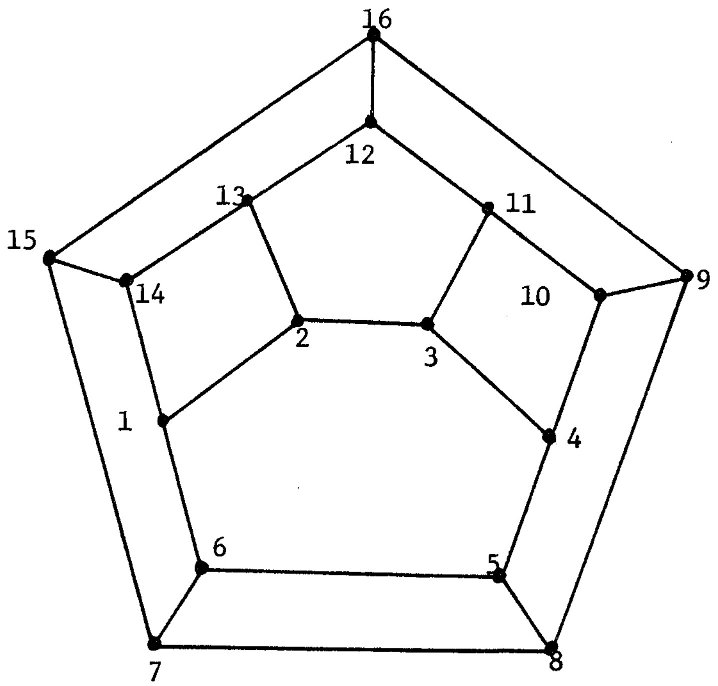
10 REGIONS : MAP 7 OF 10



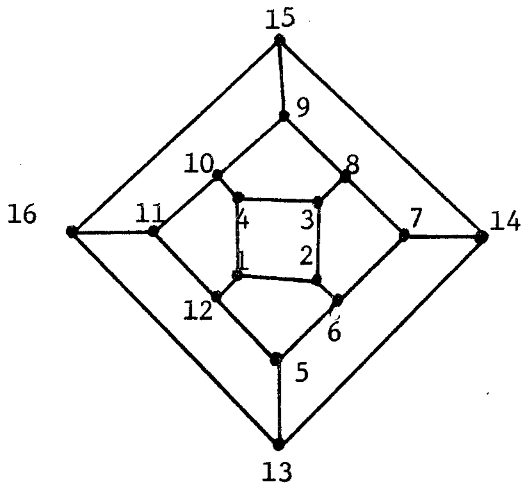


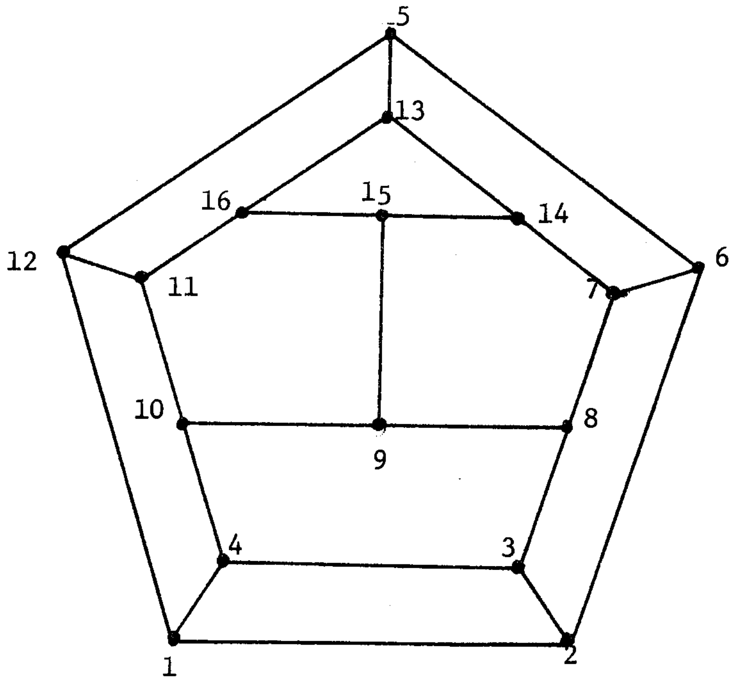
10 REGIONS : MAP 8 OF 10



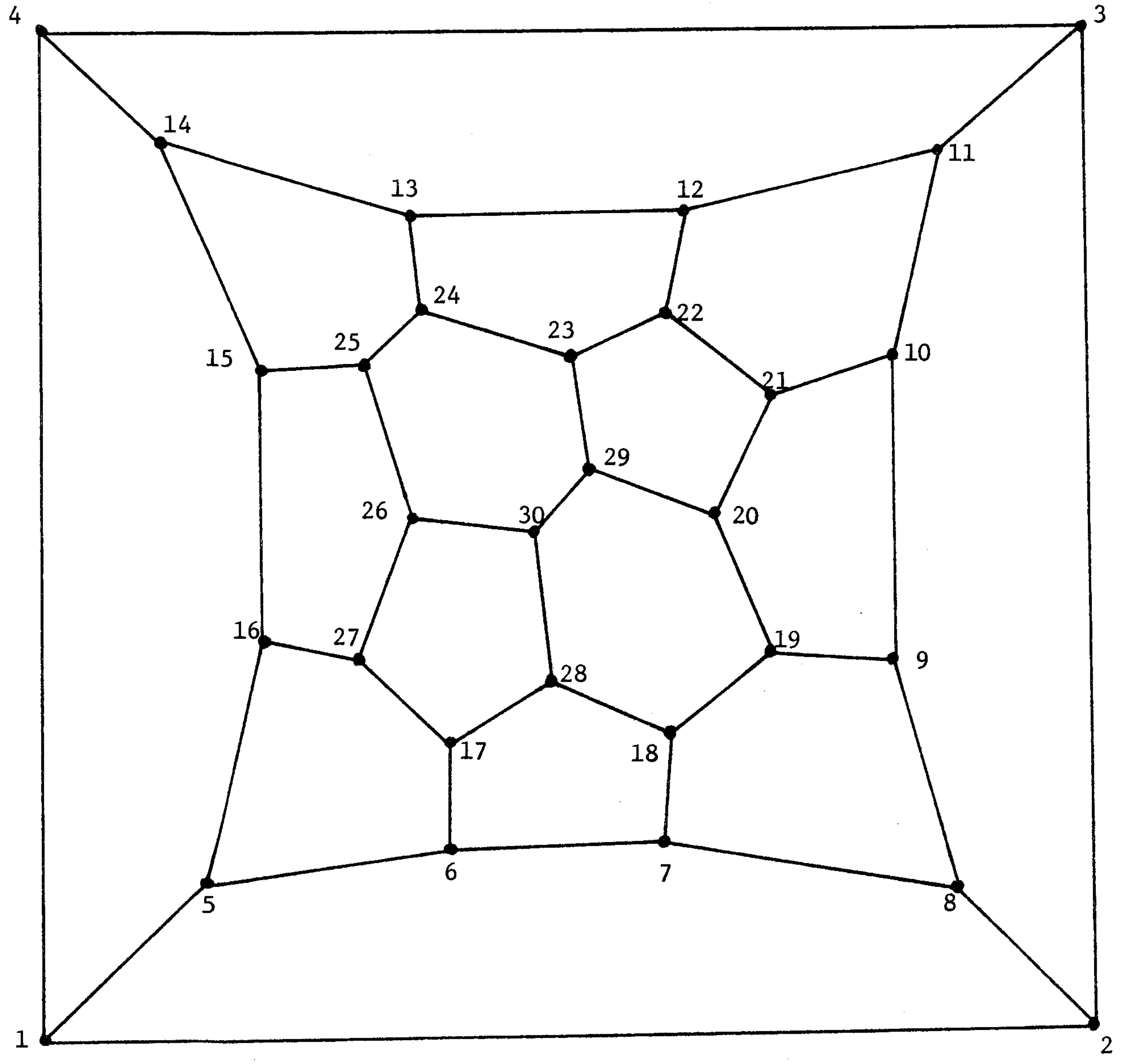


10 REGIONS : MAP 9 OF 10

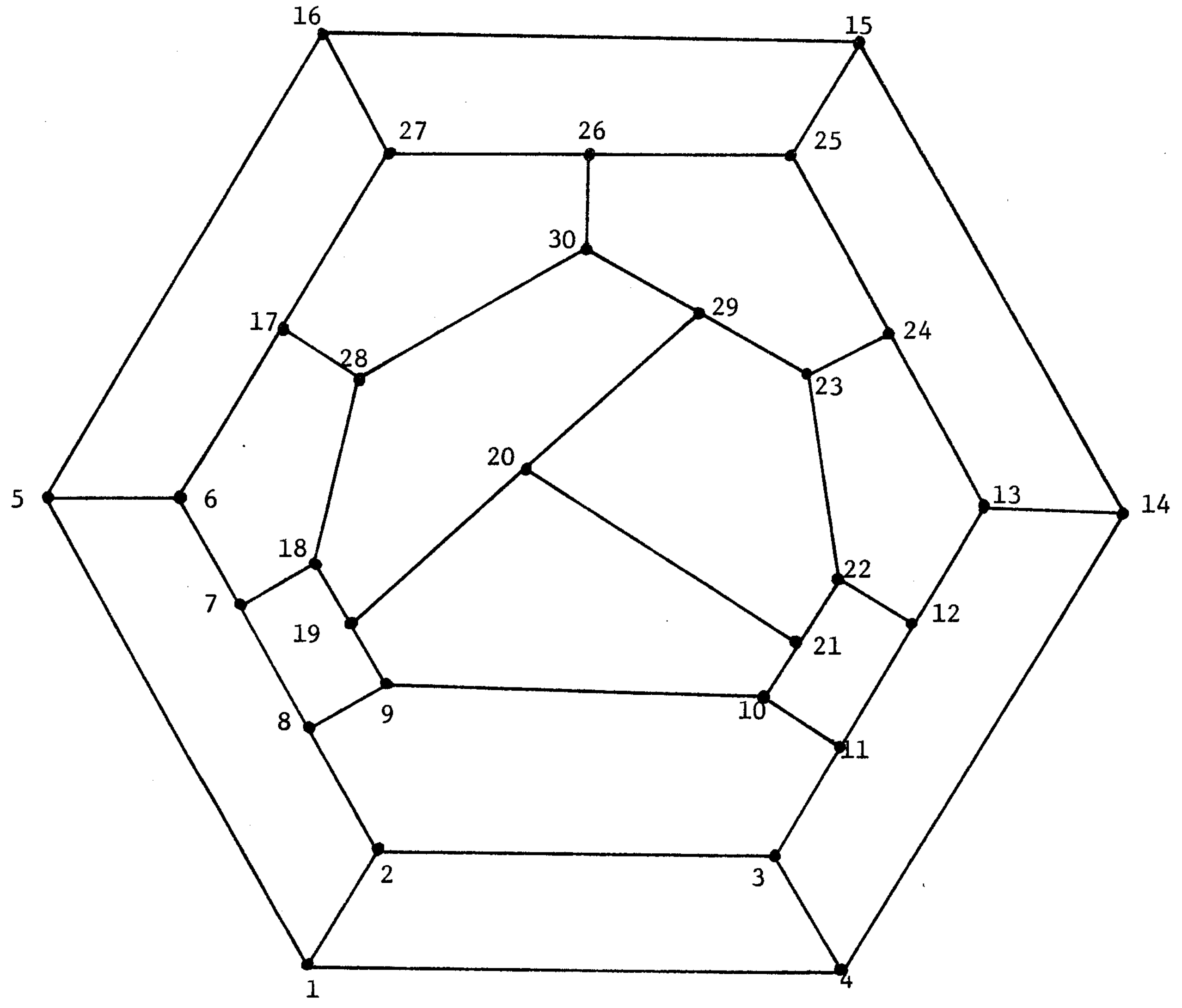




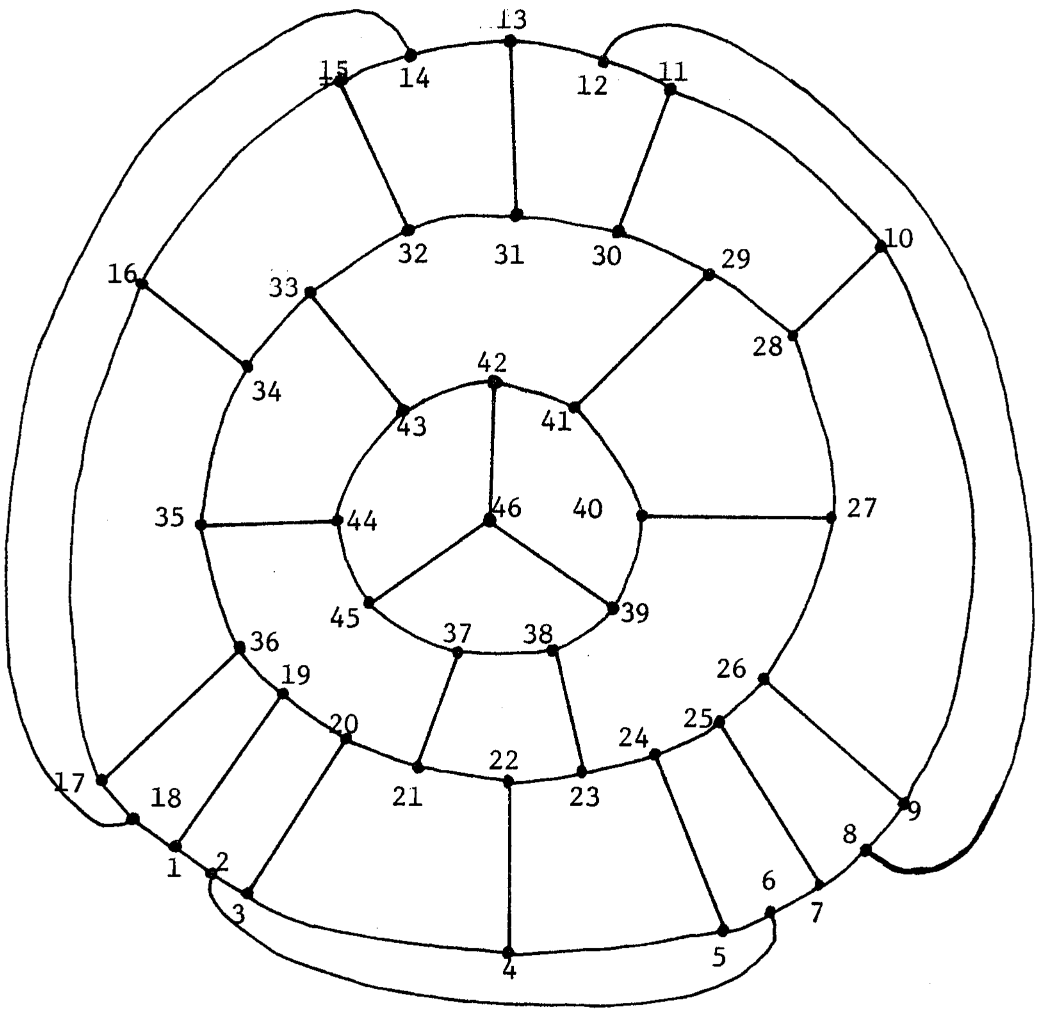
10 REGIONS : MAP 10 OF 10



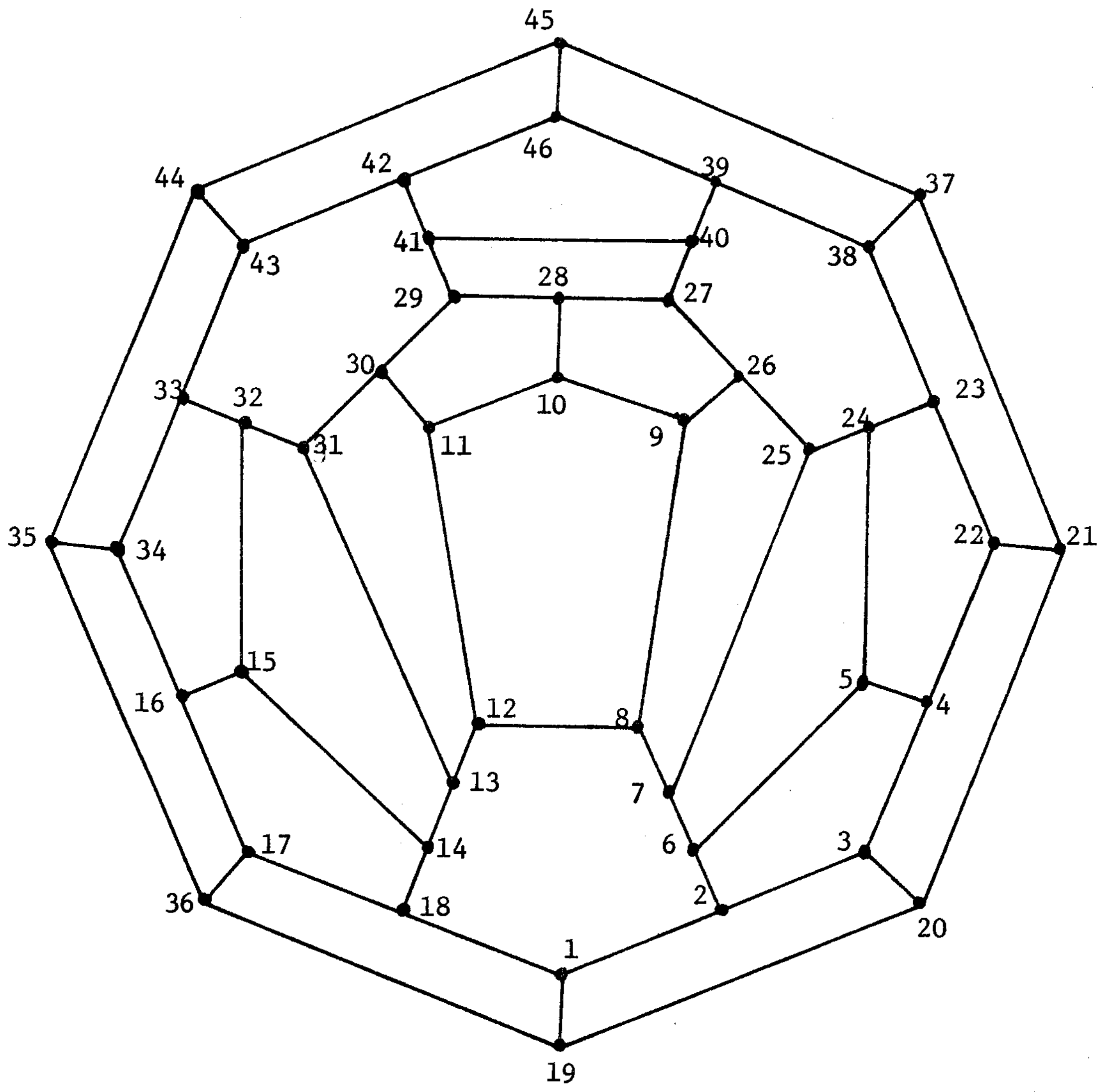
17 REGIONS HAND-GENERATED



17 REGIONS PROGRAM-GENERATED



25 REGIONS HAND-GENERATED

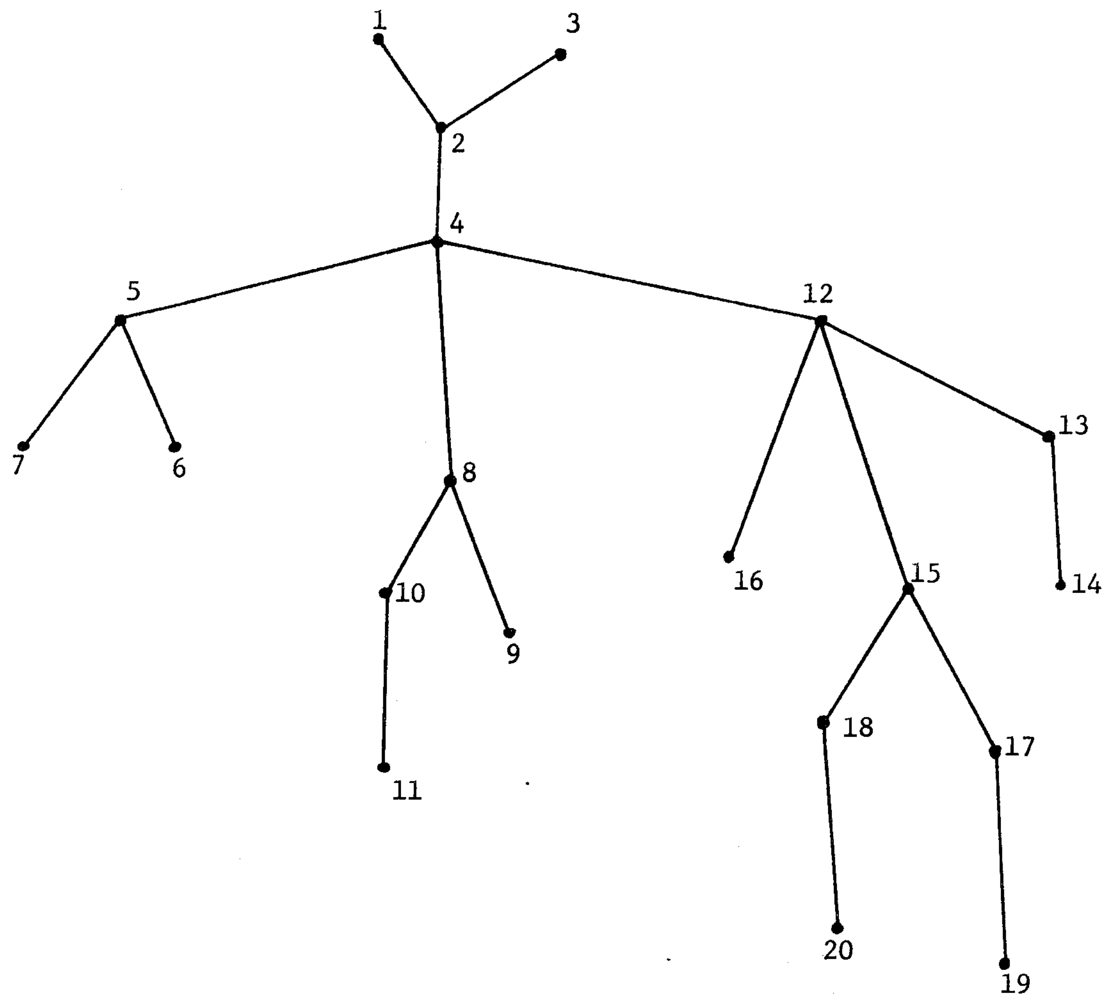


25 REGIONS PROGRAM-GENERATED

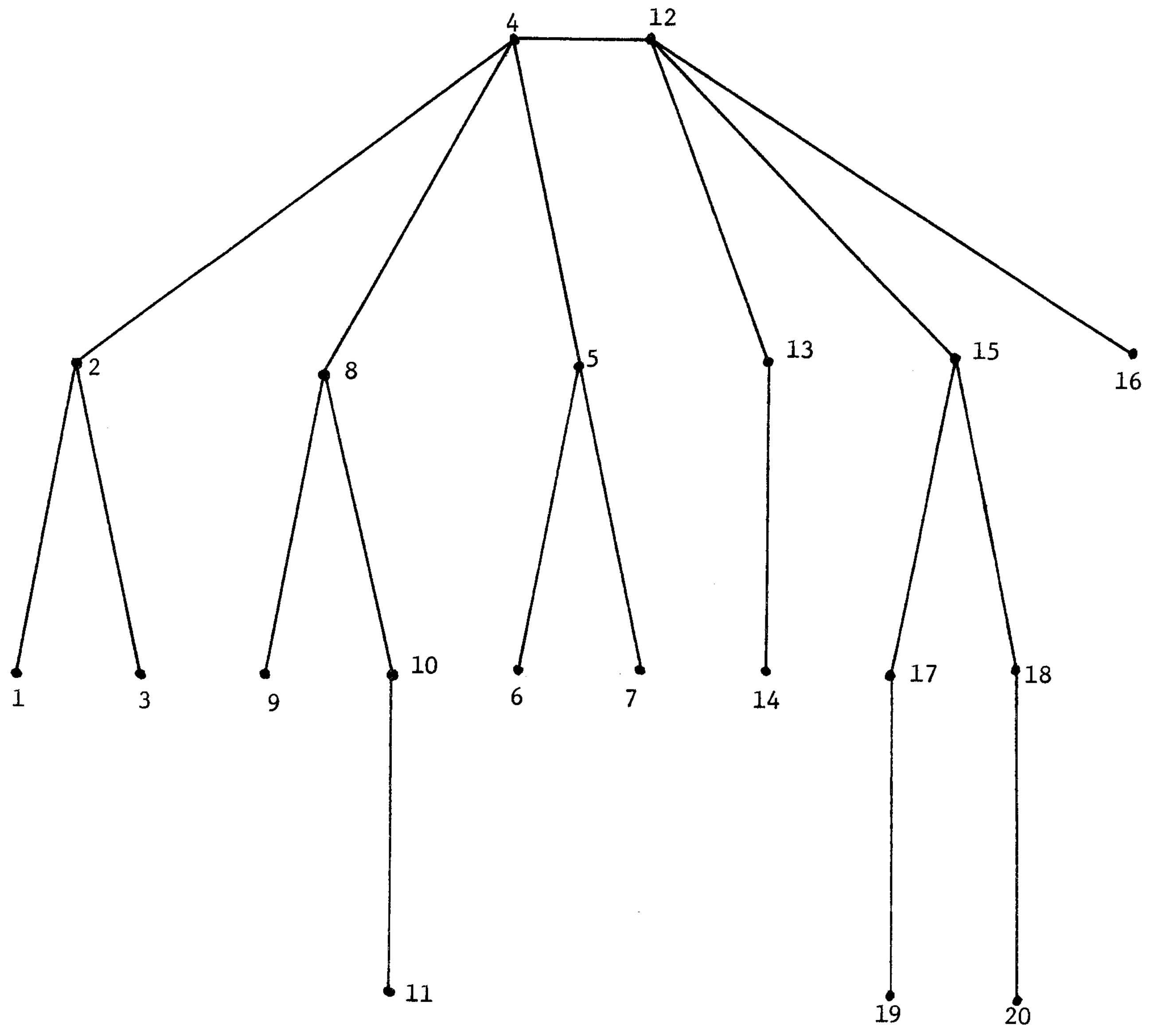
# A P P E N D I X   C

## APPENDIX C : SOME GRAPH REPRESENTATIONS

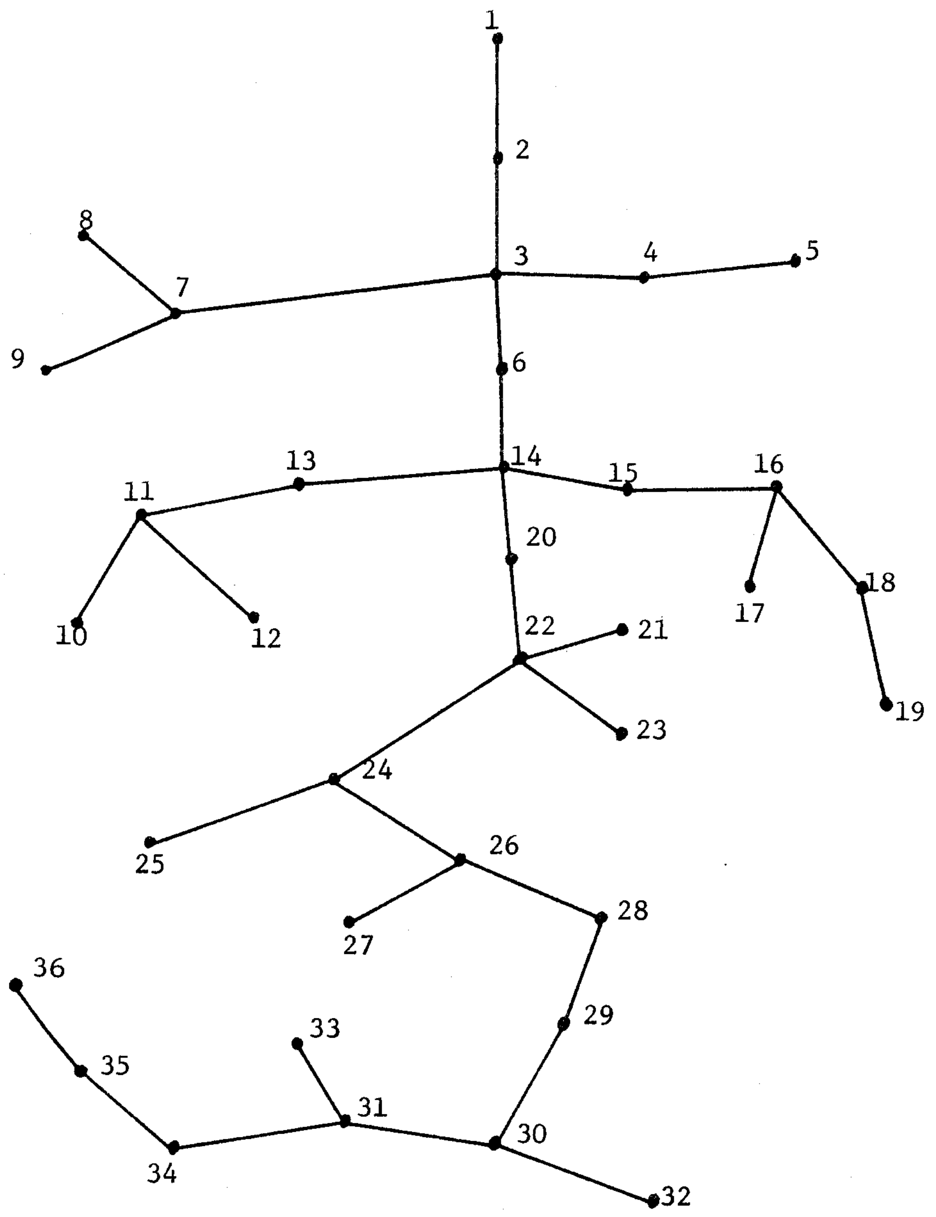
The graphs contained in this appendix were chosen at random from several of the references on graph theory and graph-theoretical applications listed in the bibliography. Each graph is first shown as it was originally pictured and then it is shown in the form of a graph representation generated by either the General Representation Routine or the Tree Representation Routine. These representations indicate that by using these routines it is possible to obtain comparable, if not better, representations than the original hand-drawn versions. The Tree Representation Routine was used for all the graphs which are trees. The postediting facilities were used in creating most of the representations.



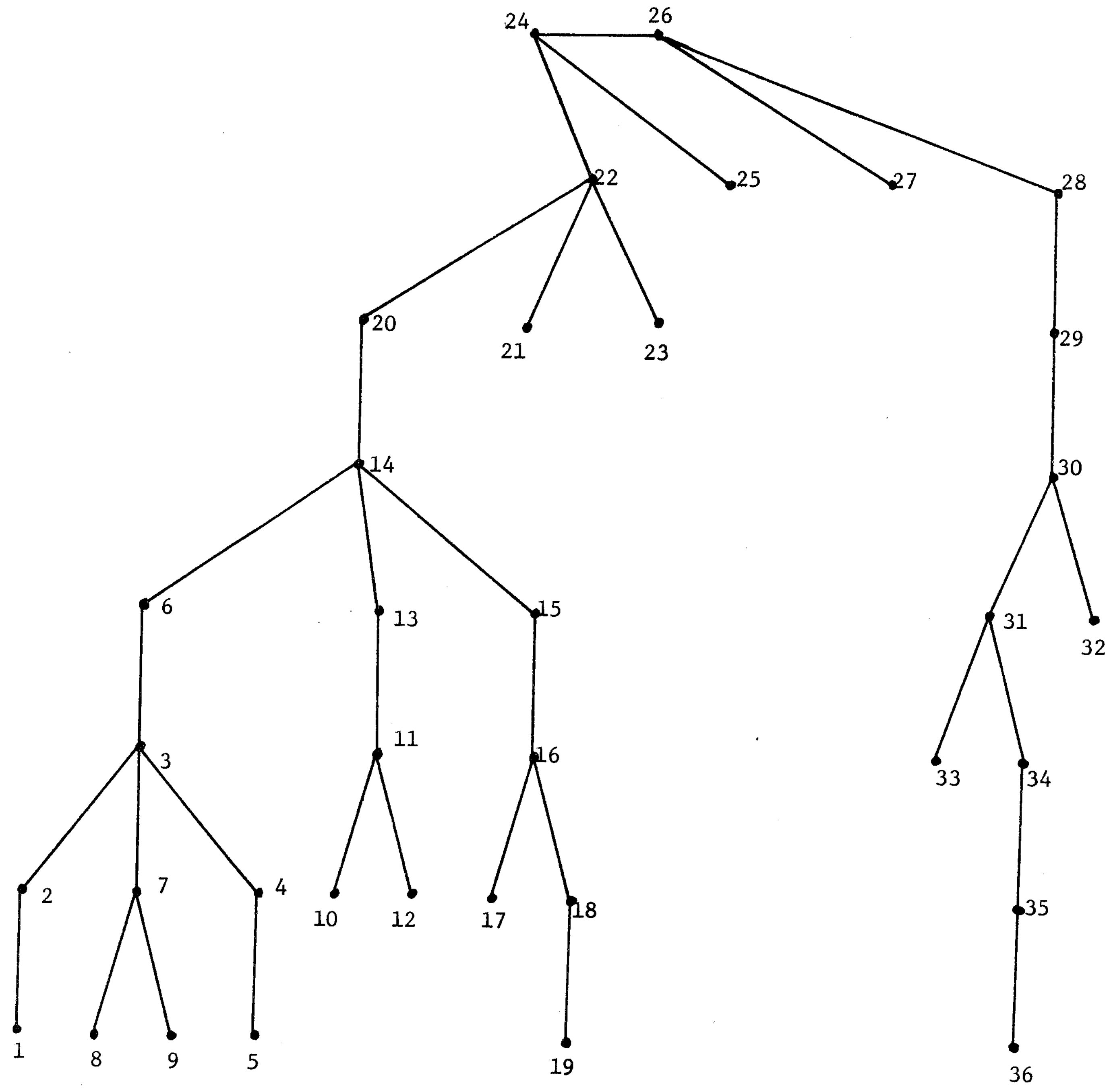
ORIGINAL PICTURE



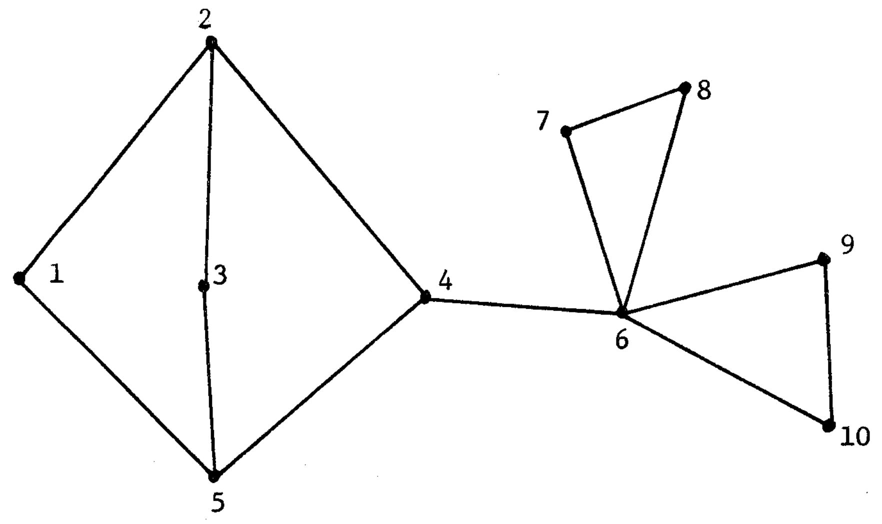
GENERATED REPRESENTATION



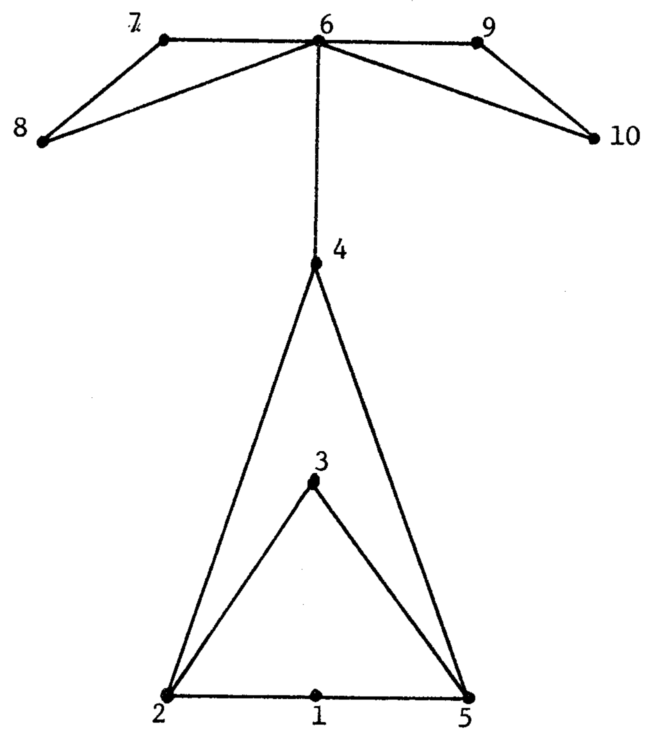
ORIGINAL PICTURE



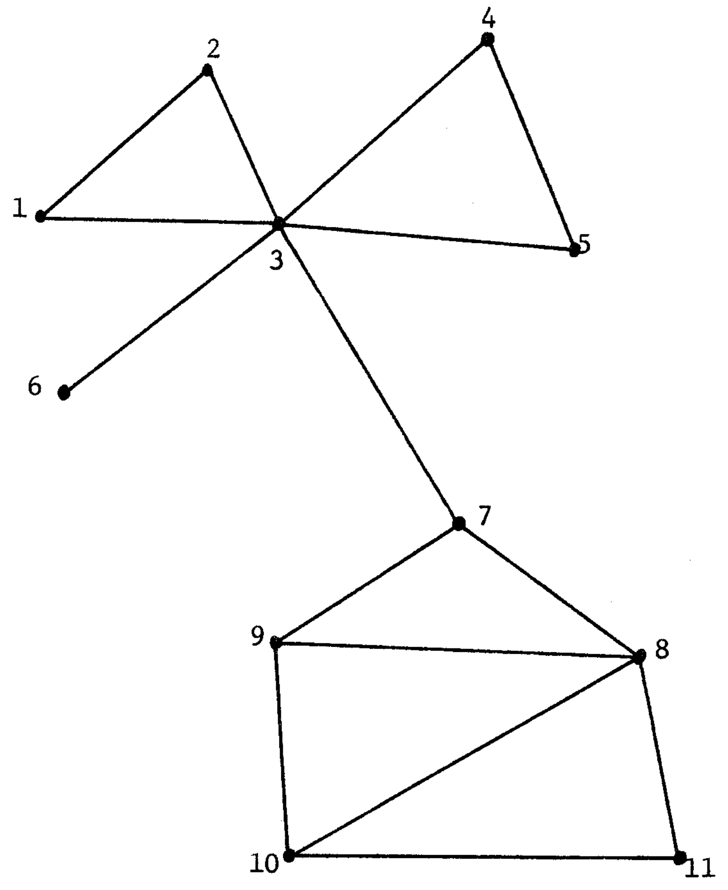
GENERATED REPRESENTATION



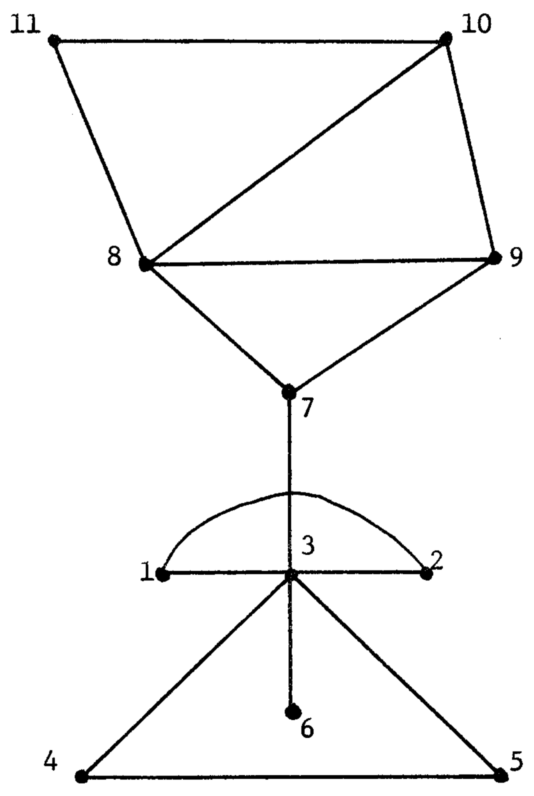
ORIGINAL PICTURE



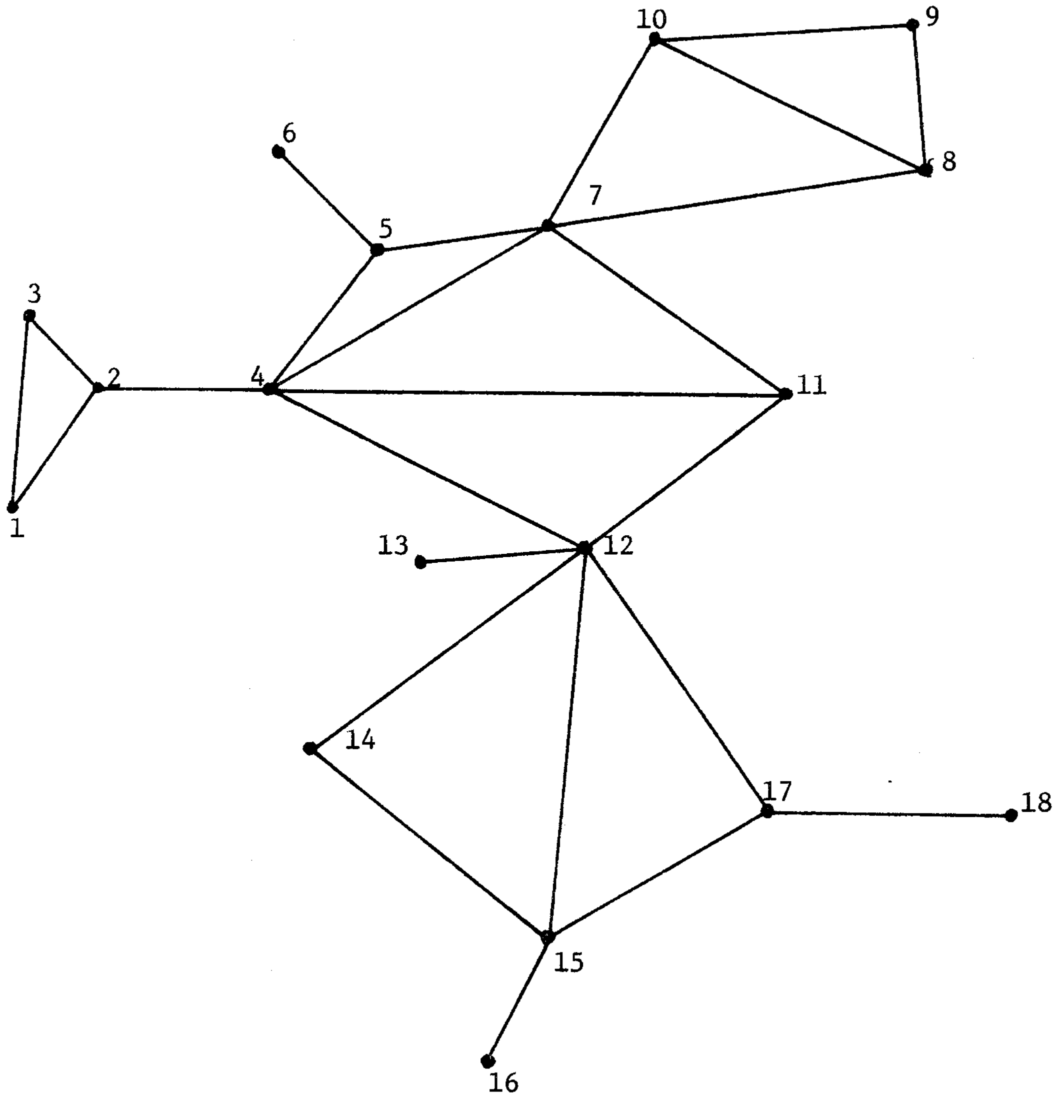
GENERATED REPRESENTATION



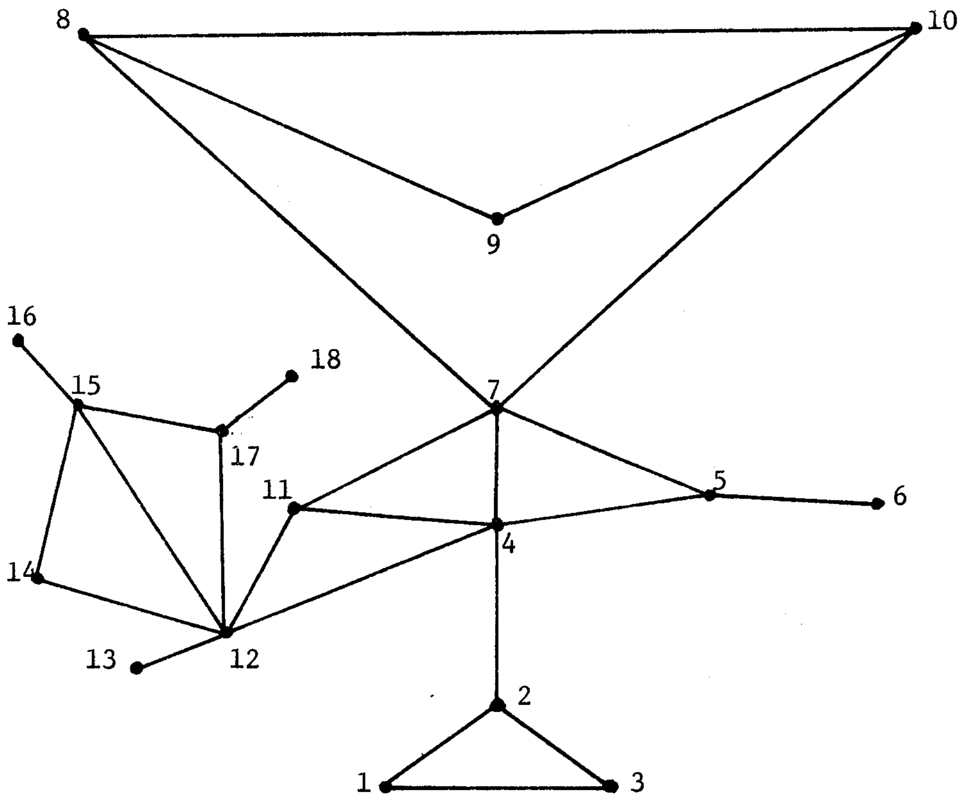
ORIGINAL PICTURE



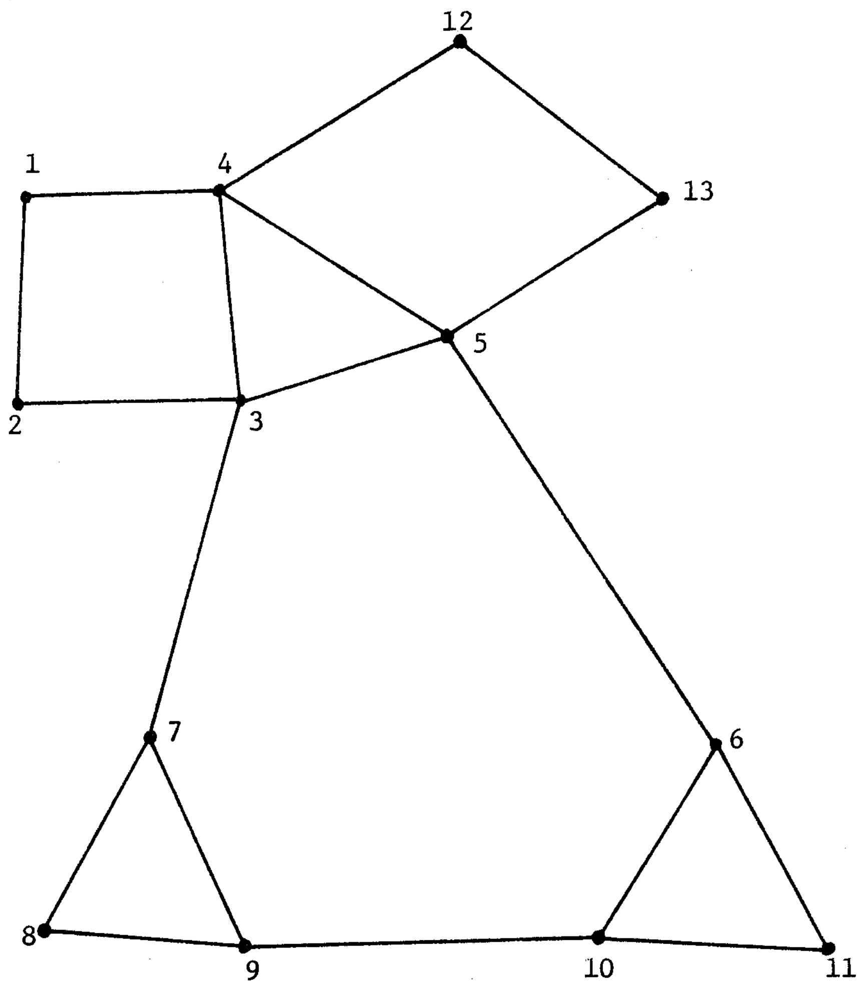
GENERATED REPRESENTATION



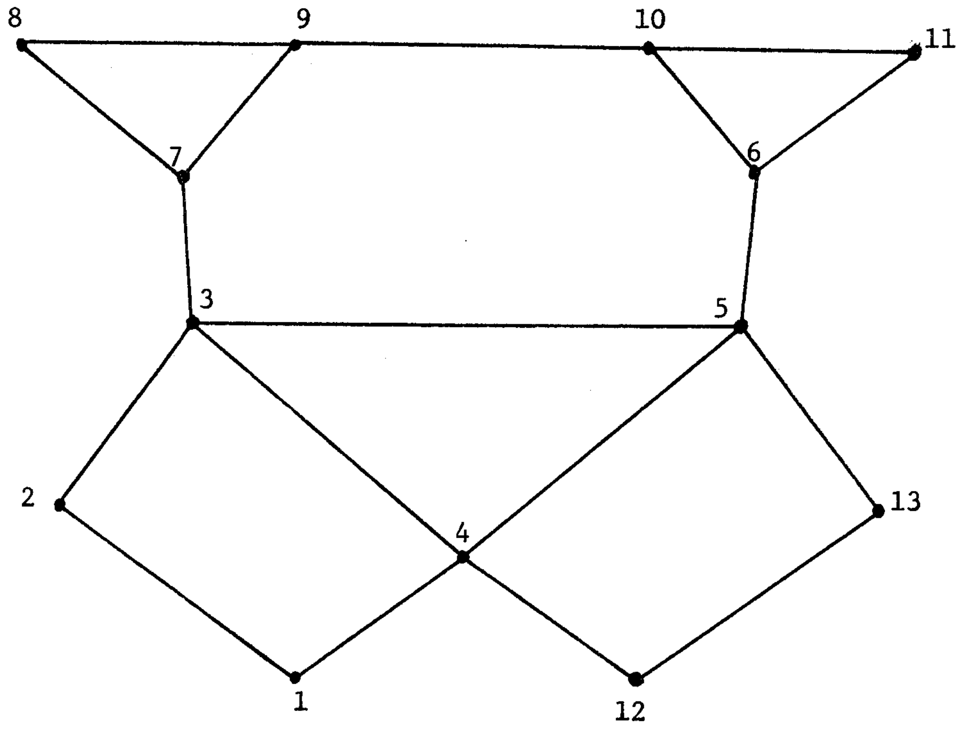
ORIGINAL PICTURE



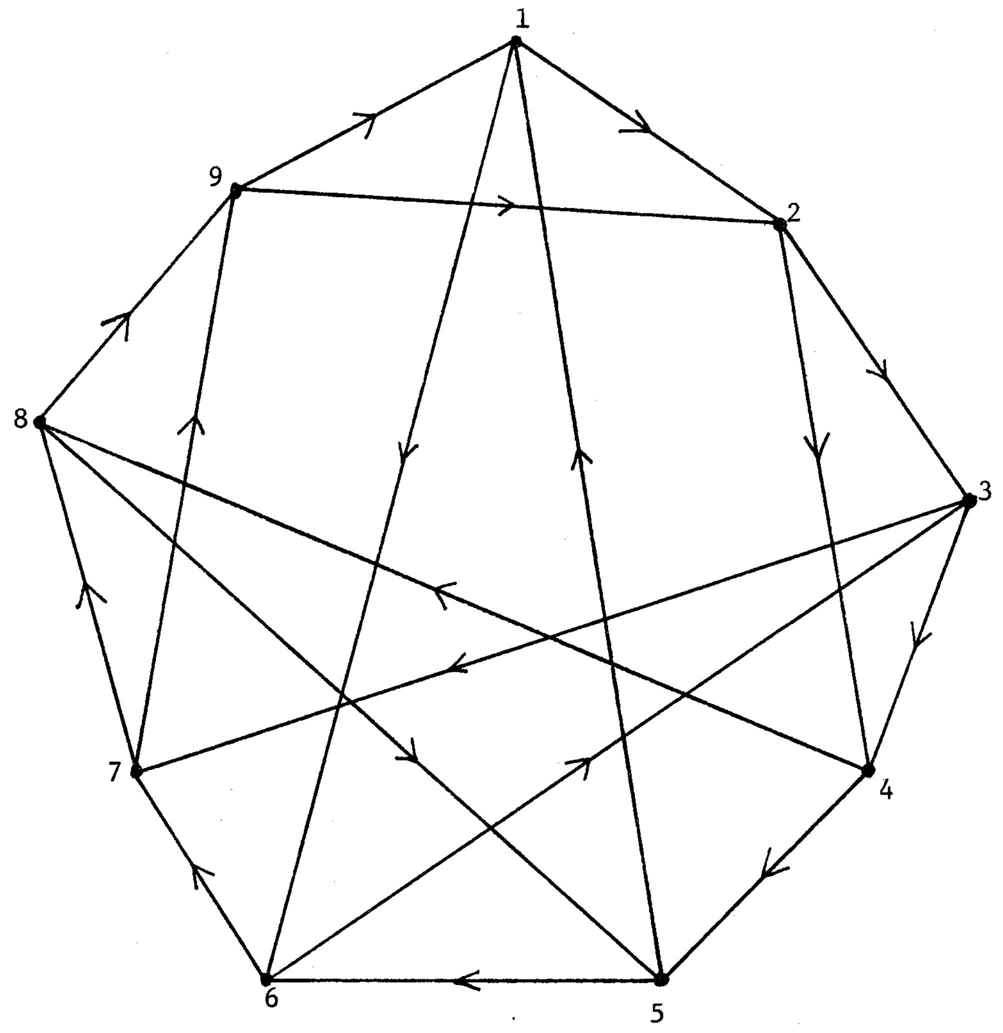
GENERATED REPRESENTATION



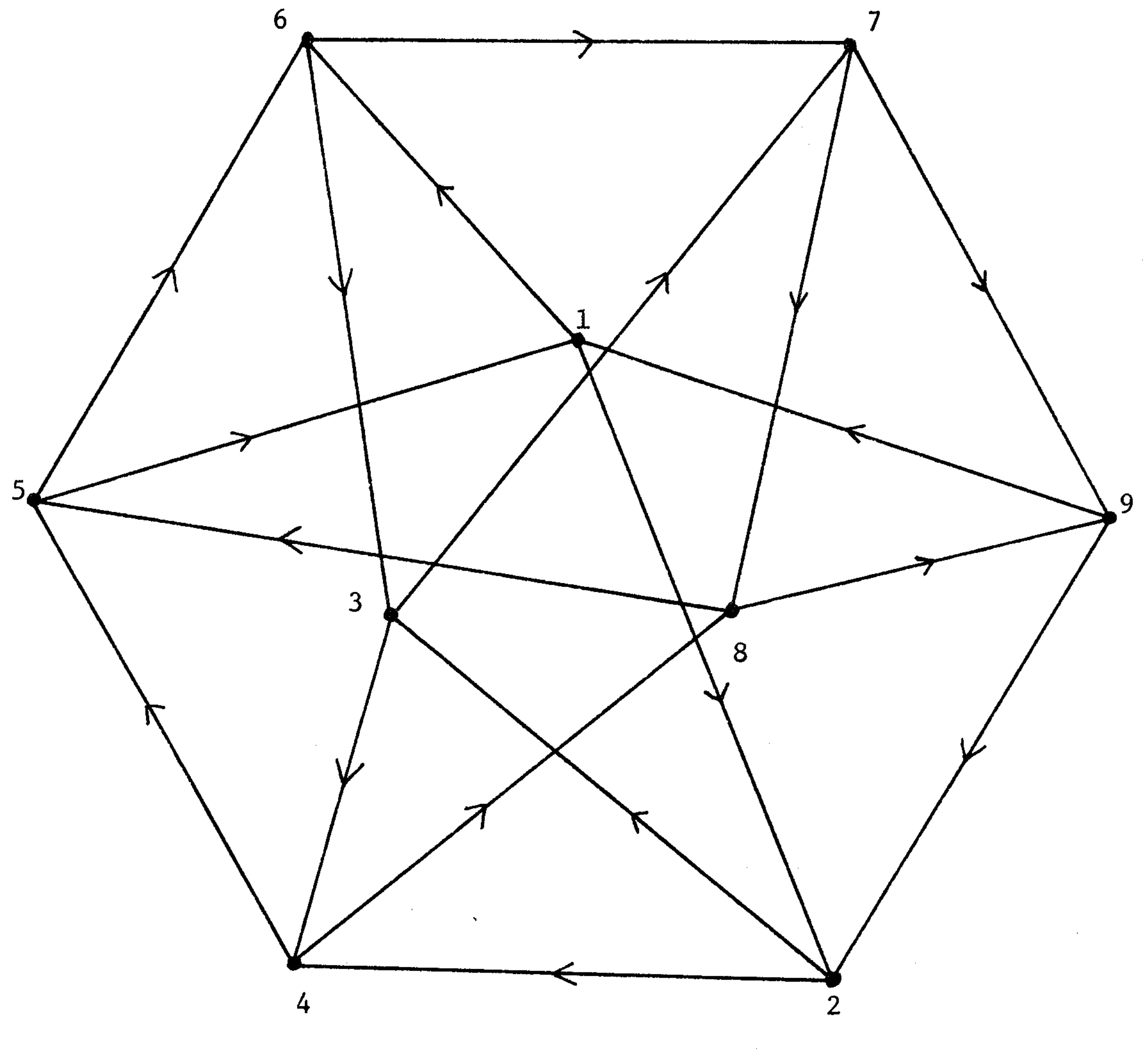
ORIGINAL PICTURE



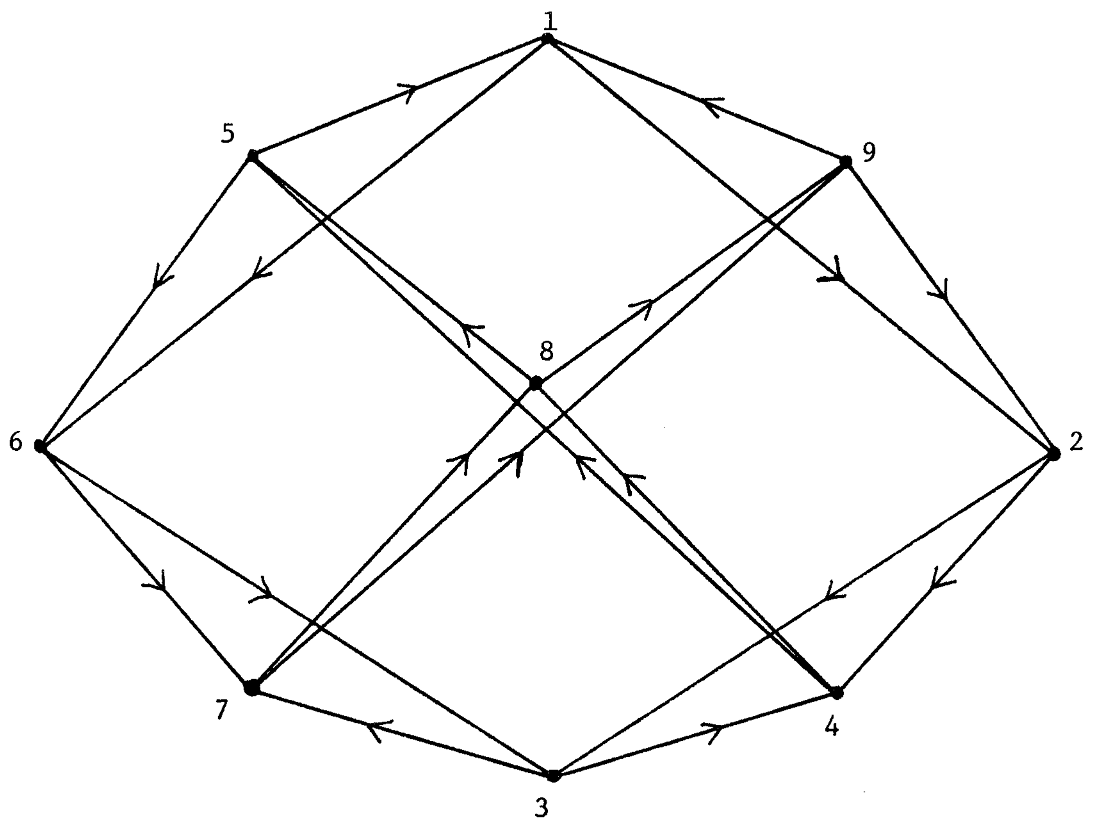
GENERATED REPRESENTATION



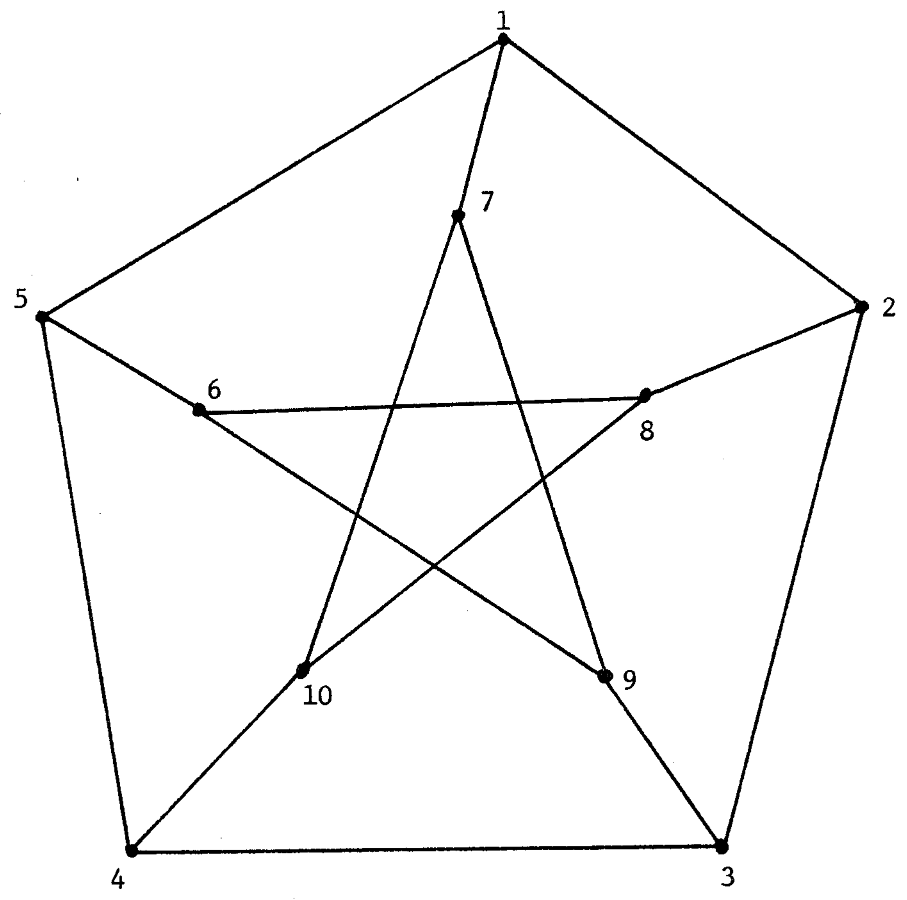
ORIGINAL PICTURE



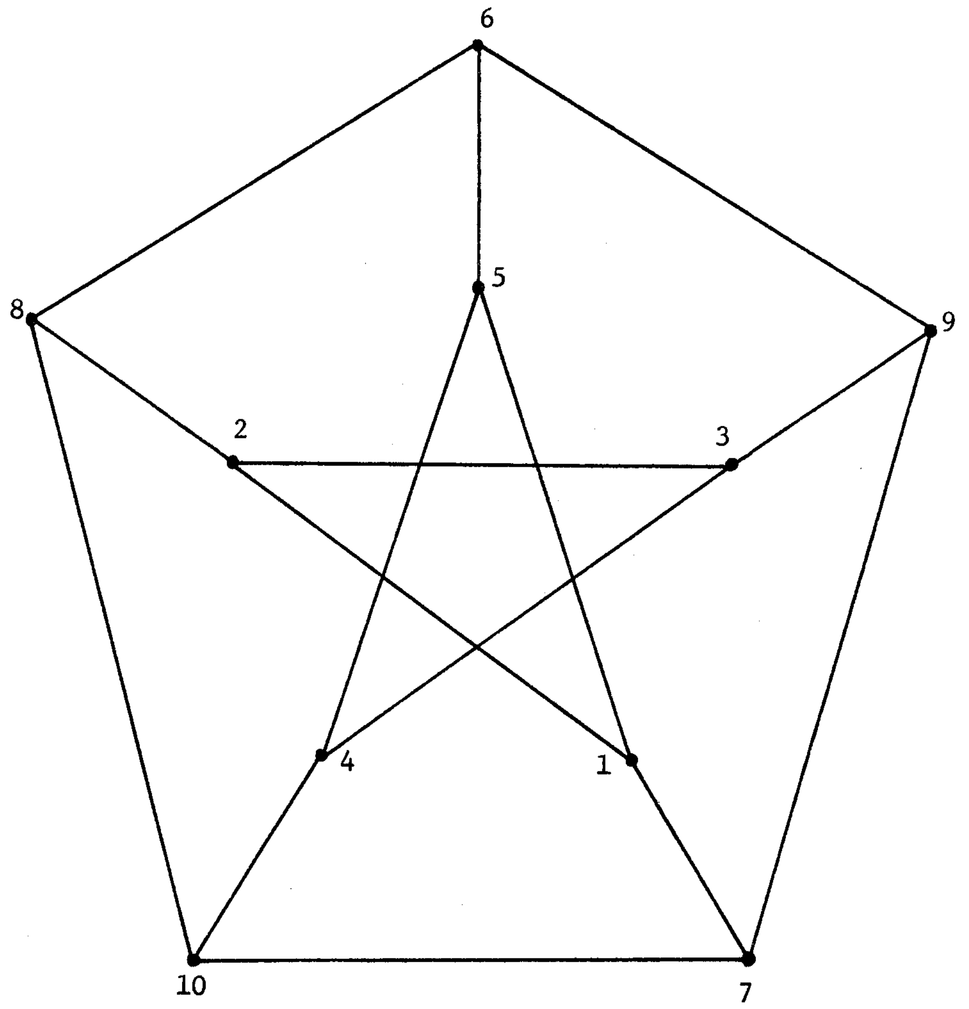
GENERATED REPRESENTATION



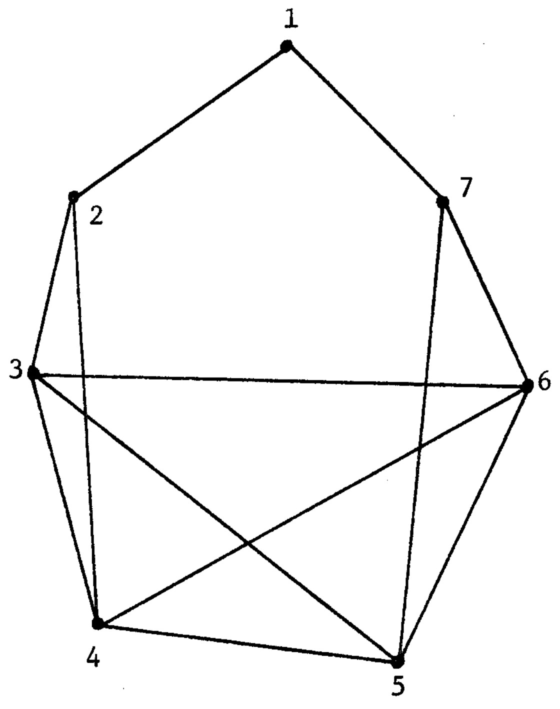
ANOTHER REPRESENTATION



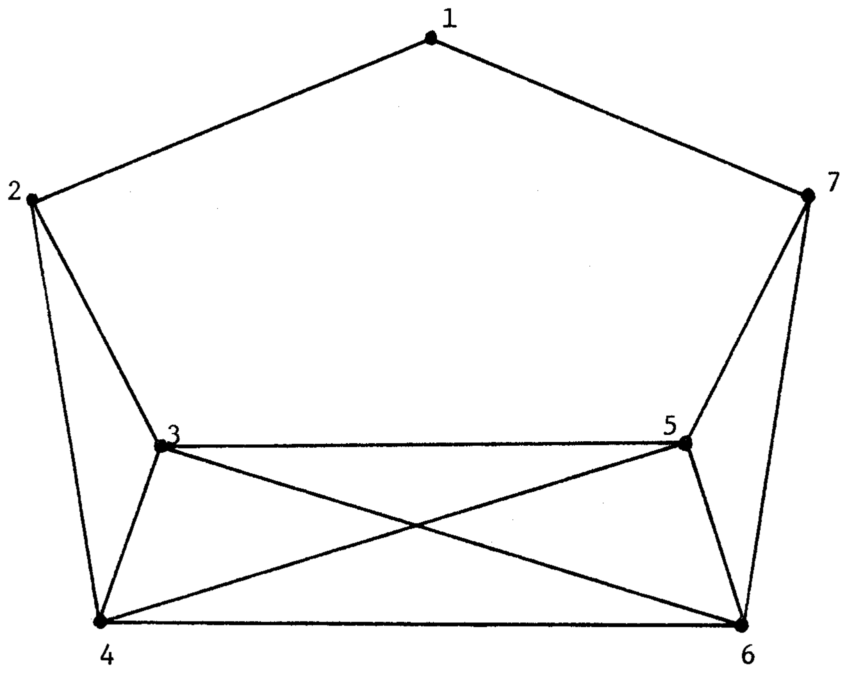
ORIGINAL PICTURE



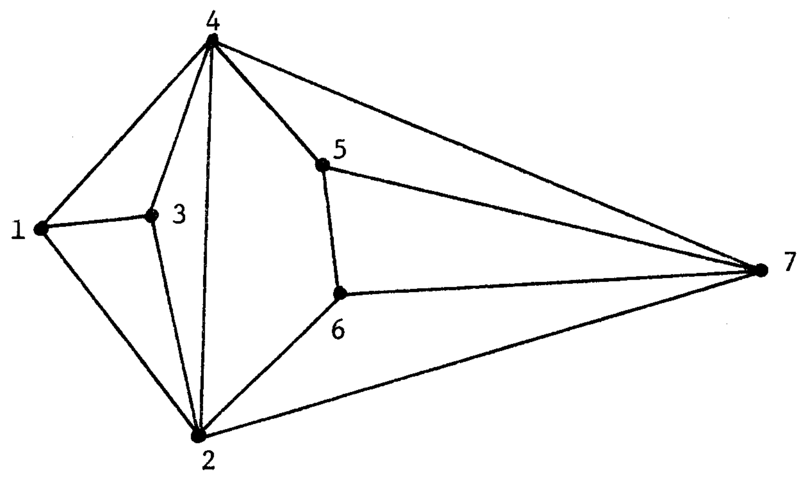
GENERATED REPRESENTATION



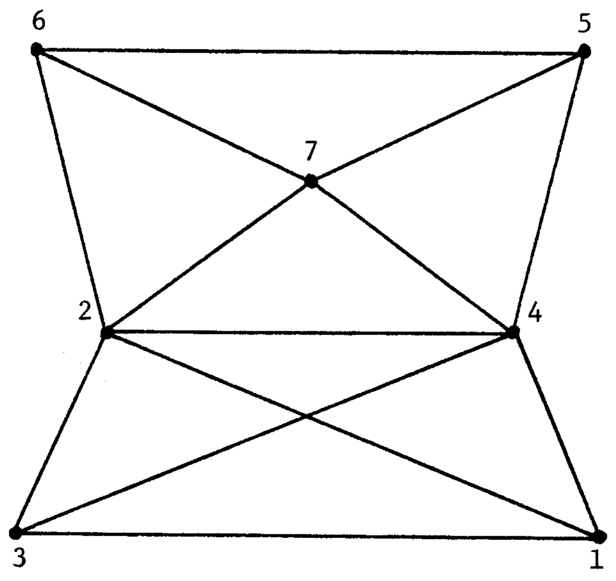
ORIGINAL PICTURE



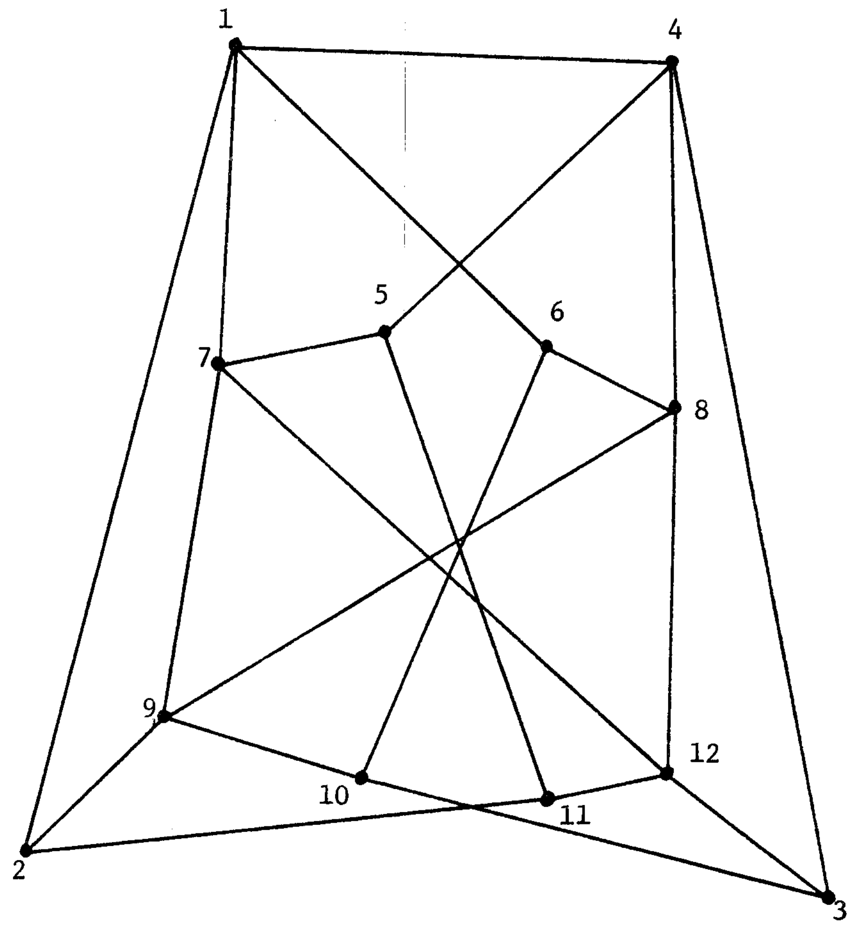
GENERATED REPRESENTATION



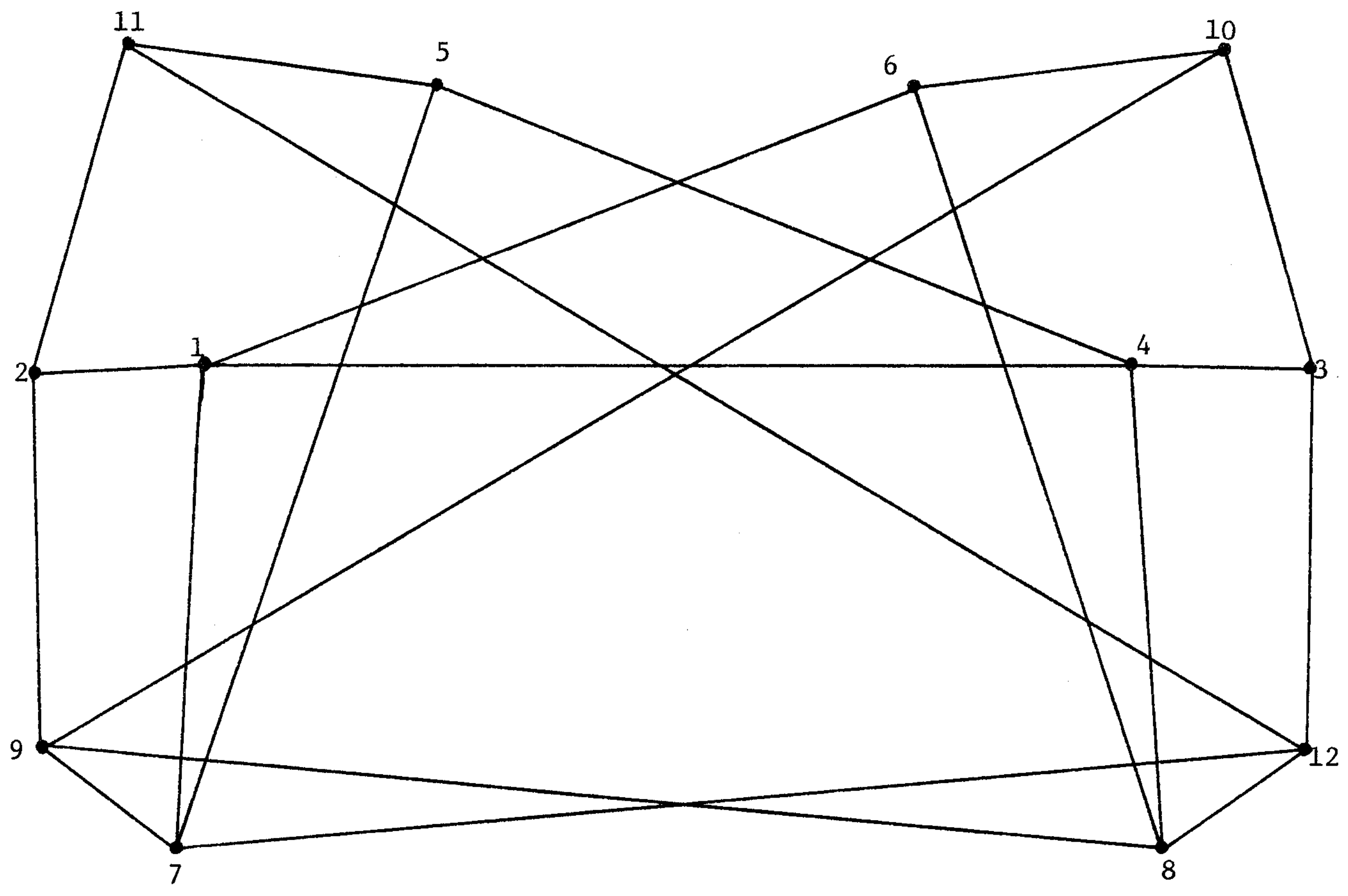
ORIGINAL PICTURE



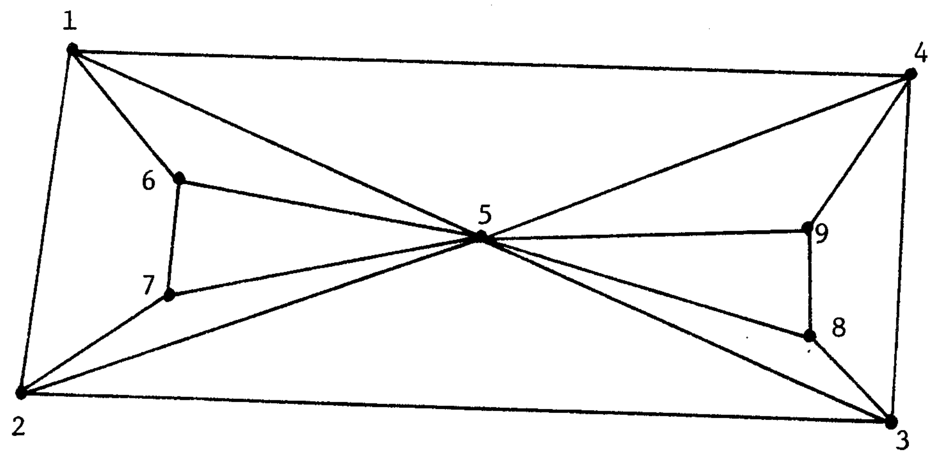
GENERATED REPRESENTATION



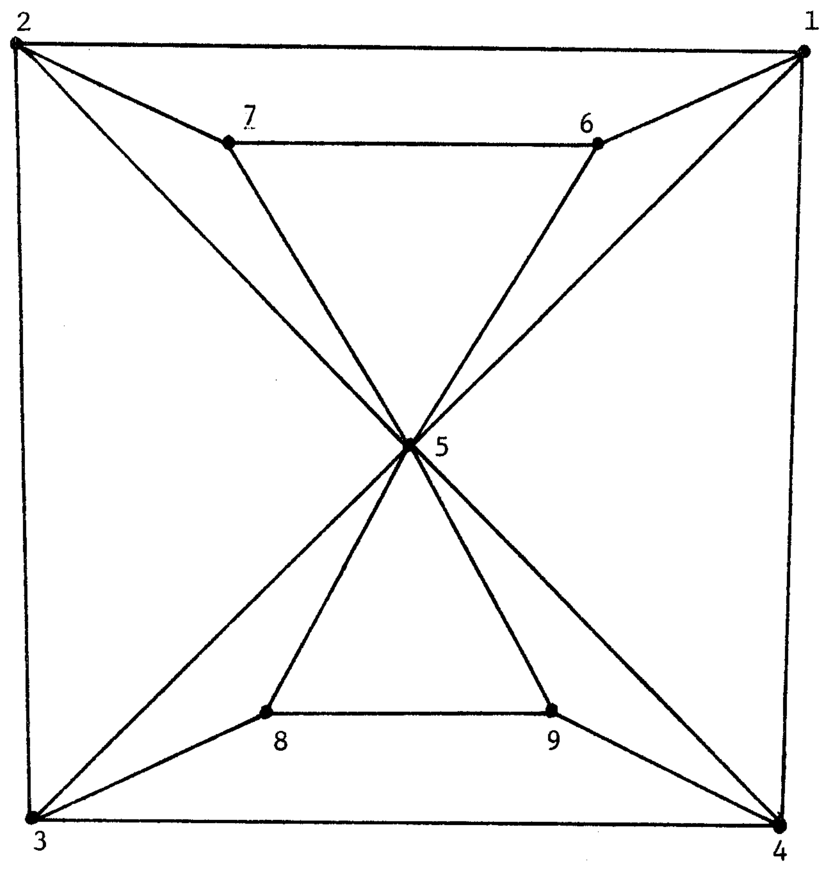
ORIGINAL PICTURE



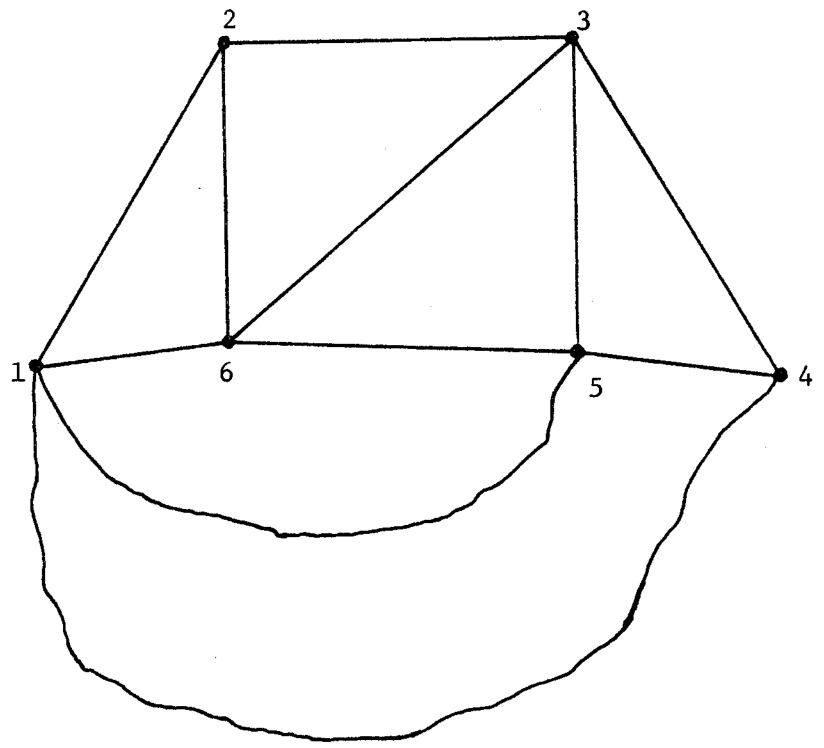
GENERATED REPRESENTATION



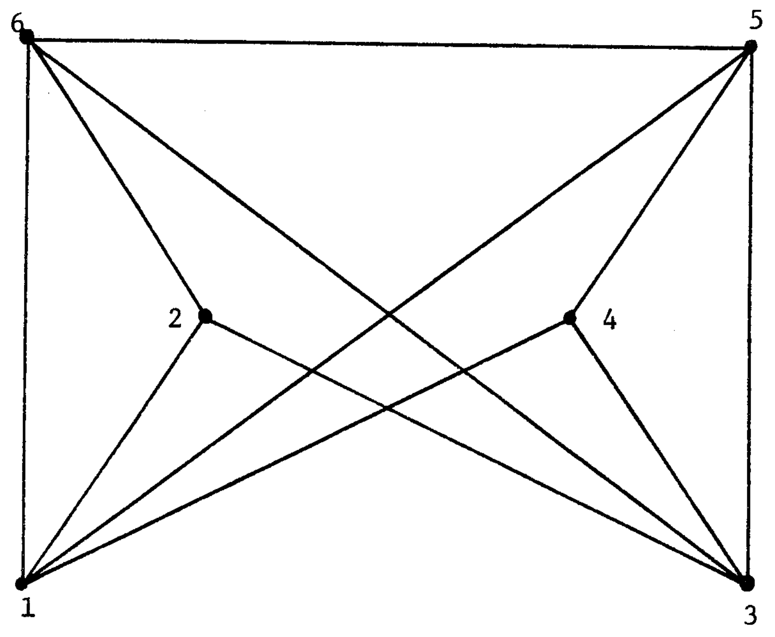
ORIGINAL PICTURE



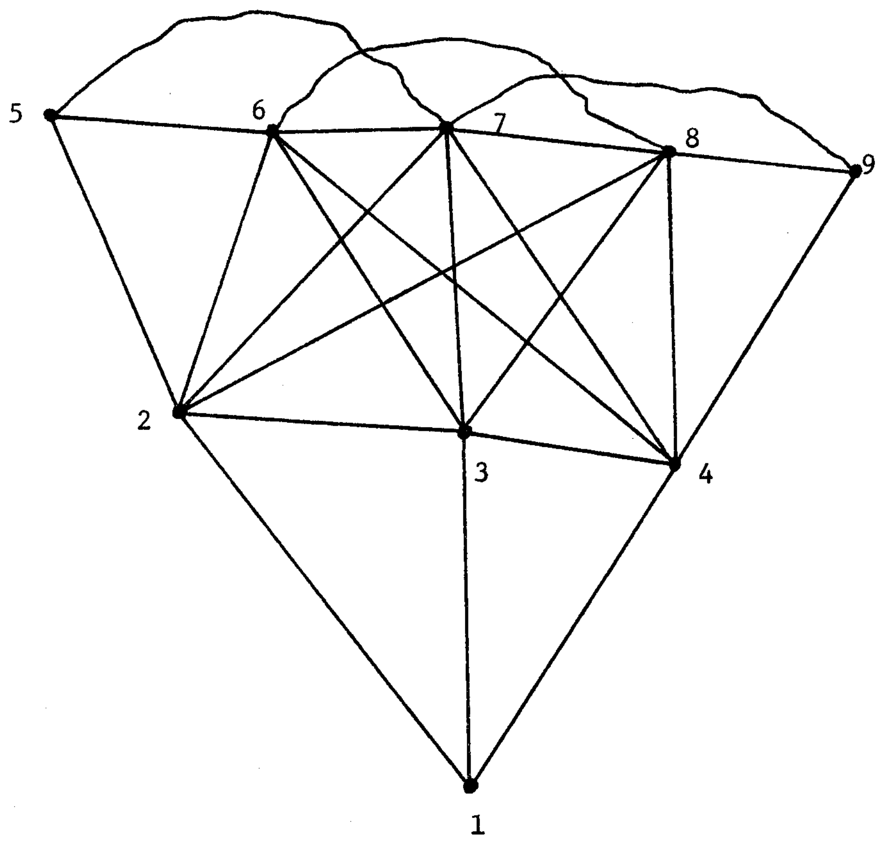
GENERATED REPRESENTATION



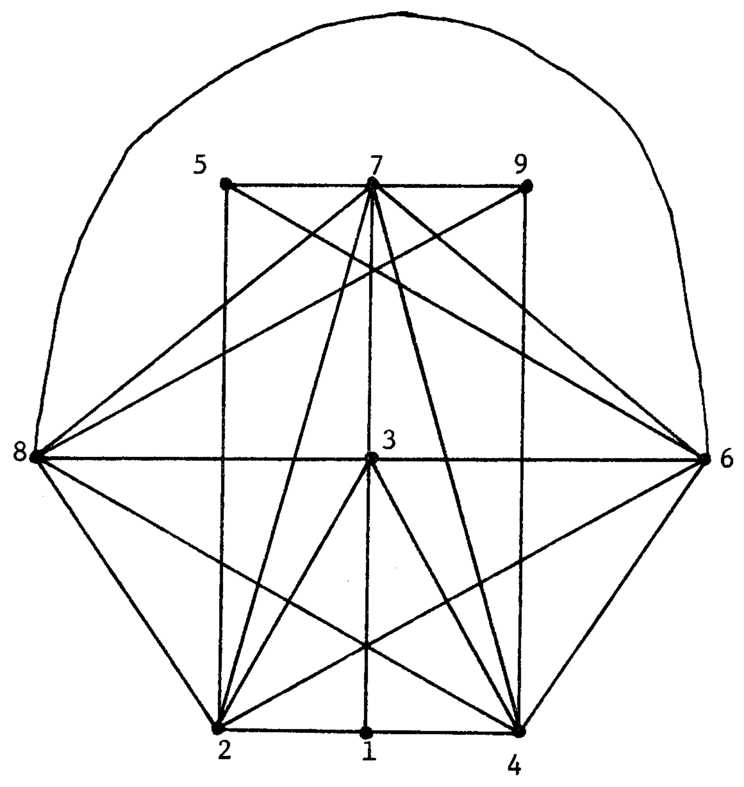
ORIGINAL PICTURE



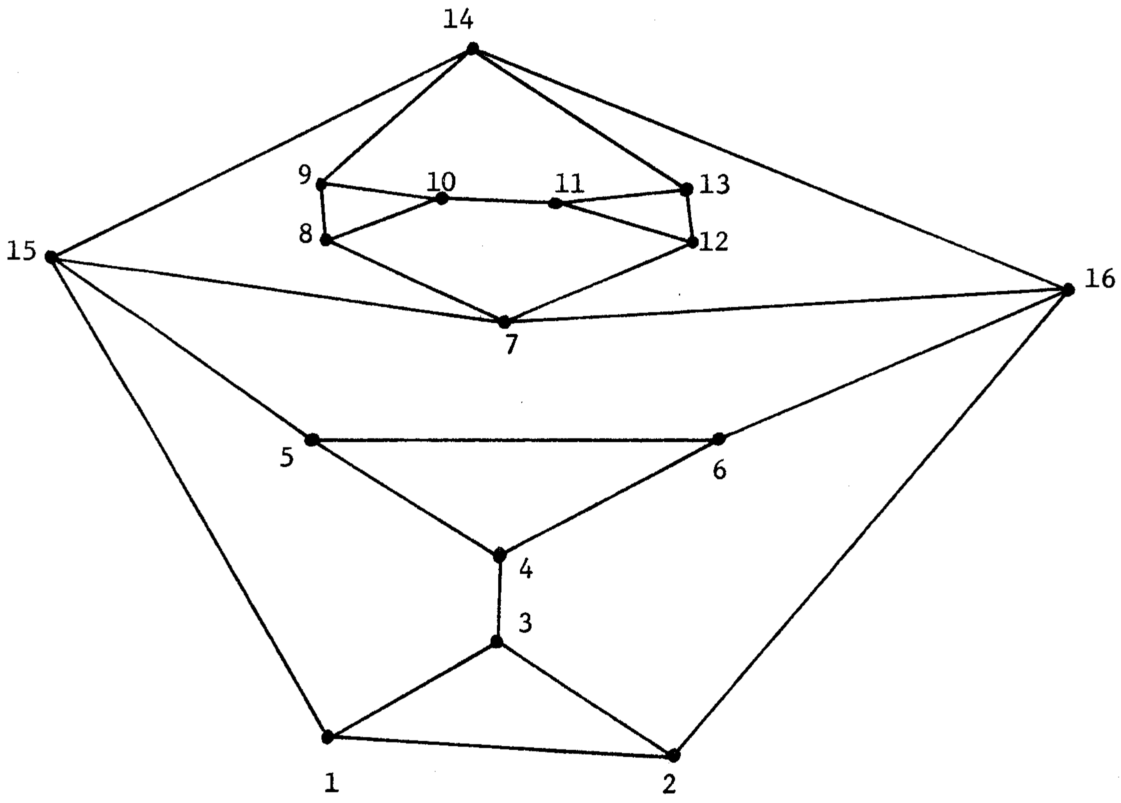
GENERATED REPRESENTATION



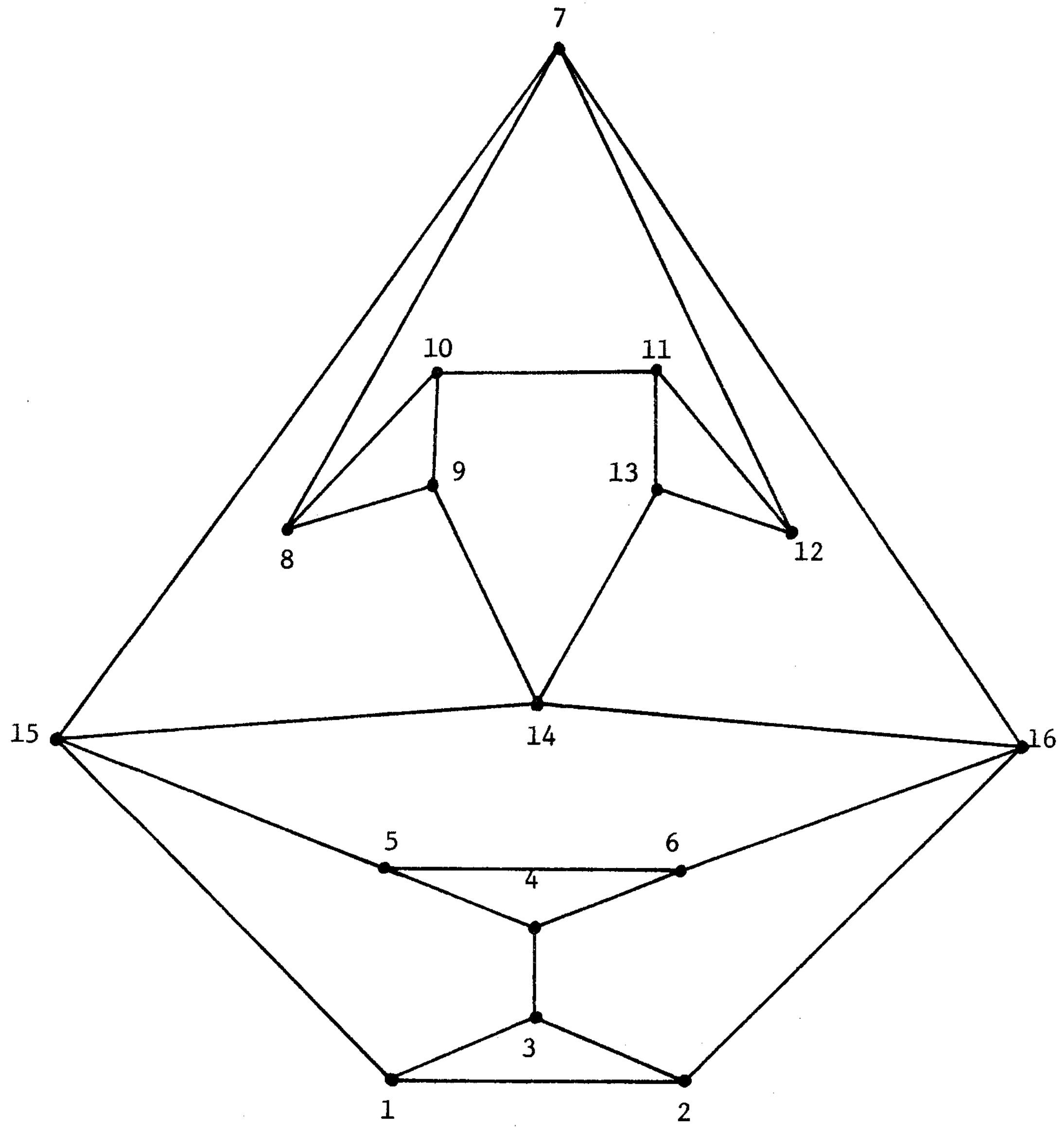
ORIGINAL PICTURE



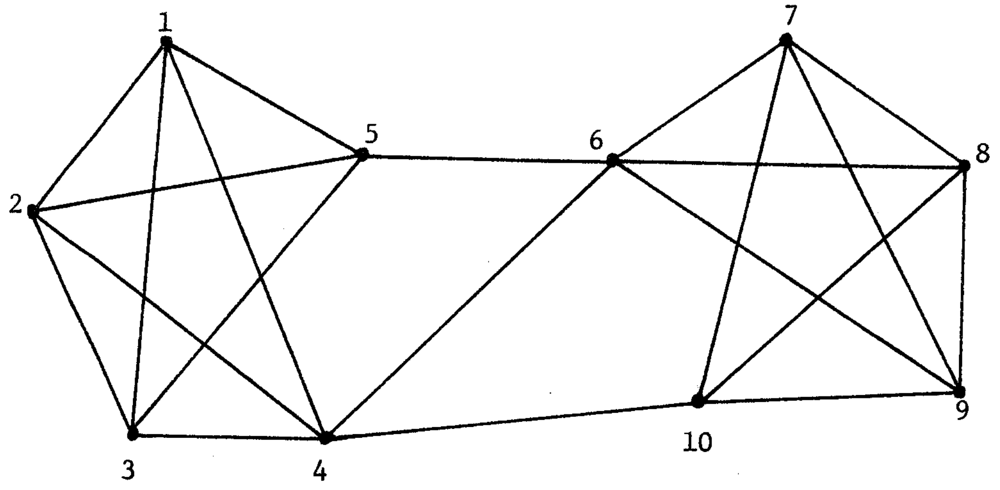
GENERATED REPRESENTATION



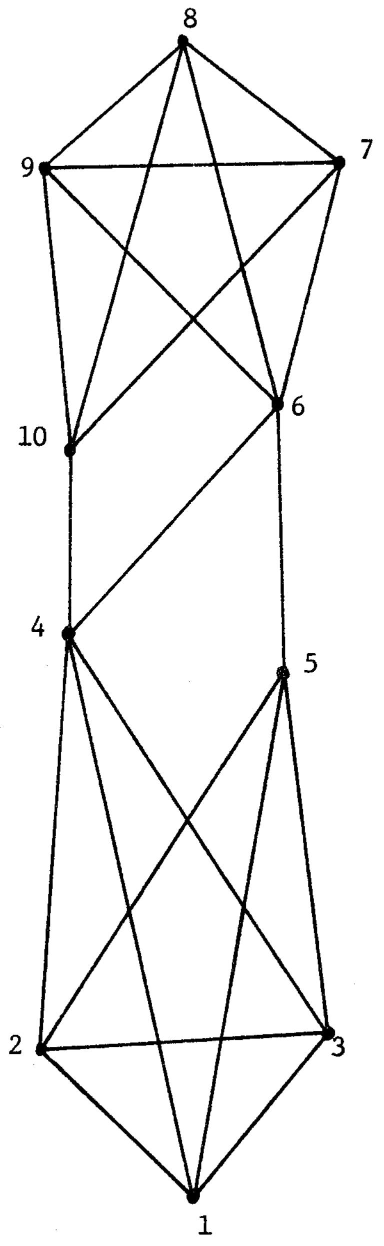
ORIGINAL PICTURE



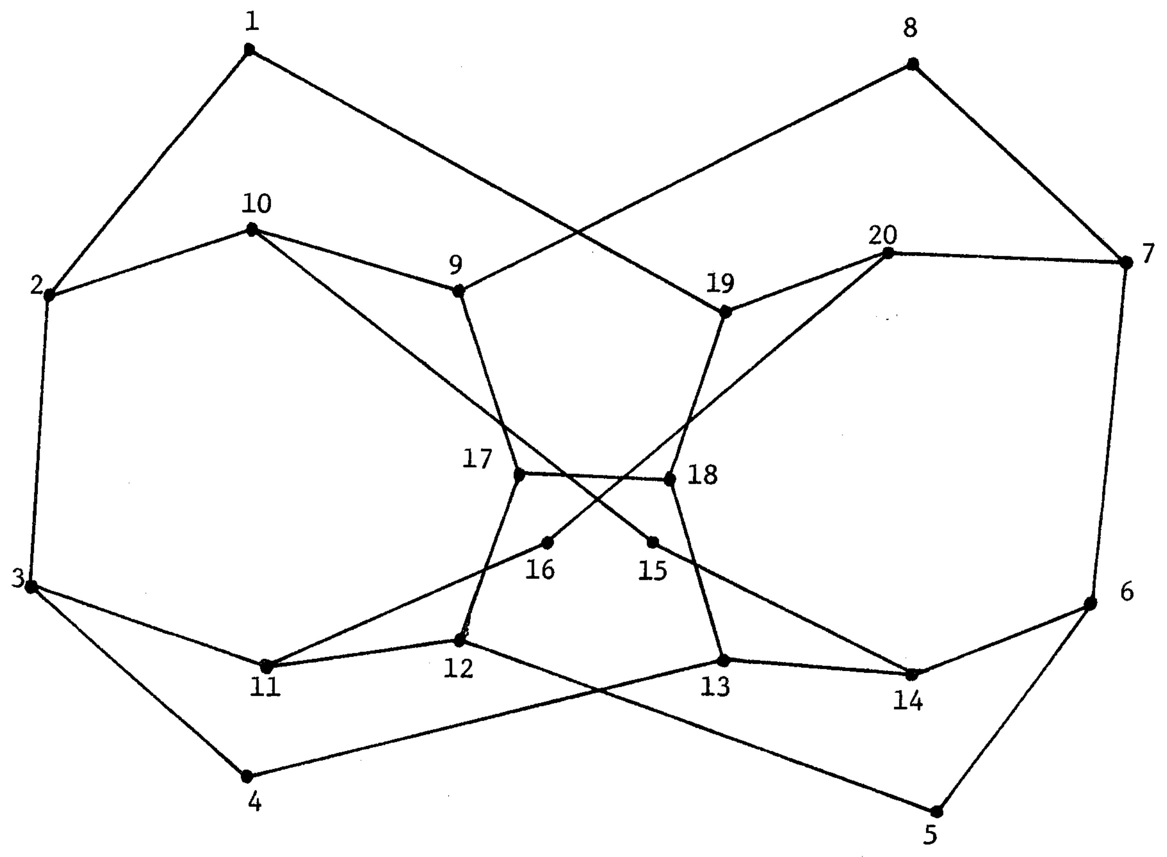
GENERATED REPRESENTATION



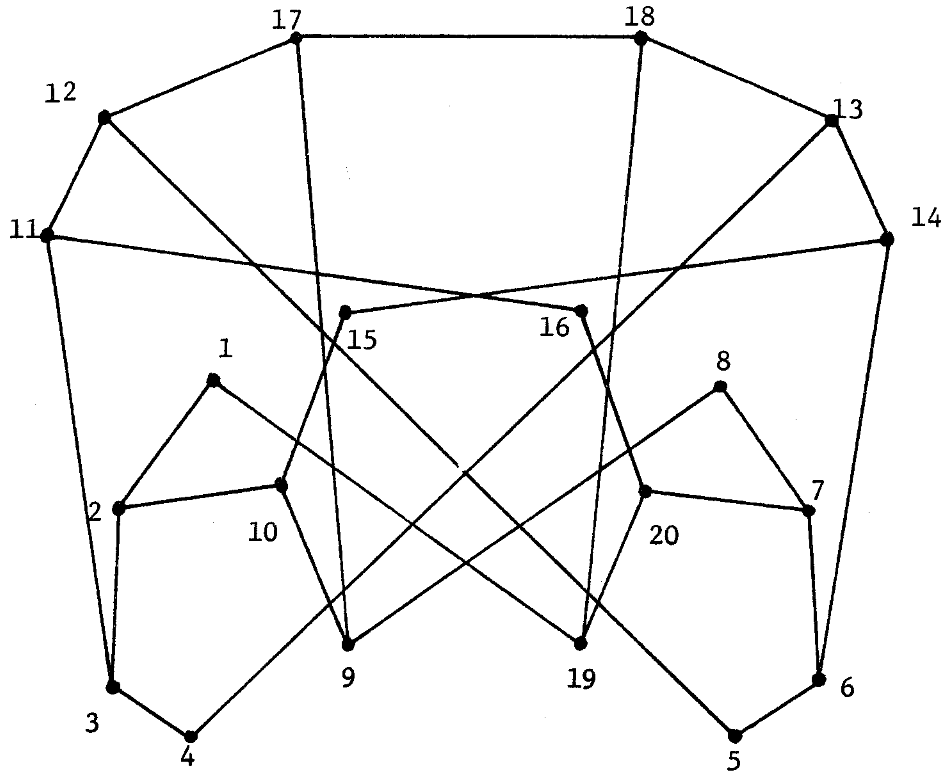
ORIGINAL PICTURE



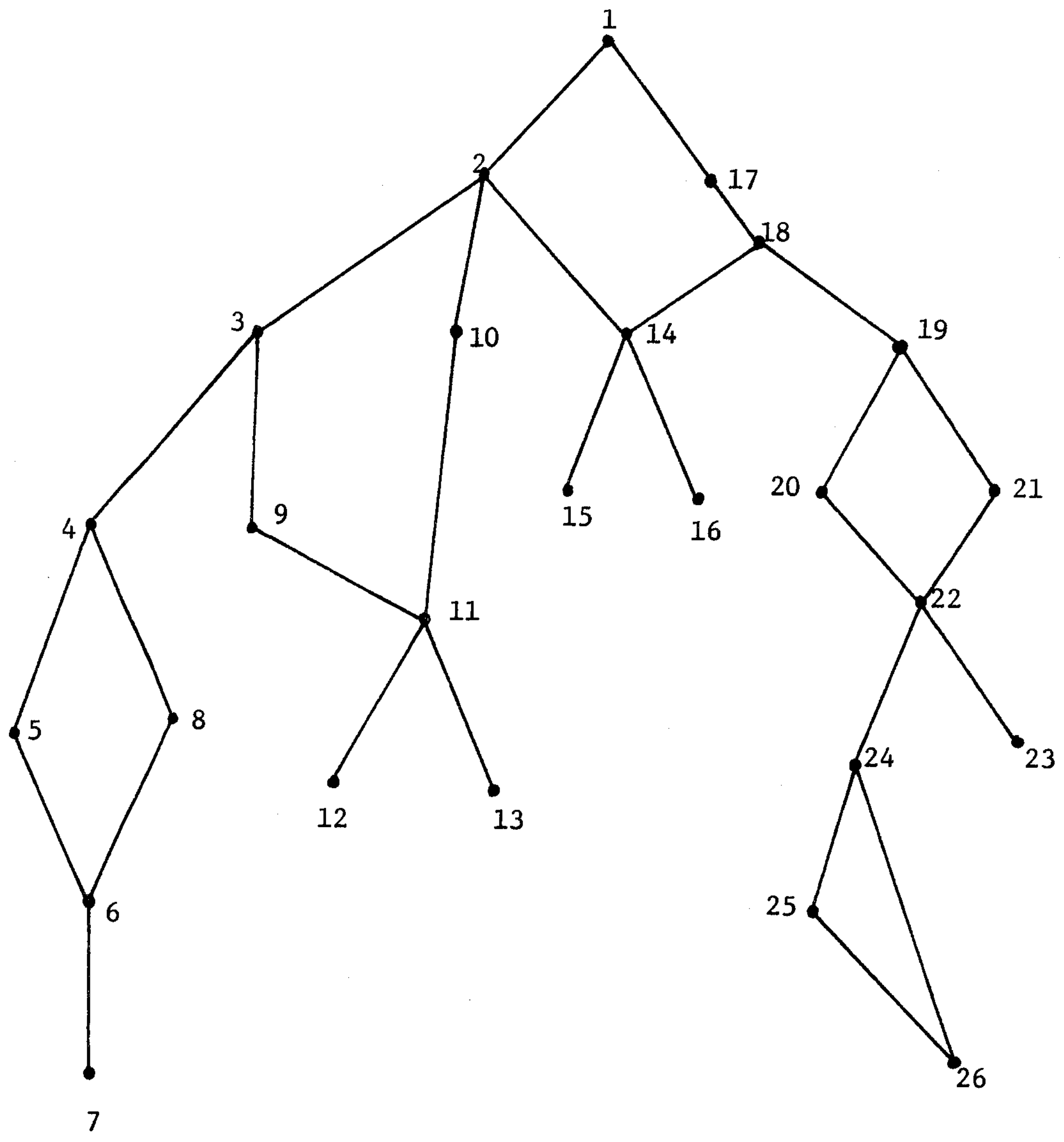
GENERATED REPRESENTATION



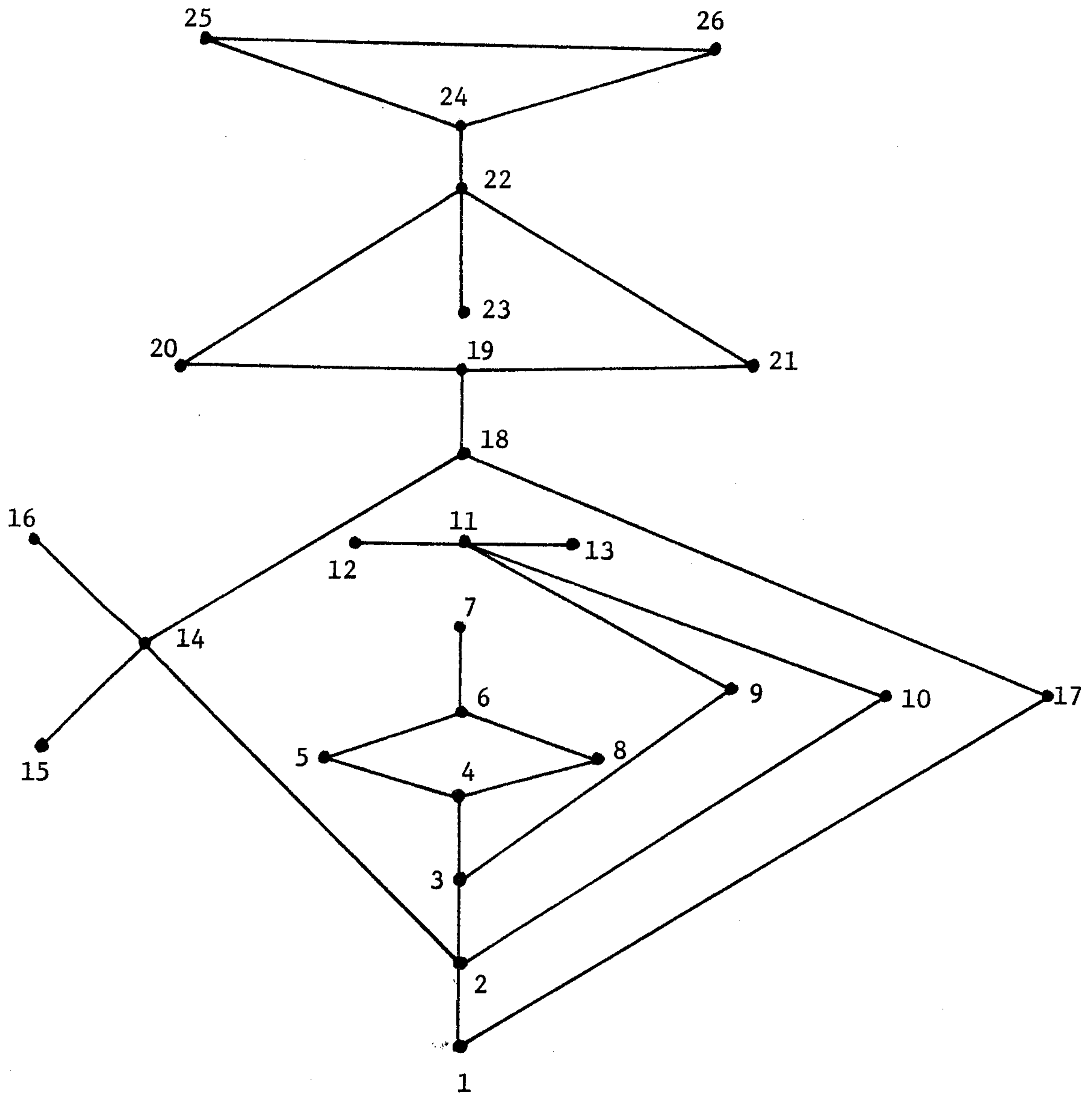
ORIGINAL PICTURE



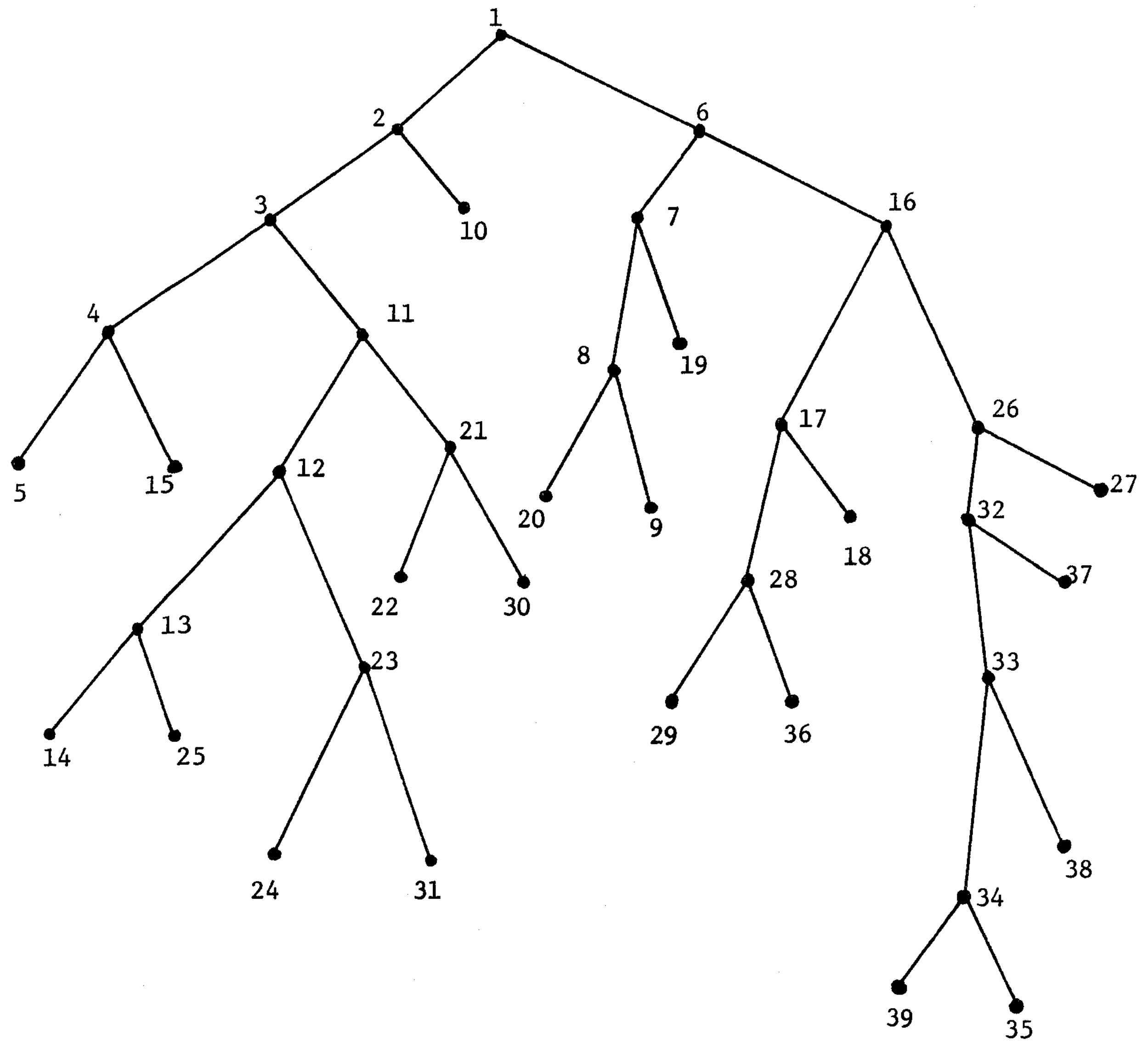
GENERATED REPRESENTATION



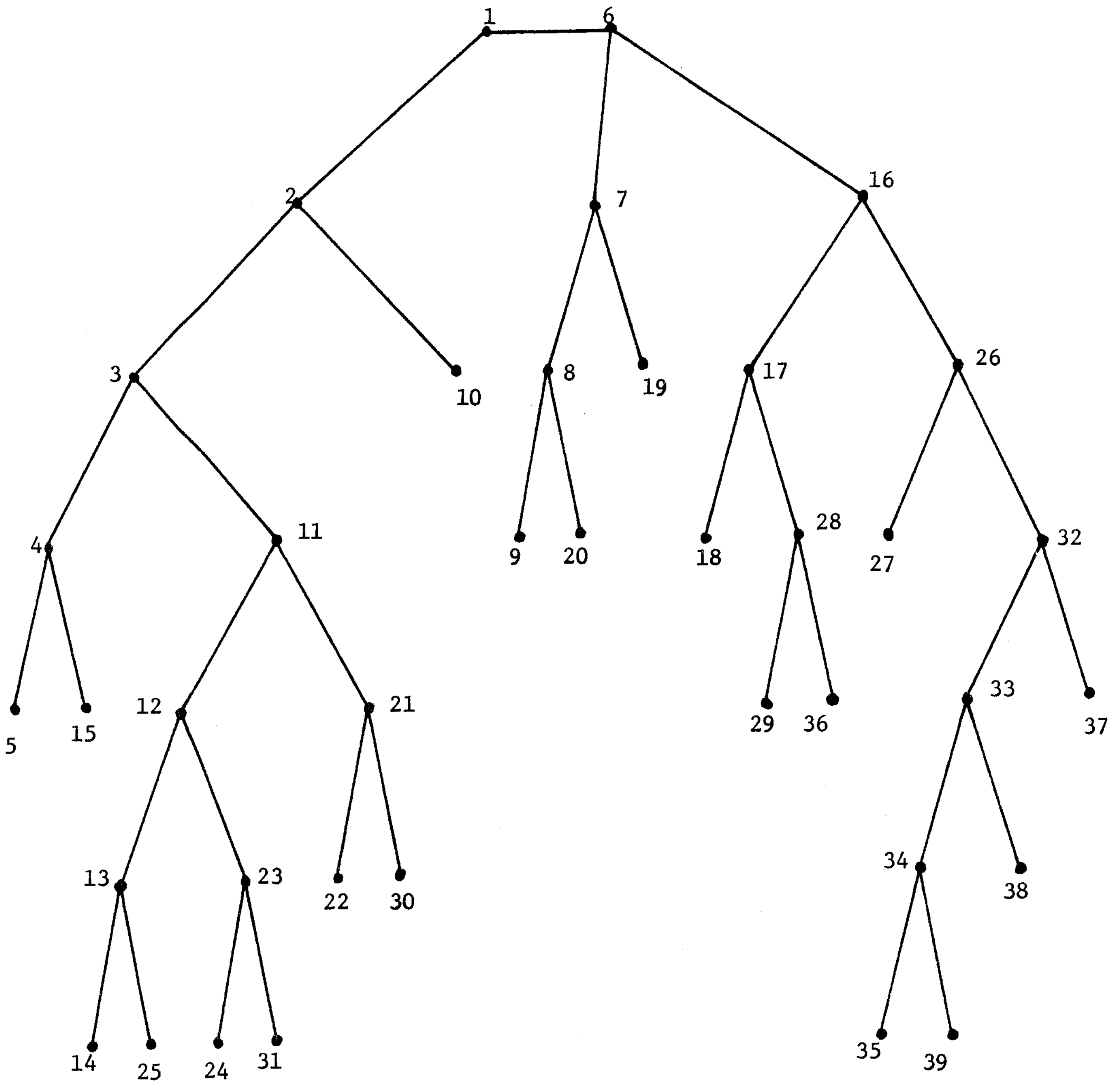
ORIGINAL PICTURE



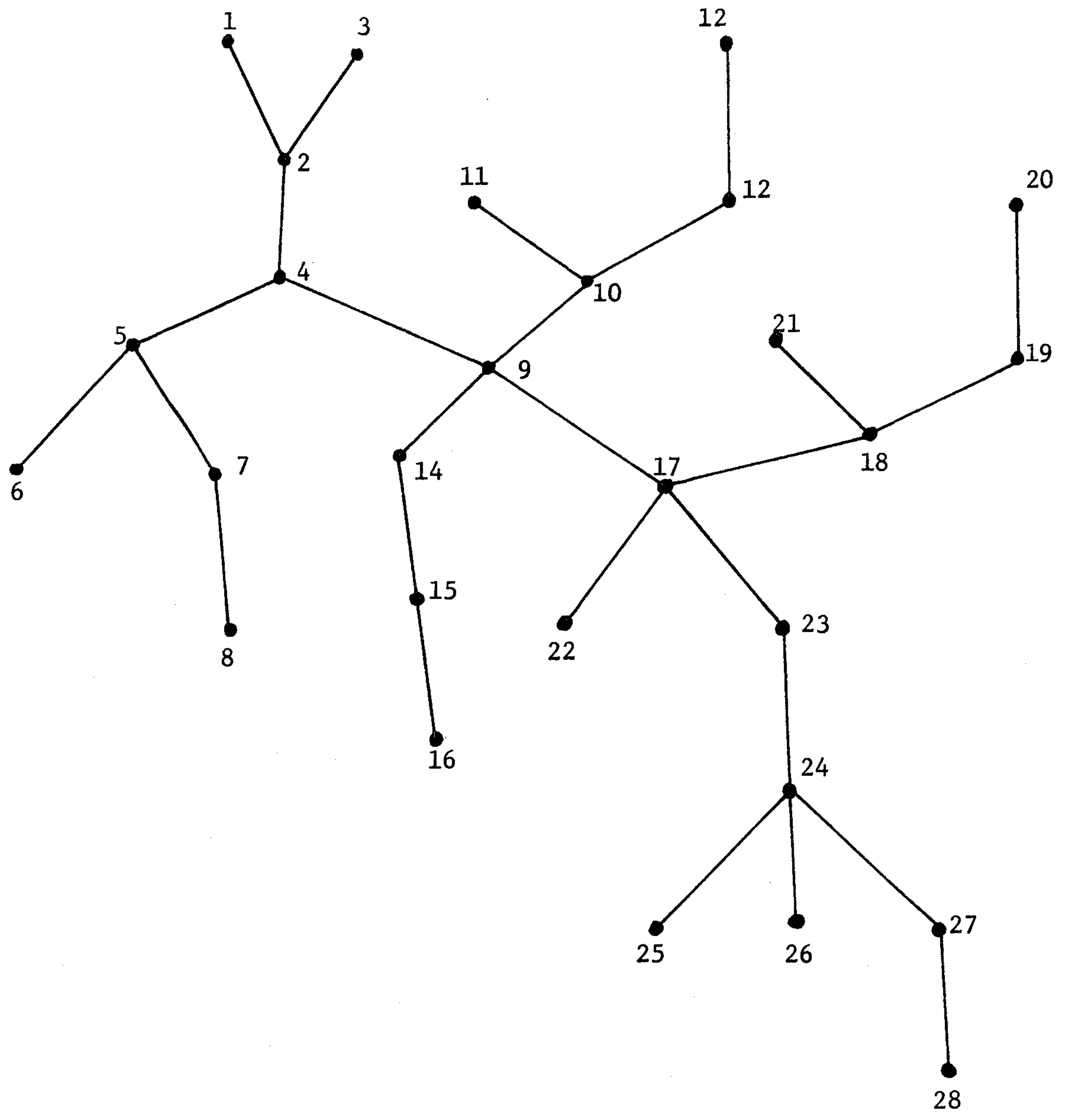
GENERATED REPRESENTATION



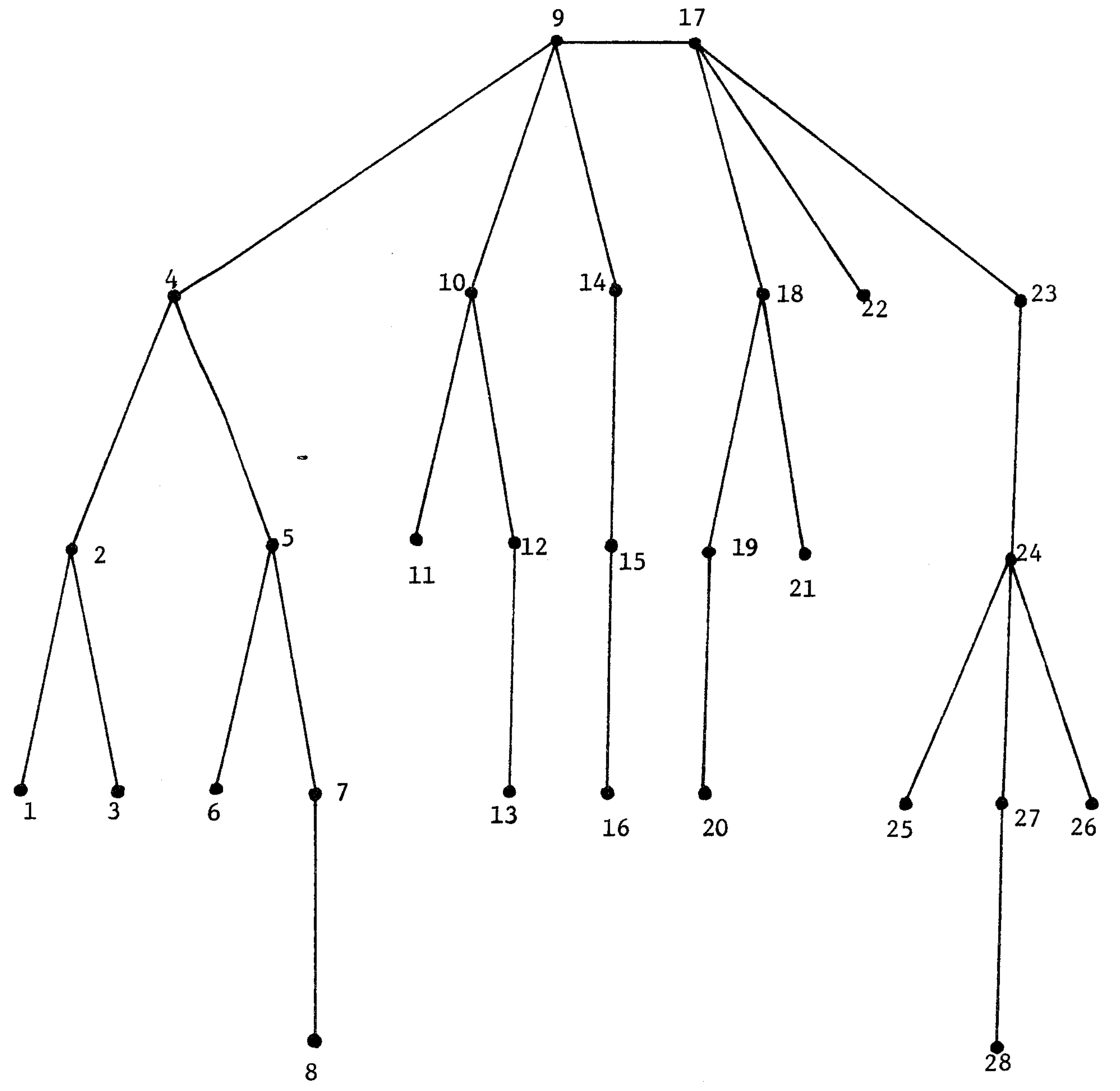
ORIGINAL PICTURE



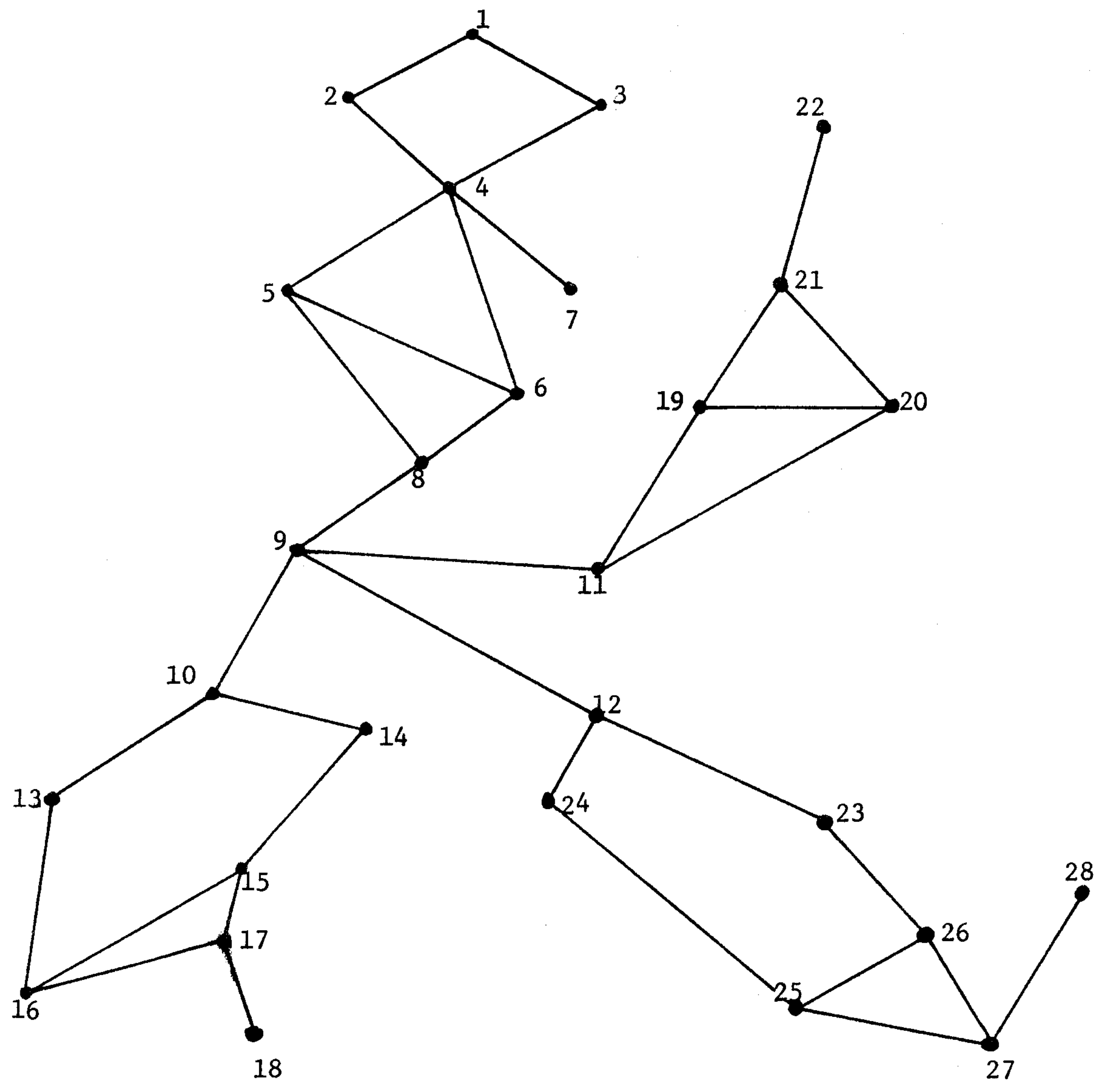
GENERATED REPRESENTATION



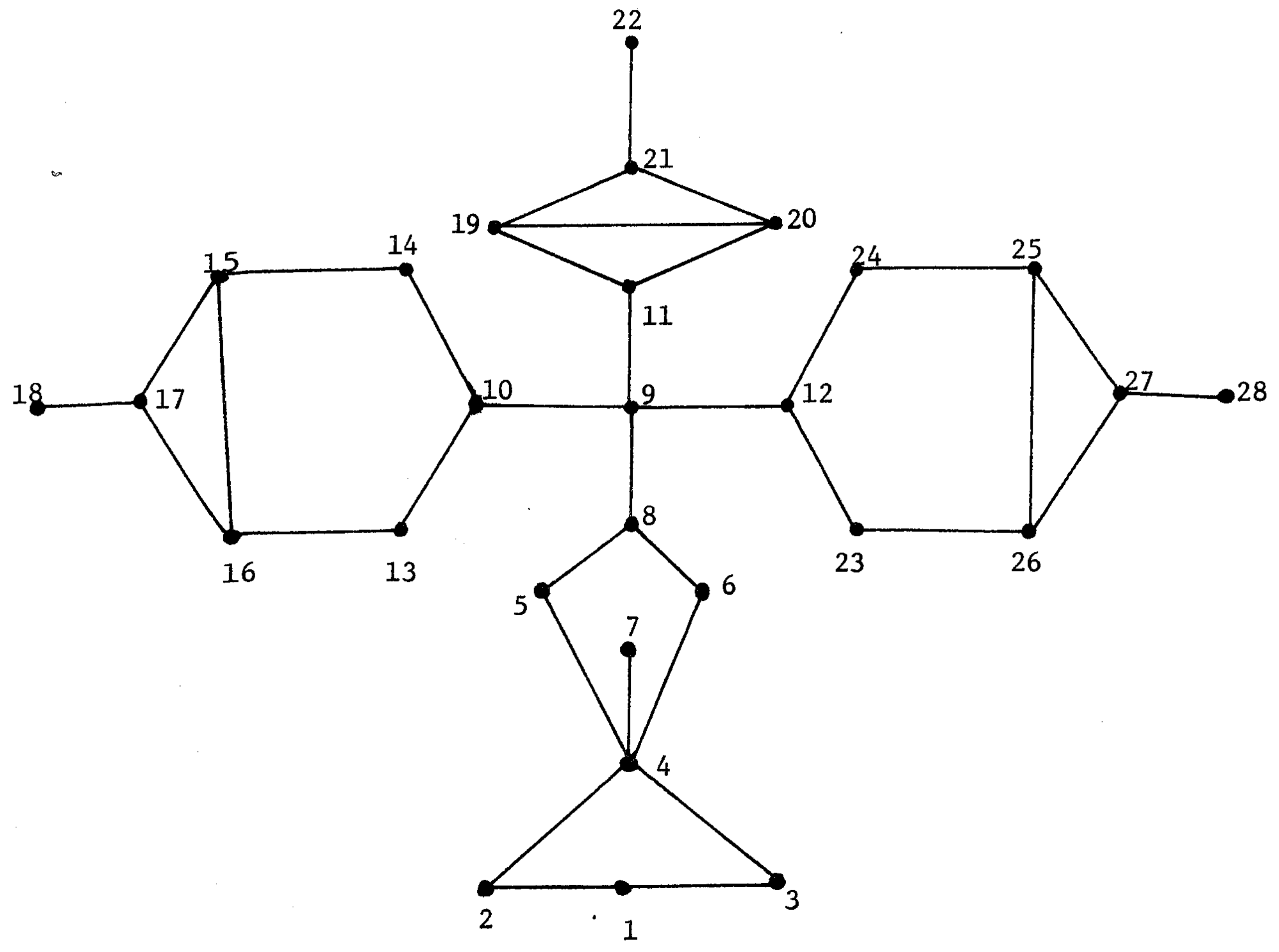
ORIGINAL PICTURE



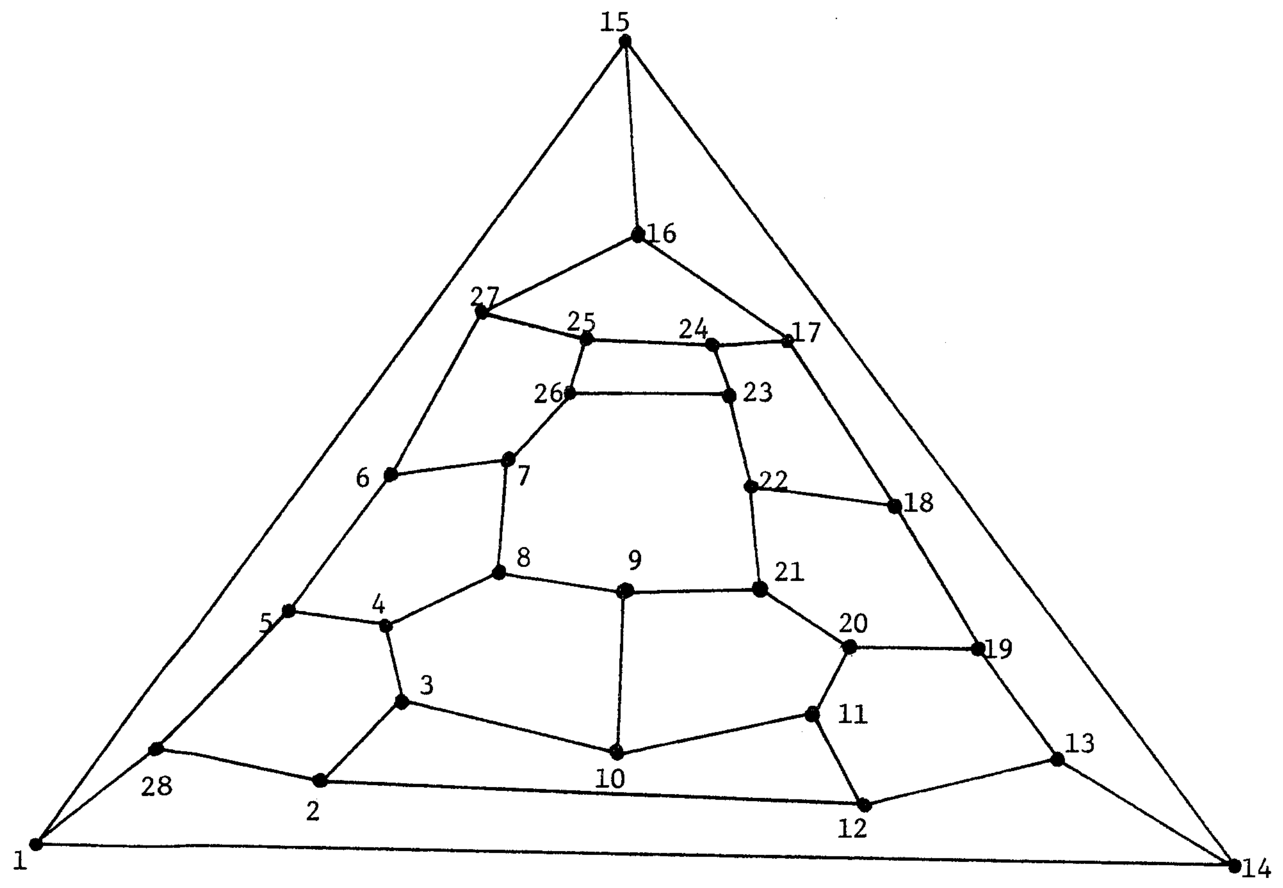
GENERATED REPRESENTATION



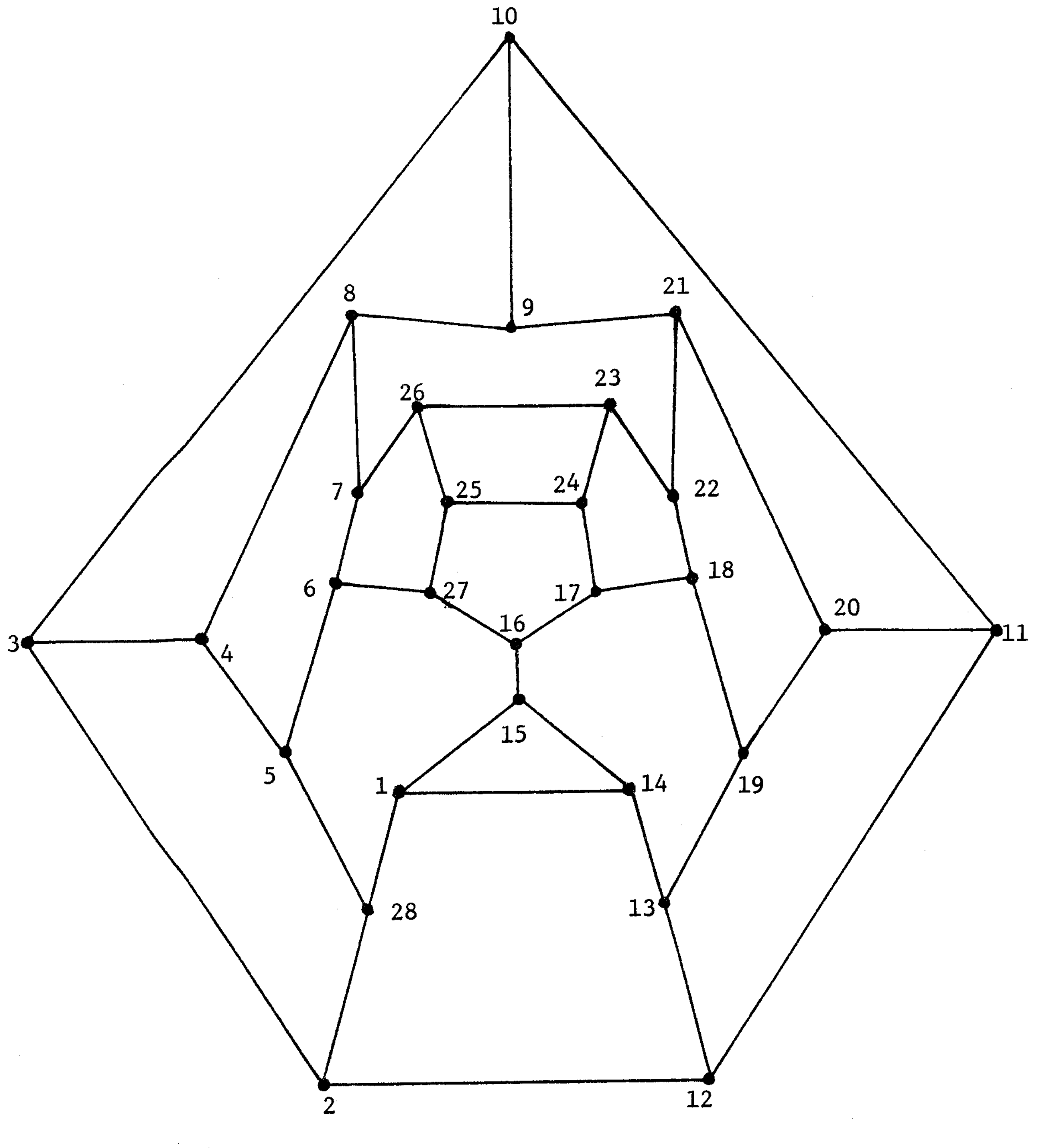
ORIGINAL PICTURE



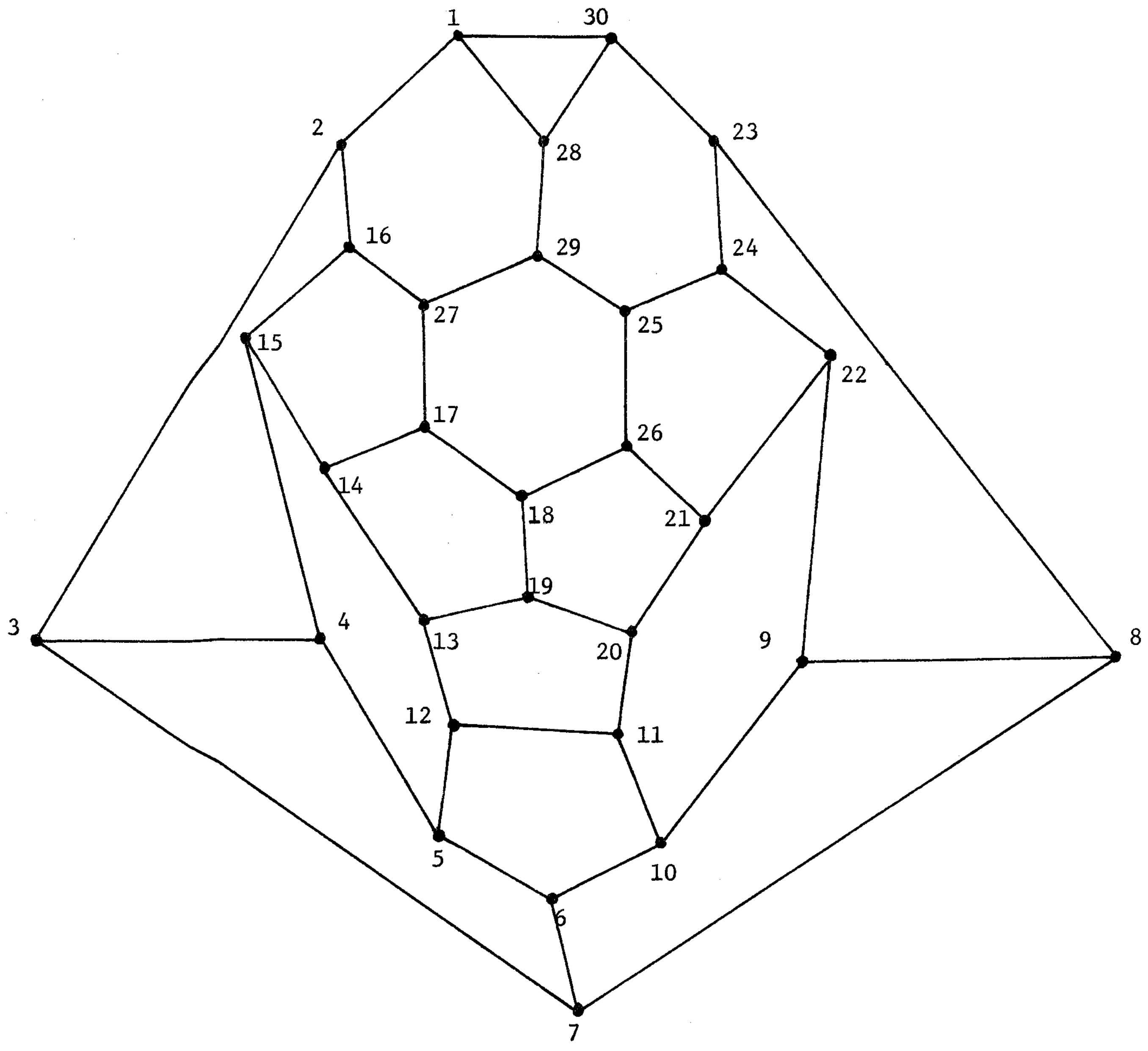
GENERATED REPRESENTATION



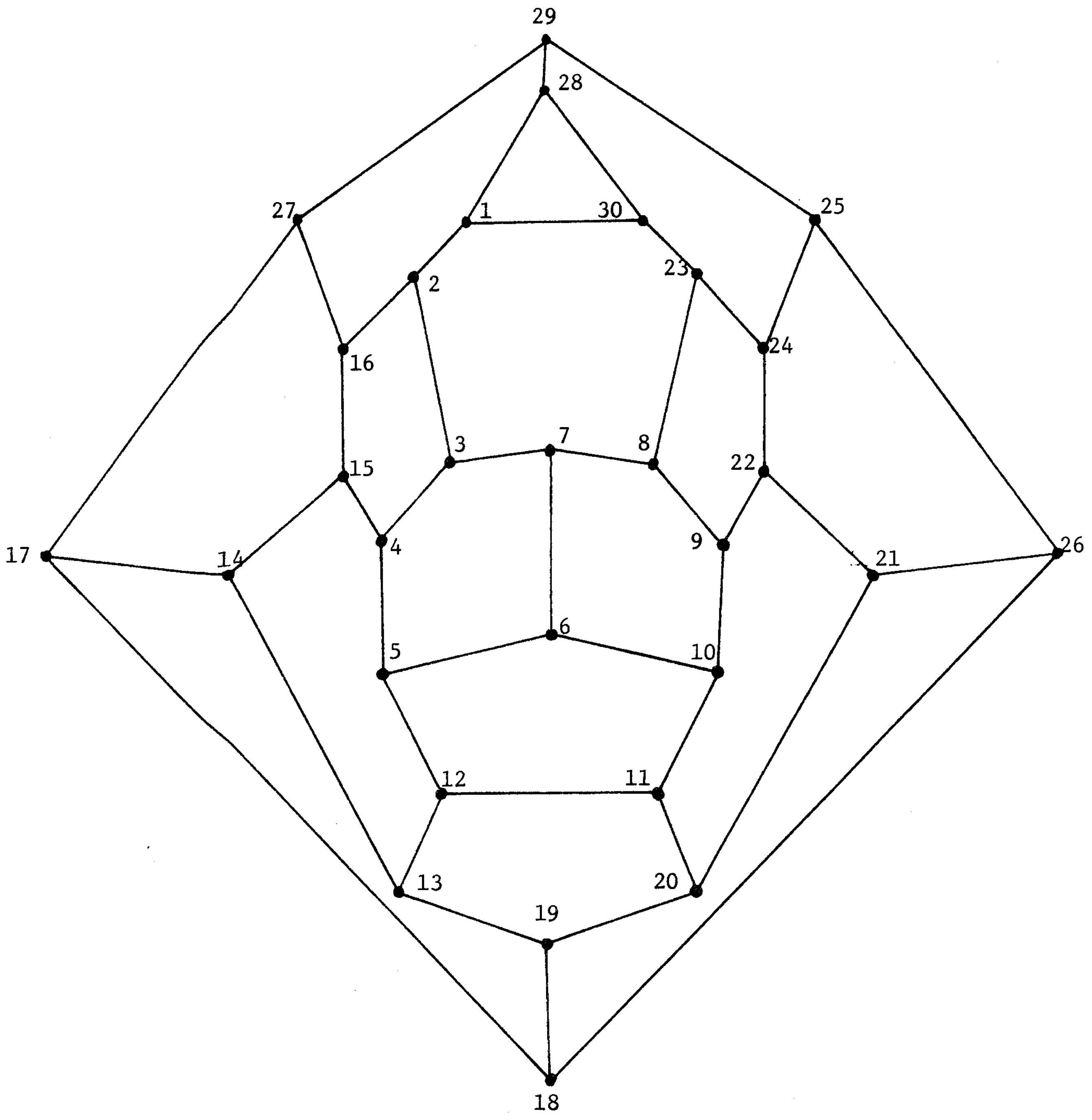
ORIGINAL PICTURE



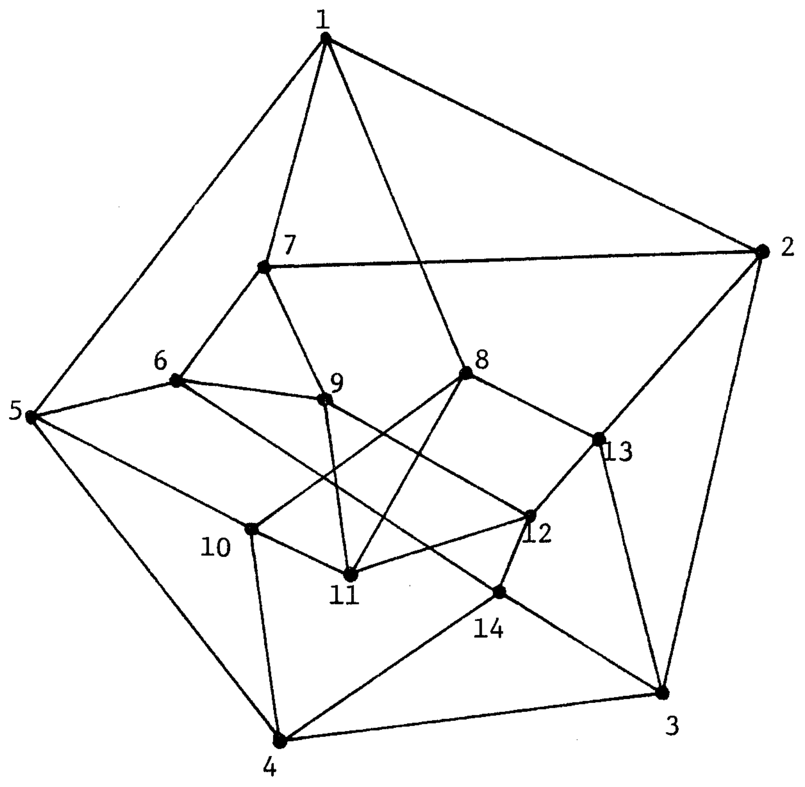
GENERATED REPRESENTATION



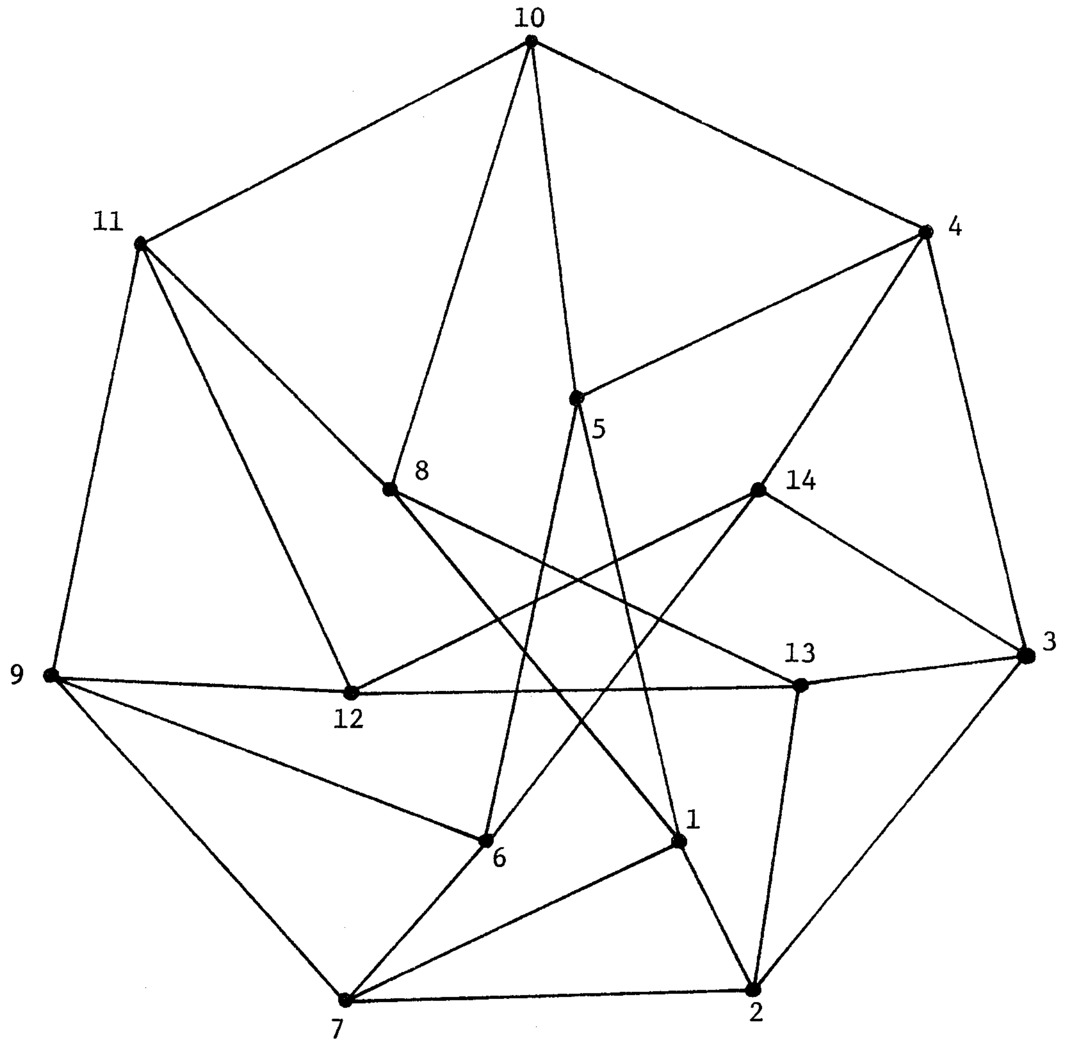
ORIGINAL PICTURE



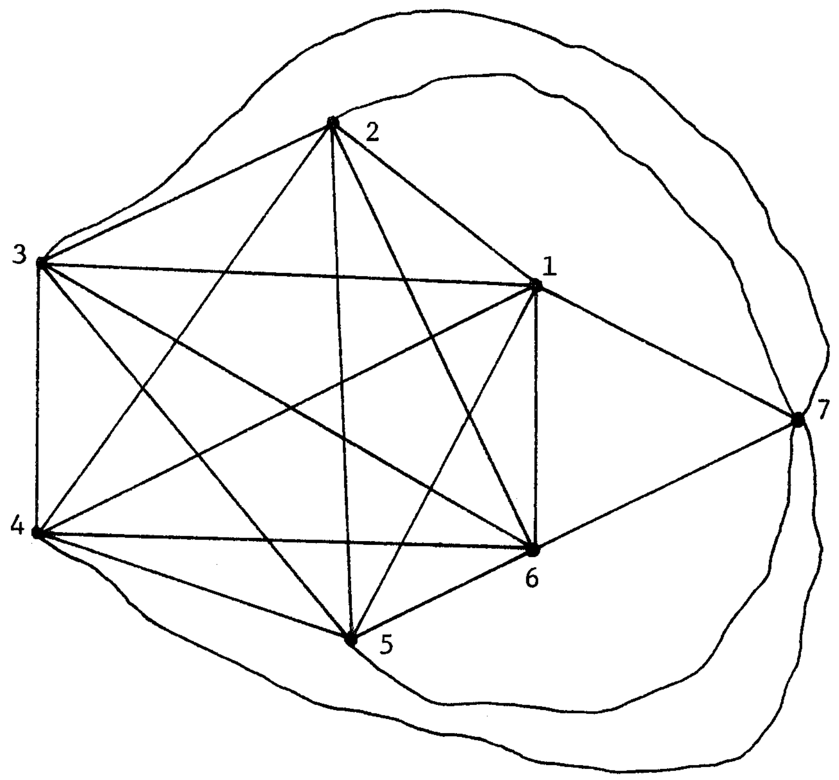
GENERATED REPRESENTATION



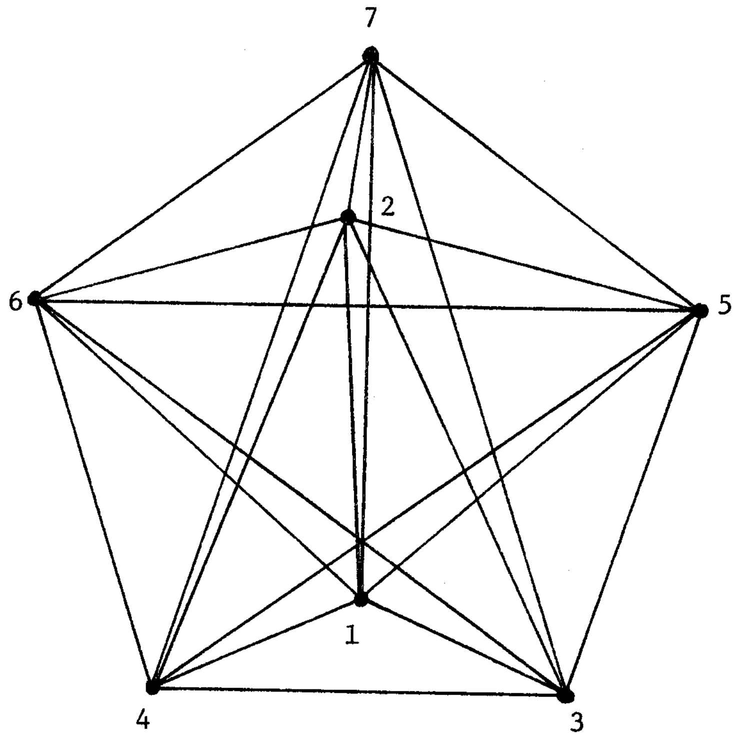
ORIGINAL PICTURE



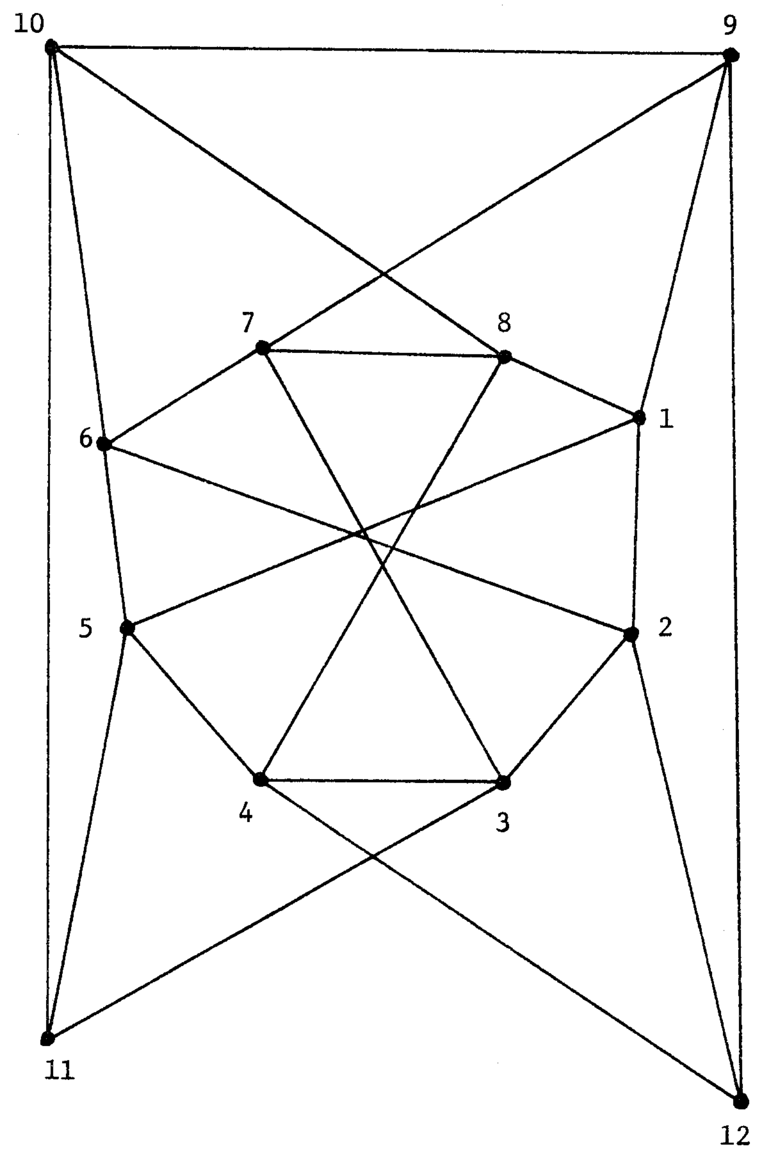
GENERATED REPRESENTATION



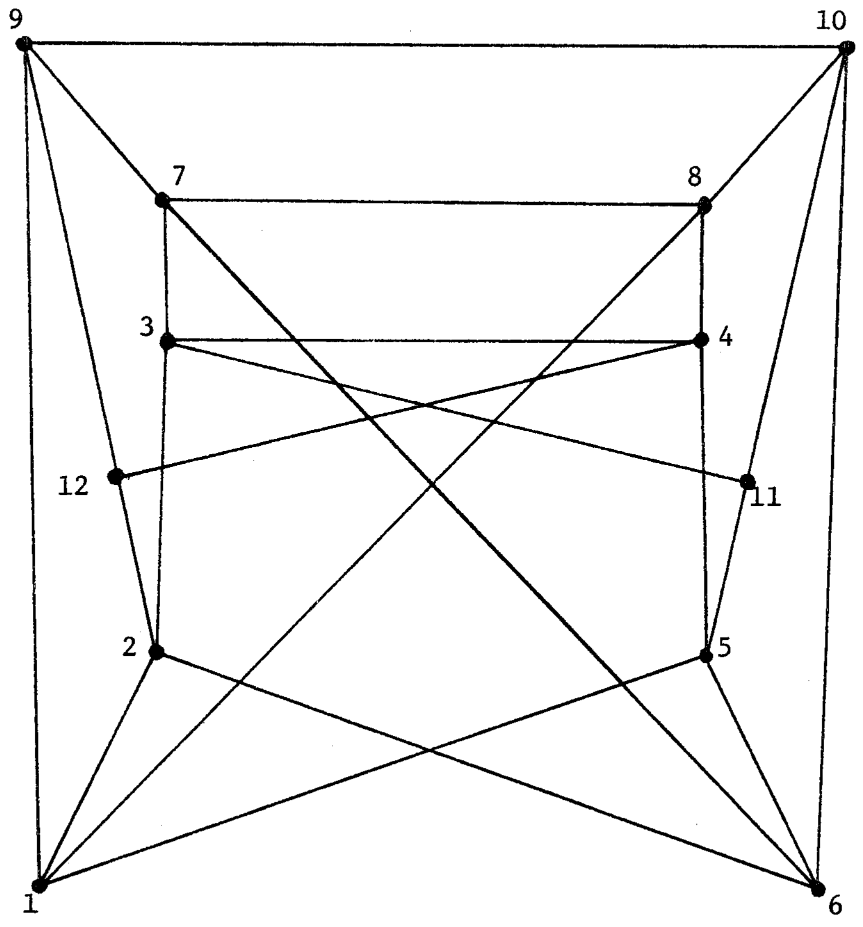
ORIGINAL PICTURE



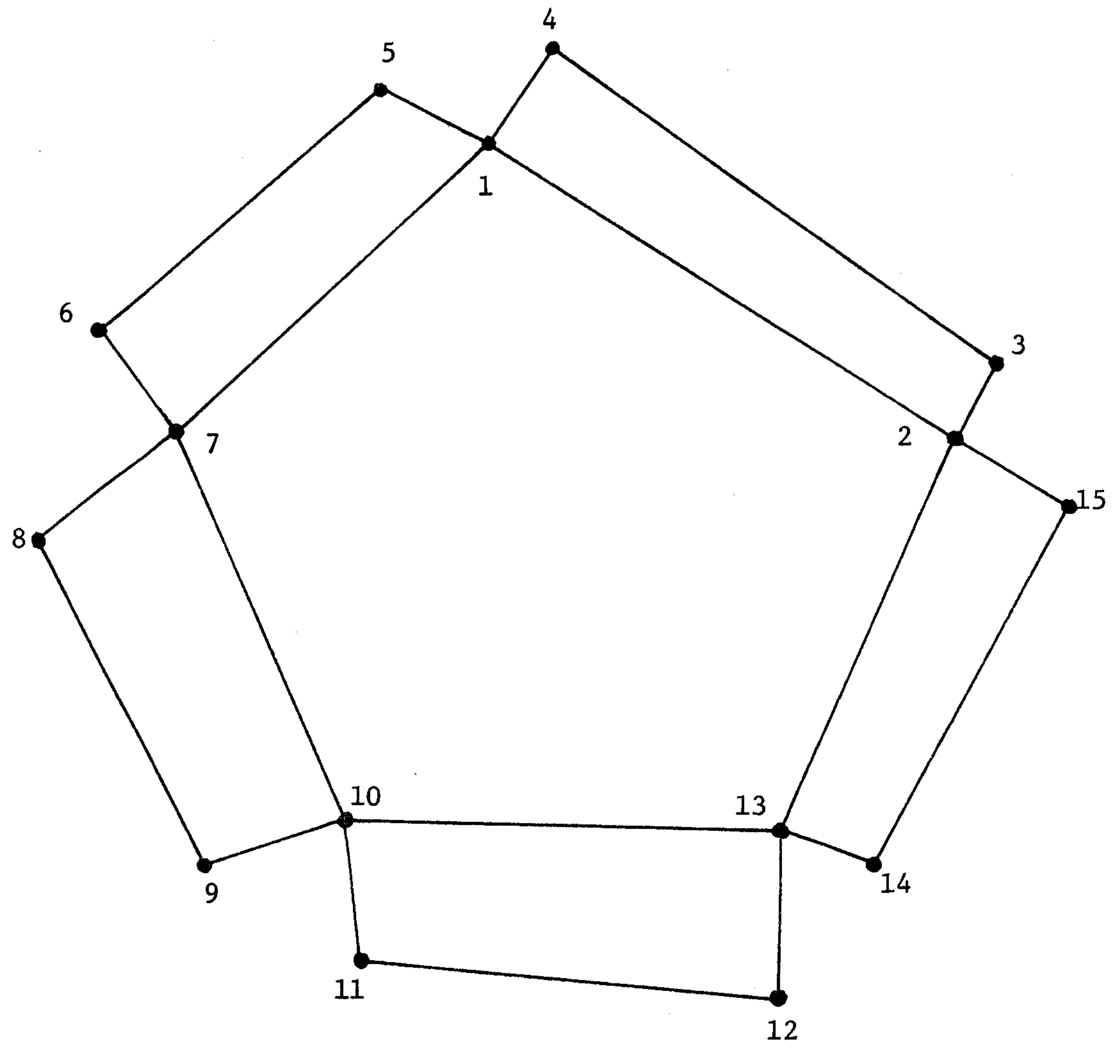
GENERATED REPRESENTATION



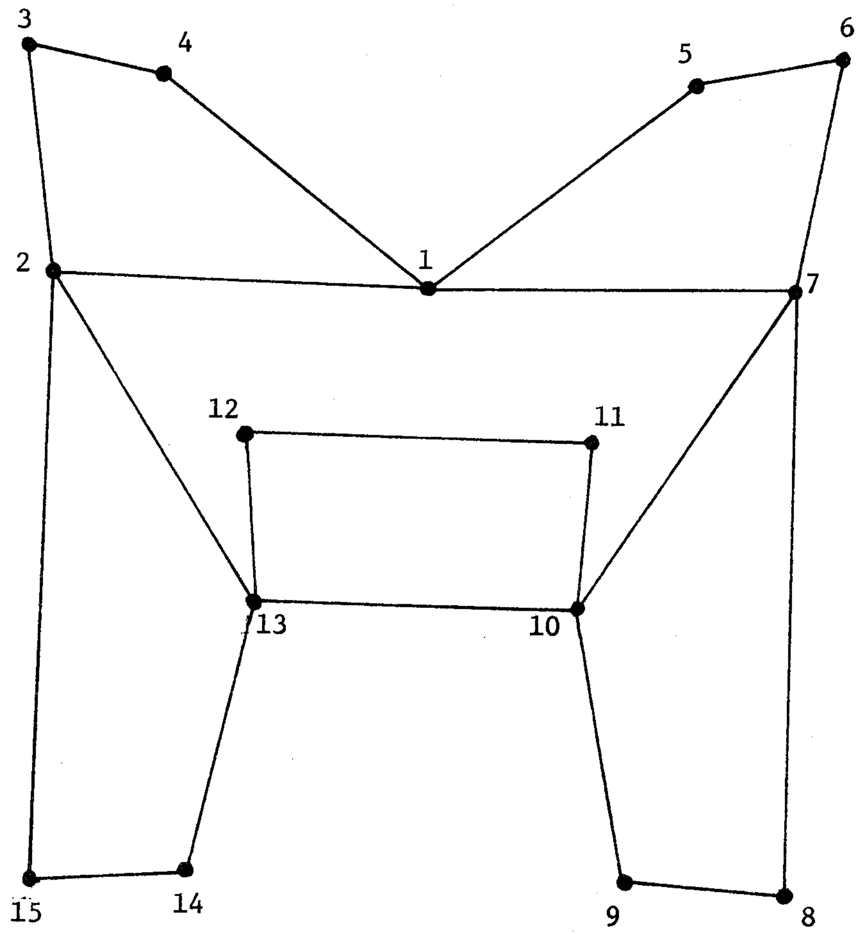
ORIGINAL PICTURE



GENERATED REPRESENTATION



ORIGINAL PICTURE



GENERATED REPRESENTATION

# B I B L I O G R A P H Y

## BIBLIOGRAPHY

( i) Friedman, Daniel P., Dickson, David C., Fraser, John J. and Pratt, Terrence W.; GRASPE 1.5: A Graph Processor and its Application; Department of Computer Science; University of Houston; 1969.

( ii) Lawsen, Harold W.; "PL/1 List Processing"; Comm. ACM 6 (June, 1967); pp. 358-367.

(iii) Hurwitz and Citron; "GRAF: Graphic Additions to FORTRAN"; Proc. SJCC; 1967; pp.553-558.

( iv) Frank, A.J.; "B-LINE, Bell Line Drawing Language"; Proc. FJCC; 1968; Part I; pp. 179-191.

( v) GPAK - Version II; IBM Document 360D-08.7.002; September; 1966.

( vi) "GASP, Graph Algorithm Software Package"; Quarterly Technical Progress Report; Department of Computer Science; University of Illinois; October; 1969; pp. 113-129.

(vii) Hart, Richard; HINT, A Graph Processing Language; CISSR Research Report; Michigan State University; February; 1969.

(viii) McCarthy, John, et al; LISP 1.5 Programmer's Manual; M.I.T. Press; 1968.

( ix) Wolfberg, Michael S.; An Interactive Graph Theory System; Moore School Report No. 69 - 25; University of Pennsylvania; June; 1969.

( x) Programmed Buffered Display 338 Programming Manual; Digital Equipment Corporation Document No. DEC-08-G61C-D; Maynard; Massachusetts; 1967.

( xi) Unger, S.H.; "GIT - A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism"; Comm. ACM. 7; No. 1; January; 1964; pp. 26-34.

( xii) King, Clarence Albert; "A Graph-Theoretic Programming Language"; Ph.D. Dissertation; University of the West Indies; 1970.

(xiii) Prins, Geert C. E.; "On the Automorphism Group of a Tree"; Ph.D. Dissertation; University of Michigan; 1957.

( xiv) Kately, Julian, "Automorphism Groups of Graphs"; Ph.D. Dissertation; Michigan State University; 1963.

( xv) Mowshowitz, Abbe; "Entropy and the Complexity of Graphs"; Ph.D Dissertation; University of Michigan; 1967.

( xvi) Frucht, R.; "Herstellung von Graphen mit vorgegebener abstrakter Gruppe"; Compositio Math.; 6; 1938; pp. 239-250.

(xvii) Kagno, I. N.; "Linear Graphs of Degree ≤ 6 and their Groups"; Amer. J. Math.; 68; 1946; pp. 505-520.

(xviii) Hemminger, R. L.; "On the Group of a Directed Graph"; Canad. J. Math.; 18; 1966; pp. 211-220.

( xix) Harary, F.; "On the Group of the Composition of Two Graphs"; Duke Math. J.; 26; 1959; pp. 29-34

( xx) Chao, C. Y.; "On Groups and Graphs"; Trans. Amer. Math. Soc.; 118; 1965; pp. 488-497.

( xxi) Jackson, W.; "A Note on a Computer Program for Finding the Automorphism Group of a Graph"; Unpublished.

(xxii) Fary, I.; "On Straight Line Representation of Planar Graphs"; Acta. Sci. Math. SZEGED; 11; 1948; pp. 229-233.

(xxiii) Tutte, W. T.; "How to Draw a Graph"; Proc. London Math. Soc.; 13; 1963; pp. 743-767.

(xxiv) Dean, Richard A.; "Elements of Abstract Algebra"; Wiley; 1966.

( xxv) <u>IBM System/360 Operating System Graphic Programming Services for the IBM 2250 Display Unit</u>; Form C27-6909-5.

(xxvi) <u>IBM System/360 Component Description IBM 2250 Display Unit Model 1</u>; Form A27-2701-2.

( xxvii) IBM System/360 Operating System PL/1 (F) Language Reference Manual; Form C28-8201-2

( xxviii) IBM System/360 Operating System PL/1 (F) Programmer's Guide; Form GC28-6594-7.

( xxix) IBM System/360 Operating System Assembler Language; Form GC28-6514-6.

( xxx) IBM System/360 Operating System System Control Blocks; Form GC28-6628-4.

( xxxi) IBM System/360 Operating System System Programmer's Guide; Form C28-6550-6.

( xxxii) IBM System/360 Operating System Supervisor and Data Management Macro Instructions; Form GC28-6647-3.

( xxxiii) IBM System/360 Operating System Supervisor and Data Management Services; Form C28-6646-2.

( xxxiv) Corneil, D. G. and Gotlieb, C. C.; "An Efficient Algorithm for Graph Isomorphism"; J. ACM 17; No. 1, January, 1970.

( xxxv) Chang, Manning and Metze; Fault Diagnosis of Digital Systems; Wiley; 1970.

( xxxiv) Erdos and Renyi; "Asymmetric Graphs"; Acta. Math. Acad. Sci. Hungar.; 14; 1963; pp. 295-315.

( xxxvii) Quintas, L. V.; "Extrema Concerning Asymmetric Graphs"; J. Combinatorial Th.; 3; 1967; pp. 57-82.