STRAIGHT-LINE REALIZATION
OF
PLANAR GRAPHS
by
LUANG-HUNG SHIAU
B.Sc. National Taiwan University, 1958
M.A. University of British Columbia, 1969

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the Department
of
Computer Science

We accept this thesis as conforming to the
required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

April, 1971
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date April 16, 1971
ABSTRACT

Few theorems are known about planar graphs. For example, Kuratowski proved that a graph is planar if and only if it has no subgraph homeomorphic to $K_5$ or $K_{3,3}$. It has remained as a direct criterion for determining whether a graph is planar or not. Powerful as the theorem is, it is not always easy to apply. This leads us to try some practical methods to test a planar graph.

In this thesis, we have an algorithm for finding an outer circuit for a simple connected planar graph. Then, we use this outer circuit to draw a straight line graph in the plane. The programme for this algorithm is written in FORTRAN for an IBM 360/67 Computer.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTION 1: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>SECTION 2: Definition of Terms and Theoretical Background</td>
<td>4</td>
</tr>
<tr>
<td>SECTION 3: A Computer Algorithm for Recognizing a Planar Graph</td>
<td>12</td>
</tr>
<tr>
<td>SECTION 4: Drawing a Planar Graph with Straight Lines Using the Adjacency Matrix</td>
<td>30</td>
</tr>
<tr>
<td>SECTION 5: Results and Examples</td>
<td>42</td>
</tr>
<tr>
<td>SECTION 6: How to Use the Programmes</td>
<td>55</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>59</td>
</tr>
<tr>
<td>APPENDIX A: The Main Programme for Finding an Outer Circuit</td>
<td>60</td>
</tr>
<tr>
<td>APPENDIX B: GRAPHPACK</td>
<td>67</td>
</tr>
<tr>
<td>APPENDIX C: All the Subroutines Used by the Programme Stated in Appendix A</td>
<td>74</td>
</tr>
<tr>
<td>APPENDIX D: The Main Programme for Drawing a Given Graph</td>
<td>84</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I am greatly indebted to Professor J. M. Kennedy for suggesting the topic of this thesis, for allowing me a generous amount of his time and for his many constructive comments during the preparation of this thesis. I also wish to thank Professor A. Mowshowitz for his criticism of the draft form of this work.

The financial support of the University of British Columbia and the National Research Council of Canada are gratefully acknowledged.
1. Introduction.

In the study of many problems arising in areas such as electrical networks and traffic control, it is often important to represent a system in the form of a simple linear graph. Since the problem is to draw the graph in a two-dimensional plane, it is desirable to draw it with a minimum number of edge crossings.

In 1930, Kuratowski proved that a graph is planar if and only if it has no subgraph homeomorphic to $K_5$ or $K_{3,3}$ (Fig. 1, Section 2) [4]. In the case of a simple planar graph, in 1948, Fary [1] has proved that if a finite graph can be represented on the plane at all, it can be represented with straight segments as edges. Powerful as those two theorems are, they are not always easy to apply.

An algorithm suitable for implementation on a digital computer for determining whether a graph $G$ is planar has been developed by Fisher and Wing [2] and [3]. All the operations are performed on matrices which are formed directly from the incidence matrix and their algorithm is based on a decomposition scheme. They have shown that $G$ is planar if and only if (1) an associated graph known as the Pseudo-Hamiltonian (Definition 2.24) graph is planar and (2) each decomposed subgraph is planar.

Our algorithm is based on a corollary to the decomposition theorem (Theorem 2.4). It leads to some practical methods by
which one can test and draw a simple connected planar graph easily, or at least systematically.

Before we introduce our algorithm in Sections 2, we give the definition of some terms and the theoretical basis for our algorithm.

Our programmes can be divided into two parts:
Part I (Section 3). All the operations in this part are performed in a representation that enables the user to describe operations in graph-theoretical terms such as edge, vertex etc., and also save storage space in the Computer. In Section 3, we explain how GRAPHPACK(Appendix B) represents a graph in the Computer and the main algorithm for finding an outer circuit of a planar graph.
Part II (Section 4). The purpose of this part of the programme is to have a straight-line representation of a planar graph in the plane if one has formed or can provide an outer circuit of the graph to be drawn. This part of programme can be used independently as long as the user provides all the required data. It will draw a graph by using the results obtained from Section 3 or the data provided by the user.

In Section 5 the execution time of the graphs of the five regular solid figures and some other graphs is given. Since our programme is written for planar graphs, it might fail to detect a nonplanar graph because \( G_1 \) (pp. 9) might be non-planar. Some of such examples are also given in this section.
In Section 6 we show how to use our programme.

Programming details of our algorithm are given in Appendices A to D.
2. Definition of Terms and Theoretical Background.

Before stating the theorems on which our algorithm is based, some standard terminology of graph theory will be introduced in this section. Further special definitions will be stated in Sections 3 and 4.

Definition 2.1 A graph, $G = (V, E)$, consists of a set $V$ of vertices, and a set $E$ of edges joining these vertices. $E$ may be considered as a subset of ordered pairs of $V \times V$.

Definition 2.2 A subgraph $G' = (V', E')$ of $G$ satisfies the conditions $V' \subseteq V$ and $E' \subseteq E$. $G'$ is said to span $G$ if $V' = V$.

Definition 2.3 A graph is said to be directed if directions are assigned to the edges, and is said to be undirected otherwise.

Definition 2.4 A graph is said to be finite, if it contains a finite number of edges.

Definition 2.5 An edge is said to be incident with the vertices it joins.

Definition 2.6 In a graph (directed or undirected), a path is an alternating sequence of distinct vertices and edges $v_0, e_1, v_1, \ldots, v_{n-1}, e_n, v_n$, beginning and ending with vertices, in which each edge is incident with the two vertices immediately preceding and following it.

Definition 2.7 A circuit is a path in which $v_0 = v_n$ and $n \geq 3$. 
Definition 2.8  The number of edges between two vertices in a graph is called the **multiplicity** of this pair of vertices. A graph that contains pairs of vertices with multiplicities larger then 1 is sometimes called a **multigraph**. A **linear graph** is a graph the multiplicities of the ordered pairs of vertices of which are no larger than 1.

Definition 2.9  A graph is said to be **simple** if it does not contain loops(edges joining the same vertex to itself) or multiple edges(several edges joining the same pair of vertices).

Definition 2.10  A graph is said to be **embedded** in a surface $S$ when it is drawn on $S$ so that no two edges intersect. A graph is **planar** if it can be embedded in the plane.

Definition 2.11  In a graph (directed or undirected), two vertices are said to be **connected** if there is a path between them.

Definition 2.12  An undirected graph is said to be **connected** if every two vertices in the graph are connected and is said to be **disconnected** otherwise.

Definition 2.13  In a graph (directed or undirected), the **degree of a vertex** is the number of edges that are incident with it. In a directed graph, the **indegree of a vertex** is the number of edges that incident into it, and the **outdegree of a vertex** is the number of edges that are incident from it.
Definition 2.14  The adjacency matrix $A$ of a graph $G$ with $n$ vertices is an $n \times n$ matrix, the $(i,j)$th entry of which is 1 if there is an edge joining the $i$-th and $j$-th vertices, and is 0 otherwise.

Definition 2.15  The incidence matrix $A$ of a graph $G$ with $n$ vertices and $m$ edges is an $n \times m$ matrix in which the rows correspond to the vertices and the columns correspond to the edges. The $(i,j)$th entry in the matrix is 1 if the $i$-th vertex is incident with the $j$-th edge and is 0 otherwise.

Definition 2.16  If $e_{ij} = v_i v_j$ ($v_i$ and $v_j$ are the initial and the terminal vertices of $e_{ij}$) is an edge of a graph $G$, and $v$ is not a vertex of $G$, then $e_{ij}$ is subdivided when it is replaced by the edges $v_i v$ and $vv_j$.

Definition 2.17  Two graphs are homeomorphic if both can be obtained from the same graph by a sequence of subdivisions of edges.

We shall describe some theorems in the following:

Kuratowski [4] has proved:

Theorem 2.1  A graph is planar if and only if it contains no subgraph homeomorphic to $K_5$ or $K_{3,3}$ (Fig. 1).

$K_5$:  

$K_{3,3}$:

Fig. 1
For some time after its publication in 1930, Kuratowski's theorem has remained the only direct criterion for determining whether a graph is planar or nonplanar. Powerful as the theorem is, it is not always convenient to apply. For example, consider the graphs shown in Fig. 2. It is not obvious which of the graphs contains Kuratowski's graph as a subgraph. The verification of the existence or nonexistence of a Kuratowski's basic nonplanar graph in a given graph may become quite difficult in some cases.

![Fig. 2](image)

In the case of a simple planar graph, Istvan Fary [1] has proved:

**Theorem 2.2** If a finite graph can be represented on the plane at all, it can be represented with straight segments as edges.

The main idea of his proof is to suppose that the theorem holds for graphs with \( n \) vertices and let \( G \) be a graph with \( n+1 \) vertices and having no multiple edges. By letting one of its edges shrink into a point we get a graph \( G^* \) having \( n \) vertices. Suppose first this graph \( G^* \) has no multiple edges; then by hypothesis we can draw it with straight segments as edges. Now,
stretch the vertex which corresponds to the shrunken edge of $G$ into a short straight edge. Thus we get $G$ drawn with straight segments as edges. Secondly, if $G^*$ has multiple edges, we can divide $G$ by a circuit of three edges into two subgraphs; drawing these with straight segments as edges, one outside and the other inside of a triangle, we obtain a straight representation of the graph $G$.

Although the fundamental step of Fary's proof is the construction of $G^*$, he did not specify which vertex of $G$ is to be chosen in order to construct $G^*$ or where $G^*$ is to be located in the plane. We shall discuss these in Section 4 in detail.

Before we introduce a decomposition theorem which is given by G. J. Fisher and O. Wing [3], we need some specific definitions for the terminology we are going to use:

**Definition 2.18** By the deletion of an edge we mean the removal of the edge but not the two vertices associated with the edge.

Let $C$ be a circuit in a graph $G$ and $G-C$ the subgraph of $G$ that remains when the edges of $C$ are deleted.

The edges of $G-C$ are classified as follows:

1. **Direct Connections**: edges with both vertices on $C$.
2. **Edges of Attachment**: edges with exactly one vertex on $C$.
3. **Exterior Edges**: edges with no vertex on $C$.

The bridges of $C$ in $G$ are defined by construction. First, delete the vertices of $C$ and all of the edges incident to these
vertices. Next, group the subgraph of G that remains into a set of connected components. Denote each component which consists of a single isolated vertex by $V_j$, and each of the remaining components by $G_i$. Finally, associate with each component the set of edges of attachment which reconnect it to C (Note that this set may be null for some $G_i$, if G is not connected).

From the above construction, we have

**Definition 2.19** A bridge of C in G is a subgraph of G which consists of (1) $V_j$ or $G_i$, (2) the edges which join the vertex $V_j$ or vertices of $G_i$ and the vertices on C and (3) the vertices on C which joined the above mentioned edges. If an edge in G-C which is a direct connection then the bridge consists of this edge and the two vertices on C which joins.

Thus a bridge of C in G belongs to one of the following types:

**Type 1**: a direct connection to C.

**Type 2**: a set of edges of attachment which connect a vertex $V_j$ to C.

**Type 3**: a connected component $G_i$, and the corresponding edges of attachment which connect $G_i$ to C.

Let $B$ be a bridge of C in a graph G.

**Definition 2.20** The vertices which are common to $B$ and C are called vertices of attachment of $B$.

We shall define a Pseudo-Hamiltonian graph $G'$ of a graph G with respect to C by construction again:

If G has Type 3 bridges which are not either Type 1 or 2, shrink
all the vertices but the attachment ones of each Type 3 bridge to form a "new" vertex. Then join each "new" vertex to the vertices of attachment of the respective bridges and form a "new" Type 2 bridge. This "new" graph $G'$ will be called a Pseudo-Hamiltonian graph of $G$. In general, we have

Definition 2.21 A graph is Pseudo-Hamiltonian if its decomposition with respect to a specified circuit $C$ consists of bridges which are all of Type 1 or 2.

Let $G'$ be a Pseudo-Hamiltonian graph with respect to a circuit $C$. Let $B$ be a bridge of $C$ in $G'$. Assume that $B$ can not be separated from $C$, i.e., there are at least two vertices of attachment. Let the vertices of attachment of $B$ be ordered in a clockwise sense on $C$. The successive vertices in this ordering divide $C$ into a set of edge-disjoint paths. Suppose $B'$ is a bridge of $C$ which is distinct from $B$, but possibly having the same vertices of attachment as $B$.

Definition 2.22 We say that $B'$ does not alternate with $B$, if all the vertices of attachment of $B'$ lie on a path defined by two successive vertices of attachment of $B$. Otherwise, we say that $B'$ alternates with $B$.

Further, if $B'$ alternates with $B$, then $B$ alternates with $B'$.

Now, we can state the decomposition theorem.
Theorem 2.3  Let G be a graph, G' the Pseudo-Hamiltonian graph of G with respect to a circuit C. G" the decomposed subgraph (is formed by the union of C and a bridge of Type 3) for each component G_i. G is planar if and only if

(1) the Pseudo-Hamiltonian graph G' is planar, and

(2) each decomposed subgraph G"_i is planar.

The problem of determining the planarity of an arbitrary graph is, therefore, by Theorem 2.3, reduced to the problem of determining whether or not a Pseudo-Hamiltonian graph is planar.

By the definition of alternation, one observes that bridges which alternate must be mapped on opposite sides of C if no two edges are to cross. On the other hand, bridges which do not alternate are not so constrained and may be mapped on either the same or on opposite sides of C. Thus we have

Theorem 2.4  A Pseudo-Hamiltonian graph G' is planar if and only if its bridges can be associated with two disjoint classes, I and O, such that no two bridges in the same class alternate.

Our algorithm uses Theorem 2.4 to find an outer circuit C of a planar graph G, and Theorem 2.2 to find a straight line representation of G in the plane.

In (3.2) of Section 3 we shall give the reasons why we have to find an outer circuit in order to draw a planar graph.
3. A Computer Algorithm for Recognizing a Planar Graph.

Assume that the graph $G$ is planar, connected and all vertices are of degree $\geq 2$.

Since a disconnected graph can always be treated as a collection of components each of which is a connected graph or an isolated vertex, it is only necessary to be concerned with the connected graphs. Furthermore, vertices of degree 1 do not alter or affect the construction of the straight line graph, so we shall concentrate on graphs whose vertices are of degree $\geq 2$. Some examples and comments about graphs with vertices of degree 1 will be given in Section 5 and Remarks 3.1, 4.1 and 4.2.

First, we shall explain how the algorithm recognizes a planar graph.

(3.1) Representation of a Graph in the Computer.

All the operation of the algorithm for testing planar graphs and extraction of planar graphs by Fisher and Wing [2] and [3] are expressed in terms of incidence matrix. By contrast, we use separate tables of edges and vertices to represent a graph. This representation not only allows one to deal in graph-theoretical terms such as edge, vertex, etc., but also saves storage space in the Computer, since the space used is proportional to the numbers of vertices and edges, rather than to some higher powers of these numbers. This feature is desirable if graphs are fairly sparse — i.e. if the degree of a vertex is
A set of subroutines known as GRAPHPACK was written originally for IBM 7040 by Dr. J. M. Kennedy and Mrs. J. Fowler and has been converted to IBM 360/67 with amendments and corrections by L. H. Shiau.

GRAPHPACK is briefly described in the following (the details are in Appendix B):

A number of singly-indexed integer arrays are used as follows:

(3.1.1) \( V, VPROP, EDO \) and \( EDI \) are **Vertex Tables**. \( NV \), the number of vertices in the graph cannot exceed 128 in the present version of GRAPHPACK.

(3.1.2) \( P, S, EPROP, SL, PL \) and \( IDEL \) are **Edge Tables**. \( NE \), the number of edges in the graph cannot exceed 400 in the present version of GRAPHPACK.

The following are subroutines of GRAPHPACK:

(3.1.3) **INITL** — (Initialize), **NEWED(V1,V2)** — (New edge) and **NEWVE(VN,V1,V2)** — (New vertex) are **Subroutines for Building Graphs**.

(3.1.4) **DELED(V1,V2,I)** — (Delete edge), **DELVE(VN)** — (Delete vertex), **RESTED(V1,V2,I,ERR)** — (Restore edge), **RESTVE(VN)** — (Restore vertex) and **RESTAL** — (Restore all edges and vertices) are **Subroutines for Deletion and Restoration of Edges and Vertices**.
(3.1.5) NEXEDO(VN,K) — (Next edge out), NEXEDI(VN,K) — (Next edge in) and NEXEDG(VN,K) — (Next edge) are for Traversal of a Graph.

(3.1.6) VERNAM(VN,J) — (Find a vertex). This subroutine is used by many of the other subroutines and may also be wanted in the mainline programme. A call of VERNAM sets J to the index of the vertex VN in the vertex tables.

(3.2) Reasons for Finding an Outer Circuit of a Planar Graph.

Having properly stored a graph in the Computer using GRAPHPACK, we have to explain why we want to find an outer circuit of a planar graph in order to draw it in the plane.

Definition 3.2.1 A finite (or inner) face of a planar graph embedded in the plane is a connected domain of the plane bounded by lines representing edges of the graph. The bounding edges taken in sequence form the circuit of the face.

Definition 3.2.2 The infinite (or outer) face of a planar graph embedded in the plane is the domain of the plane which is not bounded by edges of the graph.

Definition 3.2.3 The term face means either a finite or an infinite face.

Since each face is a domain bounded by a circuit, the complete set of circuits of the graph G, which includes all
the circuits plus the outer circuit bounding the planar graph $G$ is unique in the following sense: by choosing any one of the circuits as external, it is possible (1) to have the planar graph inscribed in it, and (2) the set of circuits is the same regardless of which circuit is chosen as the outer one [10].

Thus we can start to find an outer circuit of a given planar graph $G$.

(3.3) An Algorithm for Finding an Outer Circuit of a Planar Graph.

The algorithm is divided into four steps.

**Step 1** Pick a circuit $C$ in $G$, find its bridges and form its Pseudo-Hamiltonian graph $G'$. This is done by shrinking all Type 3 bridges until all vertices not on $C$ are coincident.

**Method:** Vertices of $G$ are read in pairs $(V_1,V_2)$ with the "elasticity" of the edge formed by $V_1$ and $V_2$. To form new edges the main line programme calls $\text{NEWED}(V_1;V_2)$ and $\text{NEWED}(V_2,V_1)$ since we are working on an undirected graph.

The reason why we need the "elasticity" of each edge will become clear at the end of Section 4.

To find a circuit $C$, the main programme calls $\text{APATH}(3.4.1.1)$
which returns the names of vertices on \( C \) in array \( \text{LIST} \).

To find its bridges, we first delete the vertices on \( C \) and all the edges incident to those vertices by calling \( \text{DELVE} \) for each vertex in \( \text{LIST} \). Next, group the subgraph of \( G \) that remains into a set of connected components by calling \( \text{ALPATH} \) (3.4.1.3) which returns the number of connected components in \( \text{VAL} \) and the vertices of connected components in array \( \text{ANS} \). If there are two or more connected components, terminator symbols, i.e. 9999, are inserted in \( \text{ANS} \) to separate them. Then call \( \text{RESTAL} \) to restore all deleted edges and vertices to \( G \).

A vertex, which is not a vertex of attachment, on \( C \) is said to be of Type 0. To obtain Pseudo-Hamiltonian graph \( G' \), every shrinking Type 3 bridge is replaced by a "new" vertex which is not on \( C \) and "new" edges are assigned between this vertex and all the vertices of attachment of this bridge. The vertices of attachment and those shrinking vertices are said to be of Type 3 and the "new" vertex is of Type 5. If the bridge is of Type 2, the vertex which is not on \( C \), \( V_i \), and the vertices of attachment are said to be of Type 2. If the bridge is of Type 1, the two vertices of attachment are said to be of Type 1.

Two arrays \( \text{VPROP} \) and \( \text{EPROP} \) are used to store these properties.

\( \text{VPROP} \): The lowest byte of \( \text{VPROP}(I) \) has the type of \( I \)-th vertex in array \( V \), the higher order bytes have the "name" of the
"new" vertex if the I-th vertex is of Type 3. If the I-th vertex is of Type 5, the higher order bytes contain its weight(number of Type 3 vertices contained in this "new" vertex).

Note, in order to distinguish whether a vertex is on C or not, vertices of attachment of Types 2 and 3 are stored in the lowest byte of VPROP(I) as a zero.

EPROP: This is used to store the "representative" of each bridge and its vertices of attachment. The highest order byte of EPROP(I) has the index J of LIST(J) (since we have to keep track of the location of each vertex of attachment on C in Step 2, to find the alternation of bridges) if LIST(J) is a vertex of attachment. The middle two bytes have the vertex "name", LIST(J), and the lowest byte has its type. Each bridge is terminated by -1.

Since array ANS has all the vertices of each connected component, we use this to find all the bridges and store the properties of each vertex in VPROP and EPROP.

For example, if a graph is given as in Fig. 3, then
we shall have Tables 1, 2 and 3:

<table>
<thead>
<tr>
<th>LIST</th>
<th>NV</th>
<th>NEW VERTEX (or WEIGHT)</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1

<table>
<thead>
<tr>
<th>VPROP</th>
<th>EPROP</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX of LIST</td>
<td>LIST</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2

Recall that the vertices of attachment of a bridge $B$ in
G with respect to C divide C into a set of edge-disjoint paths. Now, we shall "paint" each path a different colour, i.e. the vertices on C which are in a path have the same colour and the connecting vertices are doubly-coloured. By the definition of alternate bridge, we say B' does not alternate with B if all the vertices of attachment of B' are in one edge-disjoint path, i.e. all the vertices of attachment of B' have the same colour. Otherwise they have different colours.

Our programme identifies the colours by different small prime numbers 2, 3, 5, ..., etc., and uses the product of the primes for a multicoloured vertex. Then, two vertices are of the same colour if and only if their H.C.F. (highest common factor) is greater than 1. The "colour" of each vertex on C is stored in the higher order 3 bytes of each LIST(I), I=1,L-1.

For example, if we want to "paint" the circuit C (from Table 1) in Fig. 3 by using the first bridge B in EPROP (from Table 3) we store B in array IN. Then we shall have Fig. 4 (a subgraph of Fig. 3) and a "new" LIST (Table 4) in which the Type 3 bridge 11 or vertex 11 is connected to vertices 1, 2 and 4.
Then any other bridge, say B', can be drawn inside the circuit C if and only if all its vertices of attachment are of the same colour, i.e. B' does not alternate with B.

Now, if B'(the second bridge) in EPROP alternates with B, then B' will be stored in array OUT. Otherwise it is stored in array IN. If there are two or more bridges in array IN, we have to "repaint" the circuit C (since all the bridges of G with respect to a circuit must be tested against each other for their alternation properties).

For example in Fig. 3, the Type 2 bridge 7 or vertex 7 is connected to vertices 4, 5 and 6. Since their colours are the same (their common factor is 5), we see that bridge 7 does not alternate with bridge 11. Hence bridge 7 is stored in array IN and we have to "repaint" the circuit C. To do this subroutine REPANT(3.4.2.2) is called and in this example, we obtain Fig. 5 and "new" LIST (Table 6) as follows:

![Diagram](image)

**Fig. 5**

**Table 6**
In proceeding through the bridges of a given graph, we do those of Type 3 first, then those of Type 2 and Type 1 (in fact, our programme stores all the bridges in EPROP in this order). We also do this in a way that tries to put as many bridges as possible "inside" C, i.e. in array IN. This is done because our aim is either to show that C is an outer circuit or else to simplify the algorithm in Step 4 for finding a circuit outside C.

**Step 3**  If all bridges are inside C, i.e. no bridge is in array OUT, then C is the outer circuit and we are done. If there are further bridges in array OUT, we then have to "erase" the colour of C (call subroutine ERASE(3.4.2.3)). Then "paint" by the first bridge in array OUT etc., in order to test all the bridges in array OUT. If there is any pair of bridges in array OUT alternating with each other, then we conclude that the given graph is nonplanar and a message will be delivered. In this case, the testing will be terminated and no graph will be drawn. Otherwise, go to Step 4 to find a new circuit that is "outside" C — sooner or later the resulting circuit will be outside everything as required.

**Step 4**  Getting the external (outer) circuit C'. Having done the tests in Step 3, we may assume that all the bridges B_i in array OUT do not alternate. Find what this piece is, and replace it by a "detour" via C_i. Then find C' outside C, replace C by C' and go back to Step 2.

**Method:**  First we delete all bridges "inside" C, and restore
all bridges "outside" C. If the bridge in OUT is of Type 3, restore all the "shrinking" vertices and their edges connected to C, delete the "new" vertex and put VPROP("new" vertex) = -1. The last one indicates that this "new" vertex will not be used in the following iteration. Since we have tried in Section 2 to put the Type 3 bridges first and then as many bridges of other types as possible in array IN, it will save us some unnecessary work.

Next "erase" the coloured LIST which has been "painted" in Step 2. Then delete the edges on C which are covered by the bridges in array OUT, and call subroutine APATH(3.4.1.1) to find a second circuit C', replace C by C', and go back to Step 2.

For example, in Fig. 3, after Step 3 has been completed, we get two classes of bridges stored in arrays IN and OUT as shown in Tables 7 and 8.

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 7
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 8
```
Note that duplicate vertices of attachment of a bridge have been eliminated and those that remain are in the order of their occurrence on C.

Since there is one bridge in array OUT in this example, in Step 4 we find a new circuit $C' = (1,3,4,5,6)$. Repetition of Step 2 gives Fig. 6, Tables 9, 10 and 11. For the second iteration, the "new" vertex 12 is used. $C'$ is "painted" only once for there are only two bridges of G with respect to $C'$. Both are stored in array IN since they do not alternate with each other. Thus the outer circuit is $(1,3,4,5,6)$ and we are done.

![Diagram](image)

**Table 9**

<table>
<thead>
<tr>
<th>NV</th>
<th>VPROP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>-1</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 10**

<table>
<thead>
<tr>
<th>EPROP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

**Table 11**

<table>
<thead>
<tr>
<th>LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
In Step 4, if an outer circuit cannot be found because some $G_4$ is nonplanar, a message will be delivered and no graph will be drawn.

The details of this algorithm for finding an outer circuit are in Appendix A.

(3.4) Subroutines.

In the above algorithm (3.3), we use the following subroutines to find a circuit, "paint" it and classify the bridges. The details are in Appendix C.

(3.4.1) Paths or Connected Components of a Graph.

(3.4.1.1) APATH(I,END,LIST,L,FAIL) —(Find a path from vertex I to vertex END).

Given an initial vertex I and a terminal vertex END, a call of APATH will find a path from I to END and put all the vertices along the path (including I and END) in array LIST. L is the length of LIST.

FAIL is a flag, set to be 1 if there is no path from I to END, otherwise 0.

If END = I, then APATH will give a circuit.

(3.4.1.2) PATH(I,NODE,NO,ANS,IX) —(Find a path starting at I).

Given any vertex I, this subroutine will find a path from I as far as possible. NODE has all the vertices in G which have been tested in order to find a path and NO is number of elements
in NODE. If a path is found, all the vertices (including I) along the path are stored in array ANS and terminated by a symbol, 9999.

IX is the number of elements in ANS.

This subroutine is called by ALPATH(3.4.1.3) to find a connected component in a graph.

(3.4.1.3) ALPATH(VAL,NODE,NO,ANS,IX) — (Find all paths).

Given any graph G or G-C where C is a circuit, a call of ALPATH will find all the connected components of G or G-C.

NODE, NO, ANS and IX have the same meaning as those of (3.4.1.2).

VAL is the number of paths (or connected components) in G or G-C.

ALPATH is called by the main programme.

In the following subroutines, the arguments LIST and L are the same as those previously defined.

(3.4.2) "Paint" a Circuit C.

(3.4.2.1) PAINT(LIST,L,SPF,PP,SI,IN,I) — (Paint C).

In this subroutine, array PRIME contains some small prime numbers in "natural" order.

If there are two or more bridges in array IN, the circuit C might be "repainted", we have to keep track of the first and the last primes having been used in painting.
The elements in array PRIME, from PRIME(SPP) to PRIME(PP), are the prime numbers being used to "paint" C.

Since the elements in array IN, from IN(SI) to IN(I), are vertices of attachment of a bridge and divide C into edge-disjoint paths, we paint the vertices (on C) of each edge-disjoint path a same small prime number. If a vertex is both in array I and on C, it will be "painted" twice. The prime numbers being used to paint the vertices are stored in the higher order 3 bytes of their corresponding location in array LIST. Note that the last vertex in array LIST is not painted as it is the same as the first.

(3.4.2.2) REPANT(LIST,L,SPP,PP,SI,IN,I) —(Repaint C).

All the arguments are the same as those of PAINT.

This subroutine is called by ALTER(3.4.3.4), if we have to "paint" the circuit C again. The rules for painting are the same as those of PAINT.

(3.4.2.3) ERASE(LIST,L) —(Erase the colours).

This subroutine will "erase" all the colours being painted, i.e., the array LIST contains only the "name" of all the vertices on C in the lowest order byte.

(3.4.3) Classify all the Bridges of a Graph G with respect to a Circuit C.

(3.4.3.1) SAME(IN,I)

This subroutine simply eliminates the same elements in
array \ IN \ which \ has \ I \ elements; \ the \ "new" \ array \ will \ be \ put 
back \ in \ array \ \ IN \ and \ I \ is \ the \ number \ of \ elements \ in \ the \ "new" 
array \ \ IN. 

(3.4.3.2) \ ORDER(IN,I) 
Rearrange \ the \ given \ array \ \ IN \ with \ I \ elements \ such \ that 
the \ highest \ order \ byte \ of \ each \ element \ of \ array \ \ IN \ to \ be \ in 
natural \ order \ and \ put \ the \ result \ in \ array \ \ IN. 

(3.4.3.3) \ ALTNAT(LIST,L,SI,IN,I,TEST,TEST,TE,SO,OUT,OT, 
NOBIN,NOBOUT) —(Test \ if \ two \ bridges \ alternate \ or \ not). 
Use \ the \ bridge \ B, \ from \ IN(SI) \ to \ IN(I), \ to \ "paint" \ \ C. 
The bridge \ B', \ from \ TEST(STEST) \ to \ TEST(TE), \ is \ to \ be \ tested 
against \ B. \ If \ B' \ alternates \ with \ B, \ then \ B' \ is \ stored \ in 
array \ OUT, \ from \ OUT(SO) \ to \ OUT(OT), \ otherwise \ in \ array \ \ IN. 
In \ the \ latter \ case, \ both \ SI \ and \ I \ will \ be \ increased \ by \ the 
length \ of \ array \ \ TEST. 
NOBIN \ and \ NOBOUT \ are \ the \ number \ of \ bridges \ in \ arrays \ \ IN 
and \ \ OUT \ respectively. 
This \ subroutine \ is \ called \ by \ subroutine \ ALTER(3.4.3.4). 

(3.4.3.4) \ ALTER(BC,LIST,L,IN,I,OUT,OT,NOBIN,NOBOUT,*) —(To 
find \ the \ alternation \ property \ among \ all \ the \ bridges). 
All \ the \ arguments \ are \ the \ same \ as \ those \ of \ \ ALTNAT, \ except 
BC \ which \ is \ the \ length \ of \ array \ \ EPROP. \ Recall \ that \ \ EPROP \ has 
all \ the \ bridges \ of \ \ G \ with \ respect \ to \ \ C. 
The \ main \ line \ programme \ calls \ ALTER \ to \ classify \ all \ the 
bridges \ of \ \ G \ with \ respect \ to \ \ C \ into \ two \ classes \ which \ are
stored in arrays IN and OUT.

If BC = 0, then the given graph is a simple circuit, this subroutine will do nothing and return to the main programme by RETURN 1.

If there is only one bridge, then it will simply put this bridge in array IN and return to the main programme by RETURN 1, i.e. the outer circuit is found.

Otherwise, it will do the following:
(1) Whenever a bridge in EPROP is copied into arrays IN or TEST, subroutines ORDER and SAME are called.
(2) The first bridge, say B, in EPROP is copied into array IN, then subroutine PAINT is called to "paint" the circuit C by using this bridge.
(3) The next bridge, say B', in EPROP is copied into array TEST, subroutine ALTNAT is called to test if B and B' alternate with each other, if so copy TEST into array OUT, otherwise into array IN.
(4) If there are two or more bridges in array IN and EPROP, the circuit will be "repaint" (call subroutine REPANT) by using all the bridges in array IN except the first one. The bridges left in EPROP will be tested one by one against those in array IN by calling ALTNAT until all the bridges in EPROP are exhausted.

Remark 3.1 Separable bridges, i.e. those bridges having only one vertex of attachment, in particular, a vertex of degree one, are ignored and deleted in this subroutine (Examples 5.1 and 5.2)
Having found the outer circuit by the method described in this section, we are in a position to draw the given planar graph in the plane.
4. Drawing a Planar Graph with Straight Lines Using the Adjacency Matrix.

Some additional definitions are required for the following discussion.

**Definition 4.1** Let $C$ be a circuit of a graph $G$ and $\#(C)$ denote the number of bridges of $G$ with respect to $C$. If $\#(C) \leq 1$, we call $C$ a **peripheral circuit** of $G$.

**Definition 4.2** The **connectivity** $k = k(G)$ of a graph $G$ is the minimum number of vertices whose removal results in a disconnected or trivial graph. A graph $G$ is **$n$-connected** if $k(G) \geq n$.

Let $C$ be a peripheral circuit of a 3-connected graph $G$ having no Kuratowski subgraphs. (Two examples, Examples 5.2 and 5.4, for 1-connected and 2-connected graphs respectively are given in Section 5.)

Recall that $NV$ is the number of vertices in $G$ and $(L-1)$ is the number of vertices of $G$ on $C$.

Let $Q$ be a (geometrical) $(L-1)$-sided convex polygon in the Euclidean plane. We shall define a function in the following:

**Definition 4.3** Let $f$ be a 1-1 mapping of the vertices on $C$ onto the set of vertices on $Q$ such that the cyclic order of vertices on $C$ agrees, under $f$, with the cyclic order of vertices on $Q$. Let's enumerate the vertices of $G$ as $V_1, V_2, \ldots, V_{NV}$
so that the first \((L-1)\) are the vertices of \(G\) on \(C\). We extend \(f\) to the other vertices of \(G\) by the following rule. If \((L-1) < i \leq NV\), let \(A(i)\) be the set of all vertices of \(G\) adjacent to \(V_i\). For each \(V_j\) in \(A(i)\), let a unit mass \(m_j\) be placed at the point \(f(V_j)\). Then \(f(V_i)\) is required to be the centroid of the masses \(m_j\). The extended \(f\) is called a \textit{barycentric mapping} of \(G\).

**Definition 4.4** Let \(U_1, U_2, \ldots, U_k\) be subsets of a given set \(U\), not necessarily all distinct. Their \textit{mod 2 sum} is the set of all \(u \in U\) such that the number of suffixes \(i\) satisfying \(u < S_i\) is odd.

**Definition 4.5** Let \(E(G)\) be the set of edges of a graph \(G\). A \textit{cycle} of a graph \(G\) is a subset \(S\) of \(E(G)\) such that the number of links of \(G\) in \(S\) incident with any vertex of \(G\) is even. If \(C\) is any circuit of \(G\) then \(E(C)\) is a cycle, we call a cycle of this kind \textit{elementary}.

**Definition 4.6** A \textit{planar mesh} of \(G\) is a set \(M = \{S_1, S_2, \ldots, S_k\}\) of elementary cycles of \(G\), not necessarily all distinct, which satisfies the following conditions. (1) If an edge of \(G\) belongs to one of the sets \(S_i\), it belongs to just two of them. (2) Each non-null cycle of \(G\) can be expressed as a mod 2 sum of members of \(M\).
In 1963, W. T. Tutte [8] has shown by using barycentric mapping that if $G$ is a simple graph having a planar mesh, then one can find a straight representation of $G$ in the plane. He has also proved that if $G$ is any graph, propositions "$G$ is planar", "$G$ has a planar mesh", and "$G$ has no Kuratowski subgraph" are equivalent. Hence by Fary's Theorem (Theorem 2.2 in Section 2), we can draw a planar graph with straight lines in the Euclidean plane.

Assume that graph $G$ with vertices $V_i$ ($i=1,2,\ldots,N_V$) is undirected and simple. Then the adjacency matrix $A=(a_{ij})$ is symmetric and the diagonal elements $a_{ii}$ are zero, since loops are forbidden.

The problem is to locate the vertices $V_i$ at positions $(x_i, y_i)$ in the plane in a way that allows the edges to be drawn with non-intersecting straight lines.

Our method, "The Elastic Model", is to consider a mechanical model of a given graph $G$, in which the edges are represented by short (but very stretchable) pieces of elastic rubber bands. Suppose the vertices are set at some positions so that all the elastics are stretched. Then the system is in mechanical equilibrium if the net force acting on each vertex is zero.

As shown in Fig. 7, let us consider the vertex $V_i$, and let $V_j$ be one of the vertices adjacent to $V_i$. Let $d_{ij}$ be
the distance between them. Hooke's Law of elasticity says that the force is proportional to the stretching, and since the initial length was very small, the force between $V_i$ and $V_j$ has magnitude $F_{ij} = d_{ij}$ (omitting a constant of proportionality).

This force can be resolved into $x$ and $y$ components by multiplying by the appropriate Cosines or Sines:

$$F_i(x) = \frac{x_j - x_i}{d_{ij}} \cdot d_{ij} = x_j - x_i$$

$$F_i(y) = \frac{y_j - y_i}{d_{ij}} \cdot d_{ij} = y_j - y_i$$

The total force on $V_i$ comes from summing over all edges; this must be zero for equilibrium. Hence

$$\sum_j a_{ij} (x_j - x_i) = 0$$

(4.1)

$$\sum_j a_{ij} (y_j - y_i) = 0$$ for $i=1,2,\ldots,NV.$
These can be rewritten as

\[-(\sum_j a_{ij})x_i + \sum_{j\neq i} a_{ij}x_j = 0, \quad i=1,2,\ldots, NV\]

and the same for the \(y_i\). Thus (4.1) can be written as

\[(4.2) \quad DZ = 0\]

where \(Z = \begin{pmatrix} x_1 & x_2 & \cdots & x_{NV} \\ y_1 & y_2 & \cdots & y_{NV} \end{pmatrix}^T\) and the matrix \(D\) of coefficients for the linear system is just the adjacency matrix \(A\) with the negative of its row sums inserted on the diagonal.

In order to get a non-trivial solution of (4.2), we have to fix some vertices and let the others take up their equilibrium positions.

Tutte [8] has shown that if a graph \(G\) is connected and planar, fixing the vertices of any one face is necessary and sufficient to give a unique solution, so we shall fix the vertices which are in array LIST on \(C\) (The outer circuit obtained in (3.2), Section 3) to solve (4.2).

Without loss of generality, suppose that \(V_1, V_2, \ldots, V_{L-1}, \quad (L-1) \leq NV\) are on \(C\) and the coordinates of \(V_k\) are \((p_k, q_k)\), then the \(k\)-th equation in (4.1) is replaced by

\[x_k = p_k \quad \text{and} \quad y_k = q_k\]

in the "x-set" and "y-set" respectively. Then (4.2) can be rewritten as

\[(4.3) \quad D'Z = B\]
where \( B = \begin{pmatrix} p_1 & p_2 & \cdots & p_{L-1} & 0 & \cdots & 0 \\ q_1 & q_2 & \cdots & q_{L-1} & 0 & \cdots & 0 \end{pmatrix}^T \), \( Z = \begin{pmatrix} x_1 & x_2 & \cdots & x_{NV} \\ y_1 & y_2 & \cdots & y_{NV} \end{pmatrix}^T \),

\( L = \) number of elements in LIST and

\[
D' = \begin{pmatrix}
I_{(L-1)*(L-1)} & 0_{(L-1)*(NV-L+1)} \\
\cdots & \cdots \\
(\text{the same as those entries of } D)
\end{pmatrix}
\]

For example, let \( G \) be the "graph of a cube", as shown in Fig. 8.

![Fig. 8](image-url)

Let

\[
A = \begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{pmatrix}
\]

be the adjacency matrix of \( G \), then

\[
D = \begin{pmatrix}
-3 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & -3 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & -3 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & -3 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & -3 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & -3 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & -3
\end{pmatrix}
\]

and

\[
D' = \begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & -3
\end{pmatrix}
\]
If \( B = \begin{pmatrix} 0 & 10 & 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 10 & 0 & 0 & 0 & 0 \end{pmatrix}^T \)

then

\[
Z = \begin{pmatrix} 0 & 10 & 10 & 0 & \frac{20}{3} & \frac{20}{3} & \frac{10}{3} & \frac{10}{3} \\ 0 & 0 & 10 & 10 & \frac{10}{3} & \frac{20}{3} & \frac{20}{3} & \frac{10}{3} \end{pmatrix}^T
\]

and Fig. 9 is a representation of the graph of the cube in the plane.

![Fig. 9](image)

The programme is quite simple using UBC Library subroutines:

1. FSLE \([9a]\) to solve a system of linear equations \( AX = B \) and
2. Plotting routines \([9b]\).

Of course, we must decide how to choose the locations of the vertices on the outer circuit.

Let \( G \) be the given graph. Recall that in (3.3), Section 3:

1. Each edge of \( G \) is defined by its initial and terminal vertices with its "elasticity", in general the "elasticity" is 1 for we use this to form matrix \( A \). But sometimes we may wish to change the "elasticity" of some of the edges in a graph (the reason why
we want to do this will become clear later in this section).

(2) We have NV and NE, the number of vertices and edges of G, respectively. (3) We also have the name of vertices on the outer circuit C which are stored in array LIST(I), I=1,L. Those data can be stored in a file and the file supplied as input data to the graph-drawing programme.

There is a flag, called IFLAG in the second part of our programme (Appendix D), which is to be set by the user. If IFLAG = 0, the data are read from the file designated as UNIT 3, otherwise from UNIT 5 = *SOURCE*(Remark 4.3). In the former case, i.e. IFLAG = 0, our programme will assign the outer circuit in the first quadrant of XY-plane as a (L-1)-regular polygon in the order in which the vertices are given. Their coordinates are thus obtained from our programme to form matrix B and the adjacency matrix A will be completed by using the "elasticity" of each edge. Otherwise, the outer circuit will be assigned by the user anywhere in the plane he wishes. In this case, the user has to provide all the edges with their initial and terminal vertices along with their "elasticities", the names of vertices on the outer circuit and matrix B.

In doing so, we let the user use the second part of our programme independently, i.e. one can draw a graph if one wishes without using the first part of our programme.

Now, we have all the required data, the programme then forms the matrix D' from matrix A and calls subroutine FSLE to solve
D'Z = B in (4.3). The unique nontrivial solution is stored in array Z. Finally, the plotting subroutines are called to plot $G$.

For some graphs, it is important to get the "best" outer circuit. For a given graph, two different outer circuits might result in having two completely different straight line representations. Such an example will be given in Section 5 (Fig. 17a and Fig. 17b). In some cases, the coordinates of the solution in equation (4.3) might be too close to be plotted by the plotter as distinguished points in the plane because the outer circuit obtained in our algorithm is not the "best" one. One may want to improve the appearance of the graph as to change the "elasticity" of certain edges in $G$.

We shall show how to obtain matrices $A$, $D$ and $D'$ if we have to change the "elasticity" of certain edges of $G$ by an example.

Initially, let us consider $G$, the "graph of a Tetrahedron" as shown in Fig. 10, then we have the adjacency matrix $A$ as

$$A = \begin{pmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0
\end{pmatrix}.$$

From $A$ we obtain
\[
D = \begin{pmatrix}
-3 & 1 & 1 & 1 \\
1 & -3 & 1 & 1 \\
1 & 1 & -3 & 1 \\
1 & 1 & 1 & -3
\end{pmatrix}
\quad \text{and} \quad
D' = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 1 & 1 & -3
\end{pmatrix}
\]

If the vertices 1, 2 and 3 are assigned at (0,1), (0,0) and (1,0) respectively in the plane, then by solving the equation (4.3), the coordinates of vertex 4 is then \((1/3,1/3)\) (Fig.11a).

Suppose the "elasticities" of the edges 4 to 1(or 1 to 4), 4 to 2(or 2 to 4) and 4 to 3(or 3 to 4) in \(G\) have been changed to 2, 5 and 9 respectively, instead of forming the adjacency matrix of \(G\), we shall form the matrix \(A^*\) as follows:

\[
A^* = \begin{pmatrix}
0 & 1 & 1 & 2 \\
1 & 0 & 1 & 5 \\
1 & 1 & 0 & 9 \\
2 & 5 & 9 & 0
\end{pmatrix}
\]

and matrices \(D^*\) and \(D'^*\) are obtained according to the scheme we stated in early part of this section.

\[
D^* = \begin{pmatrix}
-4 & 1 & 1 & 2 \\
1 & -7 & 1 & 5 \\
1 & 1 & -11 & 9 \\
2 & 5 & 9 & -16
\end{pmatrix}
\quad \text{and} \quad
D'^* = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
2 & 5 & 9 & -16
\end{pmatrix}
\]

Now if

\[
B^* = \begin{pmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0
\end{pmatrix}^T
\]

then by solving

\[
D'^*Z^* = B^*
\]
we have

\[ Z^* = \begin{pmatrix} 0 & 0 & 1 & 9/16 \\ 1 & 0 & 0 & 1/8 \end{pmatrix} \]

(Fig. 11b).

In short, if we change the "elasticity" to \( m_{ij} \) of an edge, say \( V_i \) to \( V_j \), then the "1" in the \((i,j)\)th and \((j,i)\)th entries of the adjacency matrix \( A \) will be replaced by \( m_{ij} \) (since \( G \) is an undirected graph), call this "new" matrix \( A^* \). Furthermore, the matrix \( D^* \) is just the matrix \( A^* \) with the negative of its row sums inserted on the diagonal and

\[
D^{**} = \begin{pmatrix}
I(L-1)*(L-1) & 0(L-1)*(NV-L+1) \\
\vdots & \vdots \\
0(L-1)*(NV-L+1) & \vdots \\
\end{pmatrix}
\]

the same as those entries of \( D^* \)

Thus in our programme, the edges are read in with their "elasticities" and matrices \( A \) or \( A^* \) are formed automatically.

**Remark 4.1**  If there is a vertex \( V_i \) of degree 1 which is adjacent to \( V_j \) in the given graph, the the coordinate of \( V_i \) will be the same as those of \( V_j \) (Example 5.1).
Remark 4.2  If there is a vertex $V_i$ of degree 2 which is adjacent to $V_j$ and $V_k$ in the given graph, then $V_j$, $V_i$ and $V_k$ are collinear (Example 5.3).

Remark 4.3  *SOURCE* is used as an input file or device name. It is normally the terminal in conversational operation or the card reader in batch operation, and is the source where MTS gets its commands. The file or device which *SOURCE* represents may be changed through use of the SOURCE command. *SOURCE* is one of the MTS pseudo-device names.
5. Results and Examples.

The algorithms described in Sections 3 and 4 have been programmed in FORTRAN for an IBM 360/67 computer. The computer programme identifies a simple connected planar graph $G$ with up to 128 vertices and 400 edges in the present version of GRAPHPACK.

The execution time is divided into two parts, Part I and Part II for the algorithms in Sections 3 and 4 respectively. The execution time for some graphs are given in Table 12 and graphs are in Figures 12 - 18. Some specific examples for previous remarks are given in the following.

<table>
<thead>
<tr>
<th>Number of vertices NV</th>
<th>Number of edges NE</th>
<th>Part I (in seconds)</th>
<th>Part II (in seconds)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12/2</td>
<td>0.13</td>
<td>0.22</td>
<td>Tetrahedron Fig. 12</td>
</tr>
<tr>
<td>8</td>
<td>24/2</td>
<td>0.21</td>
<td>0.24</td>
<td>Cube Fig. 13</td>
</tr>
<tr>
<td>6</td>
<td>24/2</td>
<td>0.21</td>
<td>0.21</td>
<td>Octahedron Fig. 14</td>
</tr>
<tr>
<td>20</td>
<td>60/2</td>
<td>0.54</td>
<td>0.39</td>
<td>Dodecahedron Fig. 15</td>
</tr>
<tr>
<td>12</td>
<td>60/2</td>
<td>0.49</td>
<td>0.31</td>
<td>Icosahedron Fig. 16</td>
</tr>
<tr>
<td>80</td>
<td>240/2</td>
<td>3.70</td>
<td>1.85</td>
<td>&quot;Soccer ball&quot; Figures 17a &amp; 17b</td>
</tr>
<tr>
<td>51</td>
<td>190/2</td>
<td>3.61</td>
<td>1.24</td>
<td>Fig. 18</td>
</tr>
</tbody>
</table>

Table 12
"Graph of Tetrahedron"
"Graph of Cube"

Fig. 13
"Graph of Octahedron"
"Graph of Dodecahedron"

Fig. 15
"Graph of Icosahedron"
"Graph of a Soccer Ball"
"Graph of a Soccer Ball"

Fig. 17b
Remark 5.1  The execution time of Part I depends on the number of iterations performed in finding an outer circuit of a given graph. For example, "the Soccer ball", the execution time for 1, 2 and 3 iterations are 3.70, 5.17 and 8.95 seconds respectively.

Remark 5.2  In Section 4 we pointed out that two different outer circuits for a graph might result in two different representations. Figures 17a and 17b for "the Soccer ball" show that if the outer circuit is a "Pentagon" we get the former, if the outer circuit is a "Hexagon" we get the latter.

Remark 5.3  The execution time for Part II does not include the Plotting time for a graph.

Although the algorithms are intended for use with simple connected planar graphs all of whose vertices are of degree \( \geq 2 \) they also get results of a sort for exceptional cases. Some of these are discussed in this section.

**Treatment of Degenerate Cases.**

In the following examples, all the graphs in Figures - a are given and those in Figures - b are the results we obtained.

**Example 5.1**

![Fig. 19a](image1)

![Fig. 19b](image2)
Since vertex 4 is of degree one, we obtain the coordinates 4 coincide with those of vertex 1. In order to get the given graph, one may assign any coordinates to vertex 4. (Remarks 3.1 and 4.1).

Example 5.2

If the outer circuit is \( C = (i, 2, 3, 4) \), then \((3, 5, 6, 7, 8)\) form a separable bridge of Type 3. In this example, we get the same coordinates for vertices 3, 5, 6, 7, and 8. In fact, one may put \( (5, 6, 7, 8) \) in or out the square \((1, 2, 3, 4)\), then join the vertices 3 and 8 to form the given graph (Remark 3.1 and Definition 4.2; this is an 1-connected graph).

Example 5.3

If the outer circuit is \((1, 2, 3, 4, 5, 6)\), vertices 2, 7, 6 are collinear since vertex 7 is of degree 2.
If the outer circuit is \((1,2,3,4)\), vertices \(2,7,6,8,5,4\) are collinear (Remark 4.2 and Definition 4.2; this is a 2-connected graph).

**Example 5.4**

Clearly, this is a nonplanar graph. If the outer circuit \(C\) is \((1,2,3,4)\), then our programme identifies this graph as nonplanar. For if the bridge formed by the connected component \((5,6,7,8)\) and its vertices of attachment 1, 2, 3, and 4 is in array IN, then the Type 1 bridges \(B_1 = (1,3)\) and \(B_2 = (2,4)\) are in array OUT. Since \(B_1\) alternates with \(B_2\) a nonplanar graph message will be delivered. If the outer circuit is \((5,6,7,8)\), since we did not check if the connected component \((1,2,3,4)\) is planar or not,
no message will be given. The graph will be drawn as it is.

Example 5.5

This is one of Kuratowski's basic nonplanar graphs. If the outer circuit is \((1,2,3,4,5)\), our programme identifies this as a nonplanar graph. If the outer circuit is \((1,2,3)\), since our programme did not check if \(G_i\) are planar or not, the nonplanar message will not be given. Instead it will be considered as a planar graph and the following graph is obtained.
6. How to Use the Programmes.

Our programme can be divided into two parts:

**Part I** The main programme and subroutines of this part of algorithm are in Appendices A, B and C. It finds an outer circuit C for the given graph G.

For example, let us consider G, the "graph of a Tetrahedron", Fig. 10 in Section 4. Then one has to provide the initial and the terminal vertices of all the edges of G along with their "elasticities", i.e.

<table>
<thead>
<tr>
<th>INITIAL</th>
<th>TERMINAL</th>
<th>ELASTICITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

The edges of G.

Termination of G.

A non-positive value for any one of the three values associated with an edge indicates the termination of the data for a graph.

If there are some other graphs to be tested, a positive integer must be followed by using a new record (card) before the edges of a new graph to be read in; otherwise a non-positive integer must be provided.

After the programme in Part I is executed, one will obtain the results on UNIT 6 = *SINK* (Remark 6.1).

In the meantime, the initial and the terminal vertices
with its "elasticity" of each edge, the number of edges (since we are working on undirected graphs, the number of edges obtained is twice those of edges being read in) and the number of vertices of G, number of vertices and their names on the outer circuit C we found will be recorded on UNIT 3. This should be a file if one wishes to draw the graph, or *DUMMY*(Remark 6.2) if not.

Part II The programme is in Appendix D. It will draw a straight-line graph for the given graph G.

The user has to give the data for IFLAG and ISIZE for each graph to be drawn. The latter will give the region ISIZE*ISIZE in which the graph to be plotted [9b]. (In fact, in our programme ISIZE is converted into real, in order to use the Plotting subroutines.)

If IFLAG = 0, the data are to be read from UNIT 3.
If IFLAG ≠ 0, then the user has to provide:
(1) N, the number of vertices of the graph G to be drawn. If N is non-positive, it indicates there are no more graphs to be drawn.
(2) The initial and the terminal vertices of each edge of G with its "elasticity".
(3) L, the number of vertices on the outer circuit C plus 1.
(4) Array LIST, the "name" of vertices on C, one of them is entered twice, i.e. the first and the last ones are the same in array LIST. The order of vertices in LIST must be the same as
the circular order of C in G.

(5) The matrix B, if the vertex V(I) is on C then the user has to assign its coordinates to the I-th row of B, otherwise the entries of I-th row are zero.

For example, if we want to draw the graph \( G \), Fig. 11b, we have to provide the following data:

| No. of Card. | IFLAG & ISIZE | N = No. of vertices.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 : 1 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 : 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 : 1 2 1</td>
<td>................</td>
<td>edges of G and &quot;elasticities&quot;.</td>
</tr>
<tr>
<td>4 : 2 3 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 : 3 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 : 4 1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 : 4 2 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 : 4 3 9</td>
<td>...............</td>
<td>termination of G</td>
</tr>
<tr>
<td>9 : -1 -1 -1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 : 4</td>
<td>L</td>
<td>array LIST</td>
</tr>
<tr>
<td>11 : 1 2 3 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 : 0.0 1.0</td>
<td>................</td>
<td>matrix B</td>
</tr>
<tr>
<td>13 : 0.0 0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 : 1.0 0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 : 0.0 0.0</td>
<td>...............</td>
<td></td>
</tr>
<tr>
<td>16 : -1</td>
<td></td>
<td>the last graph to be drawn.</td>
</tr>
</tbody>
</table>

In our programmes, the input data have the following FORMAT: 3I5 for integers and 2F10.1 for floating point numbers.
Remark 6.1  *SINK* is one of the MTS pseudo-device names. It is used as an output file or device name. It is normally the terminal in conversational mode or the line printer in batch mode. However this designation may be changed through use of the SINK command.

Remark 6.2  *DUMMY* is also one of the MTS pseudo-device names. This pseudo-device may be used anywhere a file is needed for some application. For output, it represents an infinite wastebasket: lines are accepted and they disappear. On input, it acts like an empty file: every time a line is requested an end-of-file condition is returned.
BIBLIOGRAPHY


[9a] UBC SLIN (Moore, C.), Fast Simultaneous Linear Equations and Matrix Inversion Package, UBC Computing Centre, Subject Codes: 45.1, 45.4, March 1970.

[9b] UBC PLOT (Coulthard, W. J.), UBC Plotter Subroutines, UBC Computing Centre, Subject Codes: 08.6, Jan. 1969.

ST GRAPH

1  C
2  C
3  IMPLICIT INTEGER*4(A-Z)
4  REAL TIME1,TIME2,SCLOCK
5  COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
6  COMMON EDG(200),EDI(200),IDEL(400),NE,NV
7  DIMENSION IN(100),OUT(100),TEST(100)
8  DIMENSION NODE(400),ANS(100),LIST(100)
9  DIMENSION PRIME(20)
10  DIMENSION VE1(400),VE2(400),ELAST(400)
11  DIMENSION PROP(400),IN(100),OUT(100)
12  DATA PRIME/2,3,5,7,11,13,17,19,23,29,
13 31,37,41,43,47,53,59,61,67,71/
14  DATA NEG,ONES/1073741824,1073741823/
15  DATA GIS,GIS1/256,255/
16  DATA CIRC,CIRC 1/16777216,16777215/
17  C GIS=2**8=256
18  C CIRC=2**24=16777216
19  C NEG=2**30=1073741824
20  100 FORMAT(10I5)
21  101 FORMAT(1X,25I5)
22  102 FORMAT(1X,10I12)
23  KON=1000
24  210 CALL INITL
25  CALL SCLOCK(0.)
26  DO 220 I=1,100
27  LIST(I)=0
28  NCCE(I)=0
29  ANS(I)=0
30  220 CONTINUE
31  DO 222 I=1,400
32  EPROP(I)=0
33  222 CONTINUE
34  DO 224 I=1,128
35  VPROP(I)=0
36  224 CONTINUE
37  C EACH EDGE IS READ IN BY ITS INITIAL AND TERMINAL VERTICES
38  C WITH ITS "ELASTICITY".
39  I=0
40  WRITE(6,299)
41  299 FORMAT(/'/'' THE EDGES AND THEIR RESP. "ELASTICITY" OF GIVEN
42  $GRAPH ARE',/)
43  200 I=I+1
44  MN=KON+I
45  READ(5,100) VE1(I),VE2(I),ELAST(I)
46  WRITE(6,102) VE1(I),VE2(I),ELAST(I)
47  WRITE(3*MN,101) VE1(I),VE2(I),ELAST(I)
48  C A NON-POSITIVE VALUE FOR ANY ONE OF VE1(I),VE2(I) AND ELAST(I)
49  C INDICATES THE TERMINATION OF THE DATA FOR A GRAPH
50  IF(VE1(I).LE.0.OR.VE2(I).LE.0) GO TO 290
51  CALL NEWE0(VE1(I),VE2(I))
52  GO TO 200
53  290 CONTINUE
54  II=II-1
55  DO 230 J=1,II
CALL NEWEC(VE2(J),VE1(J))

C ********************************************
C FIND A CIRCUIT C
C REALNV= THE NO. OF VERTICES OF G
C REALNE= THE NUMBER OF EDGES OF G

REALNE=NE
REALNV=NV
NXY=0
DO 300 I=1,NV
CALL APATH(I,I,LIST,L,FAIL)
IF(FAIL.EQ.0) GO TO 310
300 CONTINUE
310 CONTINUE
WRITE(6,103)
103 FORMAT('THE VERTICES ON THE CIRCUIT ARE')
WRITE(6,101) (LIST(I),I=1,L)
C DELETE THE VERTICES OF C AND ALL OF THE
C EDGES INCIDENT TO THESE VERTICES
1599 CONTINUE
LL=L-1
DO 400 I=1,LL
CALL DELVE(LIST(I))
400 CONTINUE
C
C GROUP THE SUBGRAPH OF G THAT REMAINS INTO
C A SET OF CONNECTED COMPONENTS
CALL ALPATH(VAL,NGCE,NO,ANS,IX)
IF(IX.LE.1) GO TO 45C
C RESTORE ALL DELETED EDGES AND VERTICES TO THE GRAPH
CALL RESTAL
C ********************************************
C USING VPROP TO FIND THE TYPE OF EACH BRIDGE
C LIST(I,1,L-1) CONSISTS OF THE VERTICES ON THE CIRCUIT
C WHICH MAY BE CONSIDERED AS TYPE 0
C NOTE TYPE 1 ARE CONSIDERED AS TYPE O NCW
DC 500 I=1,IX
IF(ANS(I).EQ.9999) GO TO 500
VPROP(ANS(I))=3
500 CONTINUE
T=NV+1
BC=0
COUNT=0
DC 600 IA=1,IX
IF(ANS(IA).EQ.9999) GO TO 690
I=ANS(IA)
IF(VPROP(I).EQ.-1) GO TO 600
IF(VPROP(I).GE.GIS) GO TO 600
IF(VPROP(I).NE.3) GO TO 600
COUNT=COUNT+1
IF(COUNT.EQ.1) GO TO 620
C ********************************************
C TO FIND TYPE 3 BRIDGES.
C IDENTIFY A CONNECTED COMPONENT WITH A NEW VERTEX T=NV+1
C AND SETTING VPROP(T)=5
630 VPROP(I)=V(NEW)*GIS+VPROP(I)
TT=L-1
J=0
640 CALL NEXEO(I,J)
IF(J.EQ.0) GO TO 680
DC 650 LI=1,TT
116 IF(S(J).NE.LIST(LI)) GC TO 650
118 BC=BC+1
119 EPROP(BC)=LIST(LI)*GIS+3
120 EPROP(BC)=LI*CIRC + EPROP(BC)
121 CALL NEWED(V(NEW),LIST(LI))
122 CALL NEWED(LIST(LI),V(NEW))
123 650 CONTINUE
124 GO TO 640
125 620 NEW=NEWVE(T,0,0)
126 VPROP(NEW)=5
127 NXY=NXY+1
128 C THE HIGHEST ORDER 3 BYTES OF VPROP(NEW) CONTAIN THE NO. OF VERTICES
129 C IN THIS BRIDGE, I.E., THE WEIGHT OF NEW VERTEX.
130 BC=BC+1
131 EPROP(BC)=NEW*GIS+5
132 GO TO 630
133 690 IF(COUNT.EQ.0) GO TO 650
134 VPRCP(NEW)=VPROP(NEW)*COUNT*GIS
135 COUNT=0
136 T=NV+1
137 BC=BC+1
138 EPROP(BC)=-1
139 GO TO 600
140 680 CALL DELVE(I)
141 600 CONTINUE
142 GC TO 820
143 C THE NONDELETED VERTICES FORM P-H GRAPH, I.E.,
144 C THE ABOVE FORMS THE PSEUDO-HAMILTONIAN GRAPH OF THE GIVEN GRAPH.
145 C THE VERTICES BELONG TO TYPE 3 BRIDGES HAVE BEEN DELETED.
146 C ******************************************************
147 C TO FIND TYPE 2 BRIDGES FROM VPROP AND
148 C STORE ATTACHMENT VERTICES IN EPROP.
149 450 CONTINUE
150 C THERE IS NO TYPE 3 BRIDGE
151 CALL RESTAL
152 BC=0
153 820 CONTINUE
154 KEEP=BC
155 LL=L-1
156 DC 800 I=1,REALNV
157 IF(VPROP(I).NE.2) GO TO 800
158 BC=BC+1
159 EPROP(BC)=I*GIS + 2
160 BCC=BC+1
161 J=0
162 840 D=NEXEDG(I,J)
163 IF(C.EQ.0) GO TO 870
164 FLAG=0
165 DO 850 KK=1,LL
166 II=LIST(KK)
167 IF(S(J).EQ.II.OR.P(J).EQ.II) GO TO 880
168 GC TO 850
169 880 BC=BC+1
170 CALL DELED(I,II,0)
171 CALL DELED(II,I,0)
172 EPROP(BC)=II*GIS + 2
173 EPROP(BC)=KK*CIRC + EPROP(BC)
174 FLAG=1
175 GC TO 840
176  850 CONTINUE
177  870 IF(FLAG.EQ.0) GO TO 800
178  DC 810 TEMP=BCC,BC
179  TT=MOD(EPROP(TEMP),CIRC)/GIS
180  ERR1=100C
181  CALL RESTED(I,TT,0,ERR1)
182  ERR2=100C
183  CALL RESTED(TT,I,0,ERR2)
184  C ERRS ARE 0 IF THE RESTORATION IS SUCCESSFUL
185  810 CONTINUE
186  BC=BC+1
187  EPROP(BC)=-1
188  800 CONTINUE
189  IF(BC-l.EQ.KEEP) GO TO 890
190  GO TO 700
191  C *******************************************************
192  C LOOKING FOR TYPE 1 BRIDGES AND STORE EACH PAIR OF
193  C ATTACHMENT VERTICES IN EPRCP.
194  C NOTE THAT WE SHALL DELETE ALL THE EDGES ON THE CIRCUIT
195  C AND TYPE 1 BRIDGES
196  890 EPRCP(BC)=0
197  BC=BC-1
198  C THERE IS NO TYPE 2 BRIDGE
199  700 CONTINUE
200  KEEP1=BC
201  LL=L-1
202  K=1
203  710 KK=K+1
204  I=LIST(K)
205  J=0
206  720 CALL NEXEDO(I,J)
207  IF(J.EQ.0) GO TO 710
208  II=LIST(KK)
209  IF(S(J).NE.II) GO TO 720
210  CALL DELED(I,II,0)
211  CALL DELED(II,I,C)
212  K=KK
213  IF(K.LE.LL) GO TO 710
214  LF=L-2
215  DO 760 K=1,LF
216  I=LIST(K)
217  J=0
218  750 D=NEXEDG(I,J)
219  IF(D.EQ.0) GO TO 760
220  K1=K+1
221  DC 766 KK=K1,LL
222  II=LIST(KK)
223  IF(S(J).EQ.II.OR.P(J).EQ.II) GO TO 780
224  GO TO 766
225  780 CALL DELED(I,II,C)
226  CALL DELED(II,I,C)
227  VPRCP(I)=1
228  WPROP(I)=1
229  BC=BC+1
230  EPRCP(BC)=I*GIS+1
231  EPROP(BC)=K*CIRC+EPRCP(BC)
232  BC=BC+1
233  EPROP(BC)=II*GIS+1
234  EPROP(BC)=KK*CIRC + EPROP(BC)
235  BC=BC+1
EPROP(BC) = -1
CONTINUE
IF(BC-1.EQ.KEEP1) GO TO 785
IF(BC.EQ.0) GO TO 1000
GO TO 799
EPROP(BC) = 0
BC = BC - 1
C THERE IS NO TYPE 1 BRIDGE
CONTINUE
C *******************************************************
C FIND THE ALTERNATE BRIDGES
DO 1111 N = 1, BC
PRCP(N) = EPROP(N)
1111 CONTINUE
CALL ALTER(BC, PROP, LIST, L, IN, OUT, GT, NOBIN, NOBOUT, & 1000)
IF(I.EQ.0) GO TO 1000
IF(GT.EQ.0 .OR. OUT(1).EQ.0) GO TO 1000
IF(NOBOUT.LT.2) GO TO 3000
CALL ERASE(LIST, L)
CALL ALTER(OUT, OUT, LIST, L, IN, OUT, GT, NOBIN, NOBOUT, & 1000)
IF(NOBOUT.EQ.0) GO TO 3000
WRITE(6,3100)
3100 FORMAT(' 'THE GIVEN GRAPH IS NONPLANAR')
L = 1
LIST(L) = 0
GO TO 1000
C *********************************************************
C DELETE ALL BRIDGES IN THE CIRCUIT C
DO 1100 I = 1, I
IF(IN(I) .LT.0 .AND. IN(I+1) .GE. CIRC) GO TO 1110
V1 = IN(I)/GIS
CALL DELVE(V1)
1100 CONTINUE
C *******************************************************
C RESTORE ALL BRIDGES OUTSIDE THE CIRCUIT C
DC 1200 II = 1, OT
IF(OUT(II).LT.0) GO TO 1200
TYPE = MOD(OUT(II), GIS)
IF(TYPE.EQ.5) GO TO 1210
IF(TYPE.EQ.1) GO TO 1220
GO TO 1200
C *******************************************************
C IF THE BRIDGE IS OF TYPE 3, RESTORE ALL THE VERTICES CONTAINED IN
C THIS BRIDGE AND DELETE THE NEW VERTEX
NEW = OUT(II)/GIS
DC 1211 J = 1, REALNV
IF(VPROP(J).LT. GIS) GO TO 1211
V1 = VPROP(J)/GIS
IF(V1.NE.NEW) GO TO 1211
CALL RESTVE(J)
1211 CONTINUE
CALL DELVE(NEW)
GO TO 1200
1220 IF(OUT(J+1).EQ. -1) GO TO 1200
ON1 = OUT(J)/CIRC
ON2 = OUT(J+1)/CIRC
ERR1 = 1000
ERR2 = 1000
CALL RESTED(ON1 , CN2 , 0 , ERR1)
ERR2 = 1000
CALL RESE(ON2 , CN1 , 0 , ERR2)
1200 CONTINUE
CALL ERASE(LIST , L)
C RESTORE ALL EDGES ON THE CIRCUIT
LL = L-1
DO 1250 J = 1 , LL
V1 = LIST(J)
V2 = LIST(J + 1)
ERR1 = 1000
ERR2 = 1000
CALL RESTED(V1 , V2 , 0 , ERR1)
CALL RESTED(V2 , V1 , 0 , ERR2)
1250 CONTINUE
C TO DELETE EDGES ON THE CIRCUIT WHICH ARE COVERED
C BY ANY BRIDGES IN ARRAY OUT
J = 1
FIXJ = J
1260 IF(OUT(J).GE.CIRC) GO TO 1262
BRDGE = OUT(J)/GIS
J = J + 1
1262 FIRST = J
N1 = OUT(FIRST)/CIRC
J = J + 1
1265 IF(OUT(J).NE. -1) GO TO 1265
LAST = J - 1
N2 = OUT(LAST)/CIRC
NN2 = N2 - 1
NN1 = N1 + 1
DO 1270 K = N1 , NN2
V1 = LIST(K)
V2 = LIST(K + 1)
CALL DELED(V1 , V2 , 0)
CALL DELED(V2 , V1 , 0)
1270 CONTINUE
DO 1271 K = NN1 , NN2
V1 = LIST(K)
1272 IF(FIXJ.EQ.FIRST) GO TO 1272
CALL DELED(V1 , BRDGE , 0)
CALL DELED(BRDGE , V1 , 0)
1271 CONTINUE
1300 CONTINUE
DC 1300 K = 1 , NV
CALL APATH(K , K , LIST , L , FAIL)
1359 IF(FAIL .EQ. 0) GO TO 1359
END OF FILE
$LIST GRAPH,S
APPENDIX B
GRAPHPACK

SUBROUTINE INITL
IMPLICIT INTEGER*4(A-Z)
COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
COMMON EDO(200),EDI(200),IDEL(400),NE,NV
MAX=200
M.MAX=400
DO 10 I=1,MAX
   EDC(I)=0
   ECI(I)=0
CONTINUE
DO 15 I=1,MMAX
   PL(I)=0
   SL(I)=0
   ICEL(I)=0
   P(I)=0
   S(I)=0
CONTINUE
DO 20 1=1,128
   V(I)=0
CONTINUE
NE=0
NV=0
RETURN
END

SUBROUTINE NEWED(V1,V2)
INSERT NEW EDGE
IMPLICIT INTEGER*4(A-Z)
COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
COMMON EDO(200),EDI(200),IDEL(400),NE,NV
DATA NEG, ONES/1073741824,1073741823/
MAX=128
LV=0
MV=0
IF(V1.EQ.O) GO TO 50
J=10000
CALL VERNAM(V1,J)
IF(J.EQ.O) IR=NEHVE(V1,LV,MV)
IF(V2.EQ.O) GO TO 50
J=10000
CALL VERNAM(V2,J)
IF(J.EQ.O) IR=NEHVE(V2,LV,MV)
NNE=NE
NE=NE+1
P(NE)=V1
S(NE)=V2
J=NE
IF(NNE.LE.O) GO TO 18
DO 10 I=1,NNE
J = J - 1
IF (P(J) .NE. V1) GO TO 10
SL(J) = NE
GC TO 15
10 CONTINUE
J = NE
DO 20 I = 1, NNE
J = J - 1
IF (S(J) .NE. V2) GO TO 20
PL(J) = NE
GO TO 18
20 CONTINUE
18 IF (V1 .GT. MAX) GO TO 25
IF (LAND(V1), ONES) .NE. V1) GO TO 5
IF (EDO(V1) .NE. 0) GO TO 25
EDO(V1) = NE
25 IF (V2 .GT. MAX) GO TO 40
IF (LAND(V2), ONES) .NE. V2) GO TO 40
IF (EDI(V2) .NE. 0) RETURN
EDI(V2) = NE
RETURN
30 DC 30 I = 1, MAX
IF (LAND(V(I), ONES) .NE. V1) GO TO 30
IF (EDO(I) .EQ. 0) EDO(I) = NE
GO TO 25
30 CONTINUE
GO TO 25
40 DC 45 I = 1, MAX
IF (LAND(V(I), ONES) .NE. V2) GO TO 45
IF (EDI(I) .EQ. 0) EDI(I) = NE
RETURN
45 CONTINUE
RETURN
WRITE(6, 1C0) V1, V2
RETURN
END
FUNCTION NEWVE(VN, V1, V2)
C INSERT NEW VERTEX
IMPLICIT INTEGER*4 (A-Z)
COMMON P(4CO), S(4CO), SL(400), PL(400), V(128), EPROP(400), VPROP(128)
COMMON EDO(200), ECI(2CC), IDEL(400), NE, N
F 100 FORMAT(*' ' ' NUMBER OF VERTICES EXCEEDS', I7)
MAX = 128
NV = NV + 1
IF (NV .GT. MAX) GO TO 27
IF (V(NV) .GT. MAX) GO TO 20
IF (V(NV) .NE. 0) GO TO 20
V(NV) = VN
NEWVE = VN
26 IF (V1 .EQ. 0) GO TO 10
CALL VERNAM(V1, J)
IF (J .EQ. 0) GO TO 10
CALL NEWED(V1, VN)
ECI(NV) = NE
10 IF (V2 .EQ. 0) GO TO 15
CALL VERNAM(V2, J)
IF (J .EQ. 0) GO TO 10
CALL NEWED(VN, V2)
116  EDO(NV)=NE
117  RETURN
118  DO 25 K=1,MAX
119  IF(V(K).NE.0) GO TO 25
120  V(K)=VN
121  NEWVE=K
122  GO TO 26
123  CONTINUE
124  WRITE(6,100) MAX
125  RETURN
126  END
127  C
128  C
129  SUBROUTINE DELED(V1,V2,M)
130  C DELETE EDGE
131  IMPLICIT INTEGERS (A-Z)
132  COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
133  COMMON EDO(200),EDI(200),IDEL(400),NE,NV
134  I=M
135  IF(I.EQ.0) GO TO 5
136  IDEL(I)=-1
137  RETURN
138  5 CC 10 I=1,NE
139  IF(P(I).NE.V1) GO TO 10
140  IF(S(I).NE.V2) GO TO 10
141  ICII(I)=-1
142  RETURN
143  10 CONTINUE
144  DO 11 I=1,NE
145  IF(P(I).NE.V2) GO TO 11
146  IF(S(I).NE.V1) GO TO 11
147  IDEL(I)=-1
148  RETURN
149  11 CONTINUE
150  C EDGE DEFINED BY V1 AND V2 DOES NOT EXIST.
151  RETURN
152  ENC
153  C
154  C
155  SUBROUTINE DELVE(IV)
156  C DELETE VERTEX
157  IMPLICIT INTEGERS*4(A-Z)
158  COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
159  COMMON EDO(200),EDI(200),IDEL(400),NE,NV
160  DATA NEG, ONES/1073741824,1073741823/
161  100 FORMAT(//" ' VERTEX',I5,3X,'TO BE DELETED DOES NOT EXIST'\)
162  CALL VERNAM(IV,JV)
163  IF(JV.EQ.0) GO TO 50
164  IF(V(JV).GE.NEG) RETURN
165  MK=0
166  V(JV)=V(JV)+NEG
167  J=EDO(JV)
168  15 IF(J.EQ.0)GO TO 10
169  IF-IDEL(J).EQ.-1)GO TO 10
170  IDEL(J)=1
171  K=SL(J)
172  IF(MK.EQ.1) K=PL(J)
173  IF(K.EQ.0)GO TO 10
174  J=K
175  GO TO 15
SUBROUTINE RESTED(VR, VS, M, ERR)
RESTORE EDGE
IMPLICIT INTEGER*4(A-Z)
COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)
COMMON EDO(200), EDI(200), IDEL(400), NE, NV
DATA NEG, ONES/1073741824, 1073741823/
I = M
IF(I.EQ.0) GO TO 7
V1 = P(I)
V2 = S(I)
GO TO 5
7
V1 = VR
V2 = VS
5
CALL VERNAM(V1, J)
IF(J.EQ.0) GO TO 40
IF(V(J).GE.NEG) GO TO 20
CALL VERNAM(V2, J)
IF(J.EQ.0) GO TO 40
IF(V(J).GE.NEG) GO TO 20
IF(I.EQ.0) GO TO 6
IDEL(I) = 0
40
ERR = 0
RETURN
6
DC 30 I=1,NE
IF(P(I).NE.V1) GO TO 30
IF(S(I).NE.V2) GO TO 30
IDEL(I) = 0
30
GO TO 40
30
CONTINUE
215
DO 50 I = 1, NE
216
IF(P(I).NE.V1) GO TO 50
217
IF(S(I).NE.V2) GO TO 50
218
IDEL(I) = 0
219
GO TO 40
220
50
CONTINUE
221
ERR = 0
222
20
IF(ERR.EQ.1000) RETURN
ERR = LAND(V(J), ONES)
224
WRITE(6,100) V1, V2, ERR
225
100
FORMAT('EDGE*215,3X,CANNOT BE RESTORED',15,3X,'IS DELETED'
226
RETURN
227
END

SUBROUTINE RESTVE(IV)
RESTORE VERTEX
IMPLICIT INTEGER*4(A-Z)
COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)
COMMON ECC(200), EDI(200), IDEL(400), NE, NV
DATA NEG, ONES/1073741824, 1073741823/
SUBROUTINE RESTAL
  C RESTORE ALL DELETED VERTICES AND EDGES
  IMPLICIT INTEGER*4(A-Z)
  COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
  COMMON EDO(200),EDI(200),IDEL(400),NE,NV
  EXT=255
  DO 10 I=1,NE
  10 ICEN(I)=0
  DC 20 I=1,NV
  IV=LAND(V(I),EXT)
  V(I)=IV
  CONTINUE
  RETURN
END
C
SUBROUTINE NEXEDC(I,KPC)
  C FIND NEXT EDGE OUT OF VERTEX
  IMPLICIT INTEGER*4(A-Z)
  COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
  COMMON EDO(200),EDI(200),IDEL(400),NE,NV
  J=EDO(I)
  IF(J.EQ.0) GO TO 10
  15 IF(KPD.NE.0) J=SL(KPC)
  10 KPD=0
  IF(J.EQ.0) RETURN
  KPC=J
  IF(IDEL(J).NE.0) GO TO 15
  RETURN
END
C
SUBROUTINE NEXEDI(I,KPC)
C FIND NEXT EDGE INTO VERTEX

IMPLICIT INTEGER*4(A-Z)

COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)

COMMON EDG(200), EDI(200), IDEL(400), NE, NV

J = EDI(I)

IF(J.EQ.0) GO TO 10

IF(KPD.NE.0) J = PL(KPD)

KPC = 0

IF(J.EQ.0) RETURN

KPD = J

IF,IDEL(J).NE.0) GO TO 15

RETURN

END

FUNCTION NEXEDG(IV,K)

C TO FIND NEXT EDGE FROM VERTEX IV FOR AN UNDIRECTED GRAPH.

IMPLICIT INTEGER*4(A-Z)

COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)

COMMON EDG(200), EDI(200), IDEL(400), NE, NV

100 FORMAT(//' ', ' VERTEX INTO NEXTEDGE IS ZERO')

IF(IV.EQ.0) GO TO 20

IF(K.EQ.0) GO TO 10

IF(P(K).NE.IV) GO TO 15

CALL NEXEDO(IV,K)

NEXEDG = 1

RETURN

CALL NEXEDI(IV,K)

NEXEDG = 0

IF(K.NE.0) GO TO 16

RETURN

NEXEDG = -1

RETURN

WRITE(6,100)

RETURN

END

C

SUBROUTINE VERNAM(IV,JV)

IMPLICIT INTEGER*4(A-Z)

COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)

COMMON EDG(200), EDI(200), IDEL(400), NE, NV

DATA NEG, ONES/1C73741824,1073741823/

EXT = 255

MAX = 128

IF(IV.LE.MAX) GO TO 20

DO 10 I = 1, MAX

M = LAND(IV(I), ONES)

IF(M.EQ.0) GO TO 11

M = LAND(M, EXT)

IF(IV.NE.M) GO TO 10

JV = I

RETURN

CONTINUE

10 CONTINUE

11 IF(JV.EQ.10000) GO TO 12

WRITE(6,102) IV

12 JV = 0

RETURN

102 FORMAT(//' ', ' VERTEX CALLED', I5, 3X, 'HAS NOT BEEN INSERTED')
356 20 IF(IV.NE.LAND(V(IV),EXT)) GO TO 21
357 JV=IV
358 RETURN
359 END

END OF FILE

$COMMENT '"'SKIP TO TOP OF NEXT PAGE''
$COPY *SKIP
SUBROUTINE APATH(I,END,LIST,L,FAIL)
IMPLICIT INTEGER*4(A-Z)
COMMON P(400),S(400),SL(400),PL(400),V(128),ED(400),VPROP(128)
COMMON ED0(200),ECI(200),IDEL(400),NE,NV
DIMENSION LIST(1C0)
DATA NEG,ONES/1073741824,1073741823/
DATA GIS,GIS1/256,255/
C I=THE INITIAL VERTEX
C END=THE TERMINAL VERTEX
C LIST=STORING VERTICES IN THE PATH
C L=THE LENGTH OF THE LIST
C FAIL=1 IF NO PATH CAN BE FOUND, OTHERWISE 0
DO 100 T=1,NE
100 EC(T)=I
L=0
FAIL=1
CALL VERNAM(I,KK)
IF(KK.EQ.0) RETURN
IF(V(KK).GT.NEG) GO TO 500
CONTINUE
IF(V(KK).GT.GIS) GO TO 150
V(KK)=GIS+V(KK)
L=L+1
LIST(L)=V(KK)-GIS
K=LIST(L)
200 J=0
220 CALL NEXEDO(K,J)
IF(J.EQ.G) GO TO 550
IF(IDEL(J).NE.0) GO TO 220
IF(ED(J).NE.1) GO TO 220
ED(J)=ED(J)+1
II=S(J)
CALL VERNAM(II,KK)
IF(KK.EQ.0) GO TO 220
IF(KK.EQ.END) GO TO 600
CALL DELEC(KK,K,0)
GO TO 140
500 CONTINUE
C VERTEX I HAS BEEN DELETED
RETURN
550 CONTINUE
C THERE IS NO PATH BETWEEN VERTICES I AND END
RETURN
600 FAIL=0
L=L+1
LIST(L)=LAND(V(KK),GIS1)
C CLEAR ALL FLAGS
DO 700 T=1,NV
700 V(T)=LAND(V(T),GIS1)
DO 800 T=1,NE
800 EC(T)=0
RETURN
END
SUBROUTINE PATH(I,NCDE,NC,ANS,IX)  
IMPLICIT INTEGER*4(A-Z)  
COMMON P(400),S(400),SL(400),PL(400),V(128),ED(400),VPROP(128)  
COMMON EDC(200),EDI(200),IDEL(400),NE,ND  
DIMENSION NODE(400),ANS(100)  
DATA NEG,ONES/1073741824,1073741823/  
DATA GIS,GIS1/256,255/  
C I=STORE GIVEN VERTEX  
C STORE A VERTEX WHENEVER IT IS TESTED  
C NO=NO. OF VERTICES IN ARRAY NODE  
C ED=STORE ALL EDGES IN ARRAY ED WHICH ARE WHITE(=1) TO START WITH  
C ANS=ALL THE DIFFERENT VERTICES ARE STORED IN ARRAY ANS ALONG PATH  
C IX=NO. OF ELEMENTS IN ANS  
IX=IX+1  
NC=1  
FFIX=IX  
CALL VERNAM(I,KK)  
IF(KK.EQ.0) RETURN  
C STORE ANY TIME A VERTEX IS REACHED  
FIXNO=NO  
K=V(KK)  
NODE(NO)=K  
NC=NC+1  
NEXNO=NO  
C PAINT VERTEX K RED(1*GIS) IF IT IS GREEN(VERTEX ITSELF) IN ARRAY V  
IF(V(KK).GT.GIS) GO TO 110  
100 V(KK)=GIS+V(KK)  
110 ANS(IX)=V(KK)-GIS  
IX=IX+1  
K=LAND(V(KK),GIS1)  
FIXK=K  
FIXNO=NO  
200 J=0  
220 CALL NEXEDO(K,J)  
IF(J.EQ.0) GO TO 360  
C FIND WHITE EDGE  
IF(ED(J).NE.0) GO TO 220  
IF(V(VJ).GT.NEG) GO TO 390  
300 IF(ED(J).NE.1) GC TO 220  
C FOLLOW THE WHITE EDGE, COLOUR GRAY(=2) AND DELETE THE OPPOSITE SIDE  
ED(J)=ED(J)+1  
I=S(J)  
CALL VERNAM(I,KK)  
IF(KK.EQ.0) GO TO 220  
TEMP=V(KK)  
IF(TEMP.GT.NEG) GO TO 220  
IF(TEMP.GT.GIS) TEMP=V(KK)-GIS  
K=TEMP  
CALL DELED(K,FIXK,0)  
FIXK=K  
NCDE(NO)=K  
NO=NO+1  
NEXNO=NO  
C WHAT COLOR VERTEX?  
IF(V(KK).EQ.K) GO TO 100  
GO TO 220  
C IF THE VERTEX COLOUR IS REC, TRY FOR NEXT WHITE EDGE, IF NOT
SUBROUTINE ALPATH(VAL, NODE, NO, ANS, IX)

IMPLICIT INTEGER*4(A-Z)

COMMON P(400), S(400), SL(400), PL(400), V(128), ED(400), VPROP(128)

COMMON ED0(200), ED1(200), IDEL(400), NE, NV

DIMENSION NODE(400), ANS(100)

DATA NEG, GNES /1073741824, 1073741823/

DATA GIS, GISs /256, 255/

C VAL=NO. OF PATHS IN ARRAY ANS

C ALL VERTICES ARE GREEN(=1) TO START WITH

C I.E., THE NAME OF EACH VERTEX IS IN ARRAY V

C ALL EDGES ARE WHITE(=1) TO START WITH

DO 110 K=1, NE

ED(K)=1

VAL=0

IX=0

DO 200 J=1, NV

KPD=0

I=V(J)

IF(VPROP(J).EQ.-1) GO TO 200

IF(I.GE.NEG) GO TO 280

IF(I.GT.GIS) GO TO 200

CALL NEXEDO(I, KPD)

IF(KPD.EQ.0) GO TO 250

IF(IDEL(KPD).EQ.0) GO TO 210

V1=S(KPD)

IF(V(V1).GT.NEG) GO TO 270

FIX=IX+1

CALL PATH(I, NODE, NO, ANS, IX)

IF(FIX.LE.IX) VAL=VAL+1

GC TO 200

200 GCNTINUE

IF(VAL.GT.0) GO TO 400

C THERE IS NO PATH IN THIS PART OF GRAPH

250 VPROP(J)=2

C VERTEX J IS OF TYPE 2

GC TO 200

270 CALL DELED(0, 0, KPD)

GO TO 210

280 VPROP(J)=0

200 CONTINUE

400 CONTINUE

C CLEAR ARRAY ED BEFORE RETURN

DO 410 J=1, NE

ED(J)=0

410 RETURN

END
SUBROUTINE PAINT(LIST, L, SPP, PP, SI, IN, I)
C PAINT THE VERTICES ON C BY SMALL PRIME NOS., IF THE VERTICES ARE
C ON ARRAY IN THEN PAINT TWICE
C I=THE LAST INDEX OF ARRAY IN, LI=I-1
C SI=STARTING INDEX OF ARRAY IN
IMPLICIT INTEGER*4(A-Z)
COMMON P(400), S(400), SL(400), PL(400), V(128), EPROP(400), VPROP(128)
COMMON EDG(200), EDI(200), IDEL(400), NE, NV
DIMENSION IN(100), LIST(100), PRIME(20)
DATA PRIME/2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
$ 31, 37, 41, 43, 47, 53, 59, 61, 67, 71/
DATA GIS, GIS1/256, 255/
DATA CIRC, CIRC1/16777216, 16777215/
PP=SPP
XI=SI
IF(IN(SI).LT.CIRC) XI=51+1
LL=L-1
LI=I-1
KK=XI
FLAG=0
DO 330 K=1, LL
Z=IN(KK)/CIRC
IF(K.EQ.Z) GO TO 340
GO TO 330
340 FLAG=FLAG+1
IF(FLAG.GT.1) GO TO 355
FK=K
FIXK=K
KK=KK+1
IF(KK.GT.LI) GO TO 380
GO TO 330
355 IF(FLAG.GT.2) GO TO 350
DO 370 K1=FIXK, K
LIST(K1)=PRIME(PP)*GIS+LIST(K1)
370 CONTINUE
PP=PP+1
FIXK=K
GO TO 360
350 CONTINUE
TEMP=LAND(LIST(FIXK), GIS1)
LIST(FIXK)=(LIST(FIXK)/GIS)*PRIME(PP)*GIS+TEMP
K2=FIXK+1
DO 375 K1=FIXK, K
LIST(K1)=PRIME(PP)*GIS+LIST(K1)
375 CONTINUE
PP=PP+1
FIXK=K
GO TO 360
360 CONTINUE
TEMP=LAND(LIST(K), GIS1)
LIST(K)=(LIST(K)/GIS)*PRIME(PP)*GIS+TEMP
IF(K.EQ.LL) GO TO 386
K2=K+1
DO 390 K1=K2, LL
LIST(K1)=PRIME(PP)*GIS+LIST(K1)
390 CONTINUE
386 IF(FK.LE.1) GO TO 399
FFK=FK-1
DC 387 K1=1,FKK
LIST(K1)=PRIME(PP)*GIS+LIST(K1)
CONTINUE
399 TEMP=LAND(LIST(FK),GIS1)
LIST(FK)=(LIST(FK)/GIS)*PRIME(PP)*GIS+TEMP
RETURN
END

SUBROUTINE REPANT(LIST,L,SPP,PP,SI,IN,I)
PAINT THE CIRCUIT AGAIN BY USING ARRAY IN(SI) TO IN(I)
IMPLICIT INTEGER*4(A-Z)
COMMON P(4G0),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
COMMON ED(200),ED1(200),IDE(400),NE,NV
DIMENSION IN(100),LIST(100),PRIME(20)
DATA PRIME/2,3,5,7,11,13,17,19,23,29,
$ 31,37,41,43,47,53,59,61,67,71/
DATA GIS,GIS1/256,255/
DATA CIRC,CIRCl/16777216,16777215/
PP=SPP
XI=SI
IF(IN(SI).LT.CIRC) XI=SI+1
LL=L-1
LI=I-1
KK=XI
X=IN(KK)/CIRC
I=FIXI=X
NI=X+1
C IF FIXI=LL, START FROM THE FIRST VERTEX IN ARRAY LIST
IF(FIXI.EQ.LL) RETURN
COL1=LIST(FIXI)/GIS
CCL2=LIST(NI)/GIS
F12=HCF(COL1,COL2)
Q1=1 OR > 1
Q1=COL1/F12
IF(Q1.EQ.1) GO TO 500
C CASE 1: LIST(FIXI)/(GIS+F12) > 1
C REPAINT NEXT INTERSECTION BY PRIME(PP), ERASE F12 FOR THE FOLLOWING
C VERTICES TILL IT REACHES LIST(?)/(GIS+F12) > 1
KK=KK+1
IF(KK.GT.LI) RETURN
Y=IN(KK)/CIRC
IF(KK.EQ.LI) GO TO 370
KK=KK+1
IF(KK.GT.LI) RETURN
Z=IN(KK)/CIRC
C 2ND AND 3RD ARE NOT THE LAST ONE IN ARRAY IN
TEMPY=LAND(LIST(Y),GIS1)
CCLY=LIST(Y)/GIS
LIST(Y)=CCLY*PRIME(PP)*GIS+TEMPY
K1=Y+1
IF(KK.EQ.LI) GO TO 365
DO 350 K=K1,Z
TEMPK=LAND(LIST(K),GIS1)
CCLK=LIST(K)/GIS
QK=CCLK/F12
LIST(K)=QK*GIS*PRIME(PP)+TEMPK
296 350 CCONTINUE
297 Y=Z
298 PP=PP+1
299 GC TO 345
300 365 Y=Z
301 C 2ND VERTEX IS THE LAST CNE IN ARRAY IN
302 370 COLY=LIST(Y)/GIS
303 QY=COLY/F12
304 IF(QY.EQ.1) GO TO 390
305 C LIST(Y) IS THE LAST CNE AND IS OF MULTIPLE COLOURS
306 DIFF=1-XI
307 C IF THE BRIDGE HAS ONLY TWO ATTACHMENT VERTICES AND BOTH HAVE
308 C MULTIPLE COLOURS THAN RETURN
309 IF(DIFF.EQ.2) RETURN
310 TEMPY=LAND(LIST(Y),GIS1)
311 LIST(Y)=CY*PRIME(PP)*GIS+TEMPY
312 GO TO 600
313 390 TEMPY=LAND(LIST(Y),GIS1)
314 LIST(Y)=PRIME(PP)*PRIME(PP+1)*GIS+TEMPY
315 PP=PP+1
316 IF(Y.EQ.LL) GO TO 430
317 YY=Y+1
318 410 DO 420 K=YY,LL
319 COLK=LIST(K)/GIS
320 QK=COLK/F12
321 IF(QK.GT.1) GO TO 440
322 TEMPK=LAND(LIST(K),GIS1)
323 LIST(K)=QK*PRIME(PP)*GIS+TEMPK
324 420 CCONTINUE
325 430 YY=1
326 GO TO 410
327 440 TEMPK=LAND(LIST(K),GIS1)
328 LIST(K)=CK*PRIME(PP)*GIS+TEMPK
329 GO TO 600
330 C
331 C CASE 2: LIST(FIS1)/(GIS*F12) = 1
332 C REPAINT LIST(FIXI) BY PRIME(PP) AND ERRASE F12 FOR THE FOLLOWING
333 C VERTICES TILL IT REACHES LIST(?)/(GIS*F12) > 1
334 500 CCONTINUE
335 Y=X
336 GO TO 345
337 600 RETURN
338 ENC
339 C
340 C
341 C SUBROUTINE ERASE(LIST,L)
342 C ERASE THE PAINTED VERTICES ON THE CIRCUIT
343 IMPLICIT INTEGER*4(A-Z)
344 COMMON P(400),S(400),SL(400),PL(400),V(128),EPROP(400),VPROP(128)
345 COMMON EDO(200),EDI(200),IDEL(400),NE,NV
346 DIMENSION LIST(100)
347 DATA GIS,GIS1/256,255/
348 LL=L-1
349 DO 200 K=1,LL
350 LIST(K)=LAND(LIST(K),GIS1)
351 200 CCONTINUE
352 RETURN
353 END
354 C
355 C
FUNCTION HCF(A,B)  
C TO FIND HCF OF A AND B  
IMPLICIT INTEGER*4(A-Z)  
COMMON P(400),S(400),SL(400),PL(400),V(128),EPRGP(400),VPROP(128)  
COMMON EDO(200),EDI(200),IDEL(400),NE,NV  
Y=A  
Y1=B  
TEMP=MOD(Y,Y1)  
IF(TEMP.EQ.0) GO TO 200  
IF(TEMP.EQ.1) GO TO 300  
Y=Y1  
Y1=TEMP  
GO TO 150  
200 HCF=Y1  
RETURN  
300 HCF=1  
RETURN  
END  
C  
SUBROUTINE SAME(IN,I)  
C ELIMINATE THE SAME ELEMENTS IN ARRAY IN  
IMPLICIT INTEGER*4(A-Z)  
COMMON P(400),S(400),SL(400),PL(400),V(128),EPRGP(400),VPROP(128)  
COMMON EDO(200),EDI(200),IDEL(400),NE,NV  
DIMENSION IN(100)  
KI=1  
914 IF(IN(KI).NE.IN(KI+1)) GO TO 916  
917 I=I-1  
915 IN(KKI)=IN(KKI+1)  
916 CONTINUE  
917 IF(IN(KI).EQ.IN(KI+1)) GO TO 917  
918 KI=KI+1  
919 IF(KI.LE.1) GO TO 914  
RETURN  
END  
C  
SUBROUTINE ORDER(IN,I)  
C REARRANGE THE ORDER OF ARRY IN IN THE SENSE OF C BEING OBTAINED  
IMPLICIT INTEGER*4(A-Z)  
COMMON P(400),S(400),SL(400),PL(400),V(128),EPRGP(400),VPROP(128)  
COMMON EDO(200),EDI(200),IDEL(400),NE,NV  
DIMENSION IN(100)  
DATA CIRC,CIRC1/16777216,16777215/  
ST=1  
919 IF(KK.LE.0) RETURN  
918 DO 918 K=ST,KK  
917 X=IN(K)/CIRC  
916 Y=IN(K+1)/CIRC  
915 IF(X.LT.Y) GO TO 918  
914 SWAP=IN(K+1)  
913 IN(K+1)=IN(K)  
912 IN(K)=SWAP  
911 CONTINUE  
910 KK=KK-1  
909 IF(KK.GT.1) GO TO 919
SUBROUTINE ALTNAT(LI, ST, SI, IN, I, TEST, TEST, TE, SO, OUT, OT, NOBIN, NOBOUT)

C I=THE LAST INDEX OF ARRAY IN, IT=I-1
C SI=STARTING INDEX OF ARRAY IN
C TEST=STARTING INDEX OF ARRAY TEST
C TE=THE LAST INDEX OF ARRAY TEST, TEE=TE-1
C SO=(STARTING INDEX -1) OF ARRAY OUT
C OT=THE LAST INDEX OF ARRAY OUT
C TO CHECK IF TEST ALTERNATES WITH BRIDGES IN ARRAY IN, IF YES
C STORE IN ARRAY OUT, OTHERWISE IN ARRAY IN

IMPLICIT INTEGER*4(A-Z)
COMMON P(4CC),S(4CC),SL(4CG),PL(40C),V(128),EPROP(400),VPROP(128)
COMMON EDO(200),EDI(200),IDEL(400),NE,NV
DIMENSION IN(100),CUT(100),TEST(100),L1ST(100)
DATA GIS,GIS1/256,255/
DATA CIRC,CIRC1/16777216,16777215/

C
Q=SO
T=TE-1
IF(TE.LE.2) RETURN
XT=TEST
IF(TEST(STEST).LT.CIRC) XT=TEST+1
W=NO. ATTACHMENT VERTICES ON C
W=TE-XT

C THE FIRST ATTACHMENT VERTEX
X=TEST(XT)/CIRC
Y=LIST(X)/GIS

C THE LAST ATTACHMENT VERTEX
LX=TEST(TEE)/CIRC
LY=LIST(LX)/GIS
IF(Y.EQ.LY.AND.W.EQ.2) GO TO 975
H=HCF(Y,LY)
IF(H.GT.1.AND.W.EQ.2) GO TO 975
IFIY.EQ.LY.OR.H.GT.1) GO TO 400
GO TO 985

400 NT=XT
401 NT=NT+1
IF(NT.GE.TEE) GO TO 410
X1=TEST(NT)/CIRC
Y1=LIST(X1)/GIS
IF(Y.EQ.Y1) GO TO 405
H1=HCF(Y,Y1)
IF(H1.EQ.1) GO TO 985

405 Y=Y1
GO TO 401
410 IF(Y1.EQ.LY) GO TO 975
HL=HCF(Y1,LY)
IF(HL.GT.1) GO TO 975
GO TO 985

C THESE TWO BRIDGES DO NOT ALTERNATE

975 DC 970 NOT=TEST,TE
970 I=I+1
IN(I)=TEST(NOT)
970 CONTINUE
NCBIN=NOBIN+1
RETURN
THESE TWO BRIDGES ALTERNATE WITH EACH OTHER

CONTINUE

DC 980 YES=TEST,TE
OT=OT+1
OUT(OT)=TEST(YES)
CONTINUE
NCBOUT=NCBOUT+1
RETURN
END

C

SUBROUTINE ALTER(BC,PROP,LIST,L,IN,I,OUT,OT,NBIN,NBOUT,*)
FINC THE ALTERNATE BRIDGES
C I=INDEX FOR IN, IT=I-1
C J=INDEX FOR EPROP
C TE=INDEX FOR TEST, T£E=TE-1
C OT=INDEX FOR OUT
C NOBIN=NO. OF BRIDGES IN ARRAY IN
C NOBOUT=NO. OF BRIDGES IN ARRAY OUT
IMPLICIT INTEGERS!A-Z)
COMMON P(400),S(400),SL(400),PL(40C),V(128),EPROP(400),VPROP(128)
COMMON EDI(200),EDI(200),IDEI(400),NE,NV
DIMENSION IN(100),OUT(100),TEST(10C),LIST(10C)
DIMENSION PROP(400)
DATA GIS,GIS1/256,255/
DATA CIRC,CIRC 1/16777216,16777215/
DATA CIRC,CIRC1/16777216,16777215/
C
DC 200 Z=1,100
IN(Z)=0
OUT(Z)=0
TEST(Z)=0
CONTINUE
SO=0
IF(BC.LE.0) RETURN 1
II=1
GO TO 902
901 XI=IN(1)/GIS
CALL DELVE(XI)
902 I=0
SI=I+1
DO 900 J=II,BC
I=I+1
IF(_PROP(J).LT.0) GO TO 910
IN(I)=PROP(J)
900 CONTINUE
910 IN(I)=-1
NCBIN=1
IF(I.EQ.BC) RETURN 1
FIXJ=J
C REARRANGE THE ORDER OF ARRY IN IN THE SENSE WE OBTAIN C
CALL ORDER(IN,I)
C ELIMINATE THE SAME VERTICES IN IN(I)
CALL SAME(IN,I)
SPP=1
NOBOUT=0
II=FIXI+1
C SEPARABLE BRIDGES ARE IGNORED AND DELETED
IF(IN(I).LT.CIRC.AND.I.EQ.3) GO TO 901
CALL PAINT(LIST,L,SPF,FP,SI,IN,I)
C PANTI= THE INDEX OF LAST BRIDGE IN ARRAY IN HAD BEEN PAINTED
536       PANTI=I
537   590       CONTINUE
538   3980       TE=1
539       STEST=TE
540       DG 920       J=II, BC
541       IF(PROP(J).LT.0) GO TO 930
542       TEST(TE)=PROP(J)
543       TE=TE+1
544   920       CONTINUE
545   930       IJ=J+1
546       TEST(TE)=-1
547       'C REARRANGE THE ORDER OF ARRAY TEST IN THE SENSE WE OBTAIN C
548       CALL ORDER(TEST, TE)
549       C ELIMINATE THE SAME VERTICES IN TEST(TE)
550       CALL SAME(TEST, TE)
551       C SEPARABLE BRIDGES ARE IGNORED AND DELETED
552       IF(TEST(1).LT.CIRC.AND.TE.EQ.3) GO TO 981
553       NC=NOBOUT
554       SI=1
555   980       CALL ALTNAT(LIST, L, SI, IN, I, TEST, TEST, TE, SO, OUT, OT, NOBIN, NOBCUT)
556       C SO=(NEXT STARTING INDEX - 1) OF OUT
557       SC=CT
558       IF(JJ.JT.BC) GO TO 999
559       IF(NOB.LT.NOBOUt) GO TO 400
560       IF(NOBIN.LT.2) GO TO 400
561       SPP=PP+1
562       C NEXT BRIDGE IN ARRAY IN TO BE REPAINTED: FROM IN(SI) TO IN(LASTI)
563   2200       SI=PANTI+1
564   2430       CALL REPANT(LIST, L, SPP, PP, SI, IN, I)
565       PANTI=I
566   400       GO TO 2981
567   981       XI=TEST(1)/G I S
568       CALL DELVE(XI)
569   2981       II=J+1
570       IF(IJ.JT.BC) GO TO 999
571       DG 2902       IS=1, TE
572       TEST(IS)=0
573   2902       CONTINUE
574   999       RETURN
575       END
576       END OF FILE
577
578 /*COMMENT ' 'SKIP TO TOP OF NEXT PAGE''
579 $COPY *SKIP
READ(3,99) N \* KON+J
DO 200 J=1,N
\* N=5, NE=2
\* C TO FORM THE ADJACENCY MATRIX A OR A+
IF(N=.LT.1) GO TO 70
10 FORMAT(/110") N,N=5, NE=2, I5
\* N=5, NE=2
\* C CONTINUE
************************************
39 C CALL PLOTS
38 C CONTINUE
37 C CALL PLOTS
36 C CONTINUE
35 C \* IFLAG=0, IFLAG=12
34 DO 99 J=1,128
33 DO 99 K=1,128
32 C IFLAG=0, IFLAG=12
31 C IFLAG AND ISIZE WILL BE SET BY THE USER
30 C IFLAG=0, IFLAG=12
29 C IFLAG=0, IFLAG=12
28 \* \* C TO SOLVE C*X = E
27 FLAG=TRUE
26 FORMATT(X*1012)
25 FORMATT(X*2515)
24 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
23 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
22 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
21 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
20 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
19 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
18 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
17 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
16 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
15 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
14 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
13 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
12 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
11 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
10 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
9 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
8 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
7 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
6 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
5 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
4 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
3 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
2 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
1 \* \* SOLUTION OF LINEAR EQUATIONS FAILS!
116  GC TO 14
117  19 A(K,J)=1.
118  14 CONTINUE
119  LM=LM+1
120  GC TO 18
121  15 CONTINUE
122  IF(IFLAG.NE.0) GO TO 500
123  C **************************************************************
124  DC 12 K=1,N
125  DO 12 J=1,NSOL
126  B(K,J)=0.
127  12 CONTINUE
128  C THE OUTER CIRCUIT WILL BE PUT IN THE FIRST QUADRANT OF XY-PLANE
129  C AS A (L-1) REGULAR POLYGON
130  T1=TWOPI/FLOAT(L)
131  DC 600 K=1,L
132  T2=T1*FLOAT(K)
133  XX(LIST(K))=CONST*COS(T2)
134  XX(LIST(K))=XX(LIST(K))+CONST
135  YY(LIST(K))=CONST*SIN(T2)
136  YY(LIST(K))=YY(LIST(K))+CONST
137  600 CONTINUE
138  GO TO 610
139  C **************************************************************
140  C THE OUTER CIRCUIT MAY BE ASSIGNED BY THE USER ANYWHERE HE WISHES.
141  C THE REGION IS ISIZE*ISIZE CR SIZE*SIZE
142  500 CONTINUE
143  DC 550 K=1,N
144  READ(5,20) (B(K,J),J=1,NSOL)
145  WRITE(6,50) (B(K,J),J=1,NSOL)
146  550 CONTINUE
147  610 CONTINUE
148  CALL FSLE(N,LENA,A,NSOL,LENBX,B,Z,IPERM,LENT,T,DET,TEXP)
149  IF(DET) 25,30,25
150  25 CONTINUE
151  26 FORMAT(//' ' SOLUTION Z IS'
152  DC 27 K=1,N
153  WRITE(6,50) (Z(K,J),J=1,NSOL)
154  27 CONTINUE
155  CALL SCALE(X,N,SIZE,XMIN,DX,1)
156  CALL SCALE(Y,N,SIZE,YMIN,DY,1)
157  CALL AXIS(0.,0.,6HX AXIS,-6,SIZE,0.,XMIN,DX)
158  CALL AXIS(0.,0.,6HY AXIS,6,SIZE,90.,YMIN,DX)
159  DC 80 K=1,N
160  CALL PLOT(X(K),Y(K),+3)
161  IF(K.GE.N) GO TO 81
162  JJ=K+1
163  DC 90 J=JJ,N
164  IF(AA(K,J).EQ.0.) GO TO 90
165  C IF THERE IS AN EDGE BETWEEN (X(K),Y(K)) AND (X(J),Y(J)), THEN DRAW
166  C A LINE AND MOVE BACK TO (X(K),Y(K)) WITH PEN UP
167  CALL PLOT(X(J),Y(J),+2)
168  CALL PLOT(X(K),Y(K),+3)
169  90 CONTINUE
170  80 CONTINUE
171  81 CONTINUE
172  CALL PLOT(12.0,0.0,-3)
173  800 CONTINUE
174  DC 700 K=1,N
DC 700 J=1,N
AA(K,J)=0.
IF(IFLAG.NE.0) GC TO 701
LI=KON+499
READ(3'LI,102) II
GC TO 702
READ(5,100) II
CONTINUE
KCN=KON+500
TIME2=SCLOCK(TIME1)
WRITE(6,4999) TIME2
FORMAT(//', ' THE EXECUTION TIME =', F6.2, ' SECONDS')
IF(II.LE.0) GO TO 70
GO TO 1
WRITE(6,60)
GC TO 800
CONTINUE
WRITE(6,98)
FORMAT(//', ' THE OUTER CIRCUIT CAN NOT BE FOUND, PLOT ROUTINES $WILL NOT BE CALLED')
GO TO 800
CONTINUE
CALL PLOTND
STOP
END