M.TECH. (COMPUTER SCIENCE) DISSERTATION SERIES

RECTILINEAR LAYOUTING OF GRAPHS

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF MASTER OF TECHNOLOGY IN COMPUTER SCIENCE

BY

SATYABRATA NAYAK

UNDER THE SUPERVISION OF

DR. BHARGAB B. BHATTACHARYA

INDIAN STATISTICAL INSTITUTE
203, BARRACKPUR TRUNK ROAD,
CALCUTTA - 700035.

ACKNOWLEDGEMENT

I express my deep sense of gratitude to Dr. Bhargab. B. Bhattacharya for his valuable guidance and encouragement throughout my dissertation work. Also I thank Mr. S.C. Nandy for his suggestions.

Finally I thank Mr. Subhasis Pal and the staff of CSSC for their co-operation and providing an excellent computing environment.

- : CONTENTS : -

INTRODUCTION

Representation of objects and their interrelationships are often done using graphs. These graphical representations have numerous applications in the real world like FLOW DIAGRAMS, NETWORKING, PERT-CPM etc. Also it is extremely useful for Data Base Applications and CASE-TOOLs. Diagrammatic representation is widely used in the functional analysis of information systems. Now-a-days 'computer-aided diagram creation' techniques are employed in the areas of advertising and industrial design. A well-depicted graphic representation can convey a lot more than mere words. Properties that characterize a well-depicted graph are

(i) it has clarity of expression.

(ii) it has an aesthetic appeal.

(iii) it has simplicity of notation.

A frequently encountered problem in these applications is 'LAYOUTING'. Creation of a layout for a graph on a plane may become very complex as the number of edge-crossings, the number of bends, etc, increase. Such a diagram will be very difficult to interpret and analyze.

In this report an effort is made to create a layout for a conceptual graph. The graph is first split into connected components such that the layout for each of these components can be easily created. The number of edge-crossings is minimized by first creating a representation for a maximal planar subgraph of the graph and then adding the remaining edges such that the crossings are minimized.

CHAPTER 1

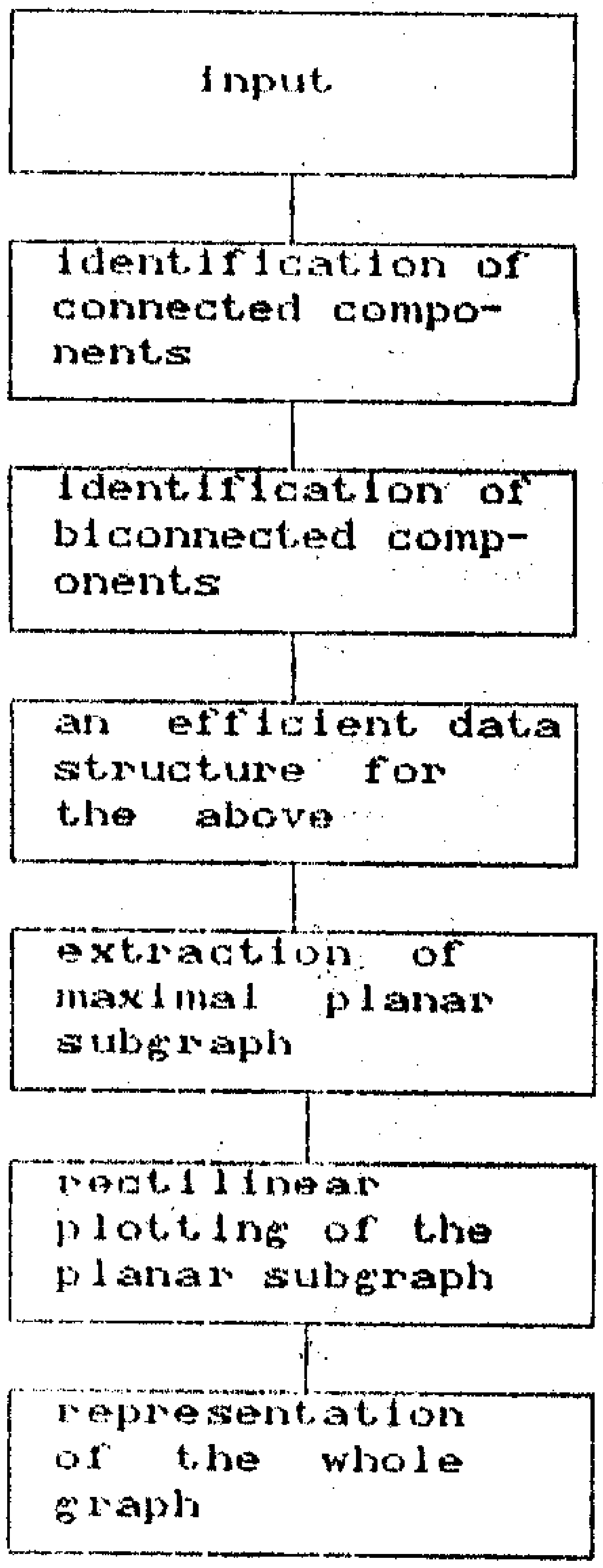
PROBLEM DEFINITION

Our goal is to formulate an algorithm for rectilinear layout of a given graph such that the following conditions are met.

1. The number of edge-crossings is minimized.
2. The global number of bends is minimized.
3. The global length of edges is minimized.
4. The total area of drawing is minimized.

A flexible environment should be created to incorporate new conditions at a later stage. The layout should easily be interpretable and should convey more informationthan the abstract graph. The general flowchart given in the next page shows the different steps of the algorithm. These steps are discussed in details from the next chapter onwards.

GENERAL FLOWCHART :


input
identification of connected compo- nents
identification of biconnected comp- onents
an efficient data structure for the above
extraction of maximal planar subgraph
rectilinear plotting of the planar subgraph
representation of the whole graph

CHAPTER 2

PREPROCESSING STEPS FOR PLANARITY DETERMINATION

2.1.Creating a suitable structure for the input :

The input graph will be of the form of two lists. For the graph shown in Figure 1, the inputs will be :

Set of vertices : 1,2,3,4;

Set of edges : (1 2),(1 3),(2 3),(2 4),(4 3);

First the list of vertices is scanned, and a structure for each vertex is created. The actual list itself is constructed by scanning the set of edges. For the above example the adjacency list structure is shown in Fig 2.

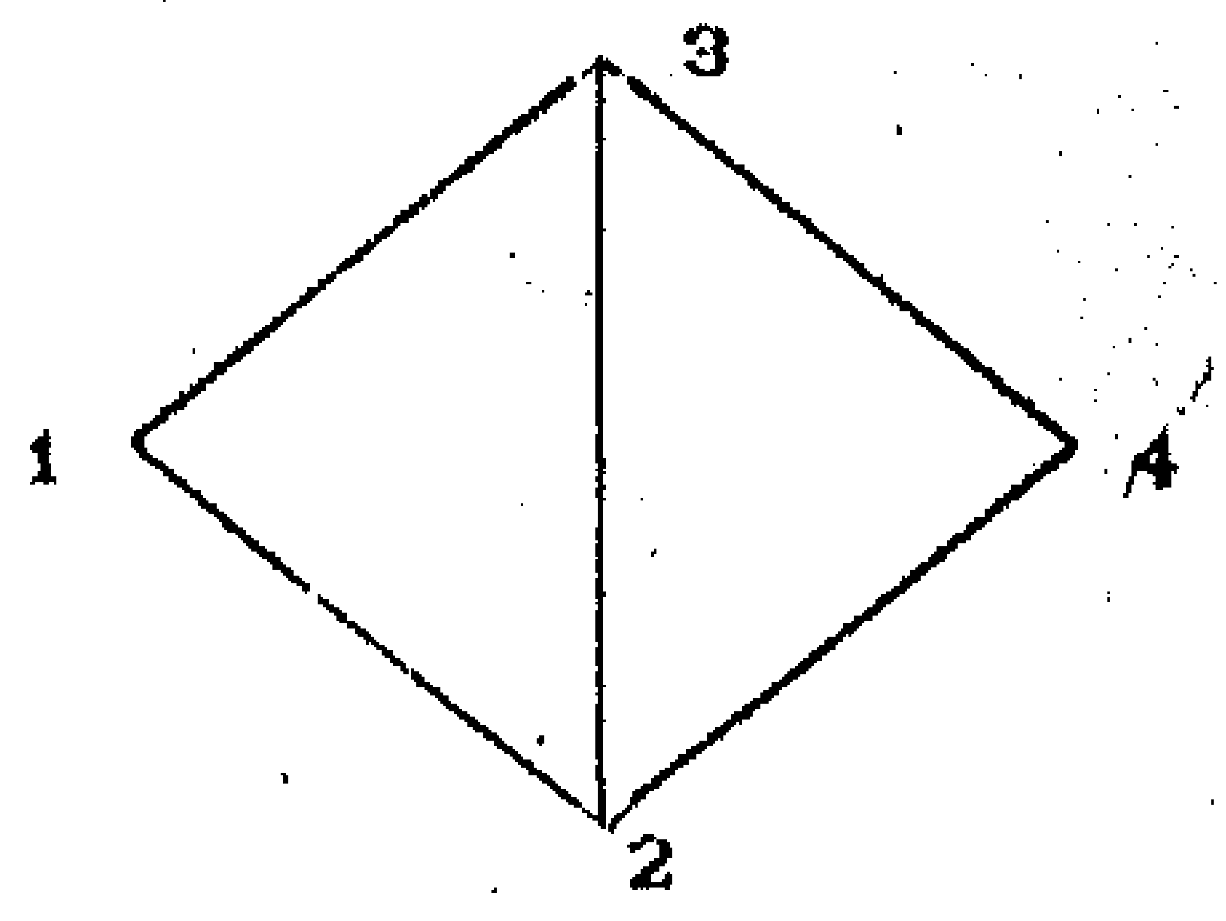


Figure 1

$$\begin{tikzcd}
\text{vertex} \\
1 \arrow[r] \arrow[d, dash] & 2 \arrow[r] & 3 \arrow[r] & \text{NULL} \\
2 \arrow[r] \arrow[d, dash] & 1 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & \text{NULL} \\
3 \arrow[r] \arrow[d, dash] & 1 \arrow[r] & 2 \arrow[r] & 4 \arrow[r] & \text{NULL} \\
4 \arrow[r] \arrow[d, dash] & 2 \arrow[r] & 3 \arrow[r] & \text{NULL} \\
\text{NULL}
\end{tikzcd}$$

Figure 2

The advantage of such a structure is that the vertices if added in future can be easily entered to the structure by simply extending the lists. Also such a structure is very useful in answering queries regarding adjacency.

2.2.Finding the connected components :

A connected graph is one in which any vertex of the graph can be reached from any other vertex by traversing a list of adjacent edges.

A few important properties of a connected graph is given in the following lemmas.

Lemma 1. A connected component of an unconnected graph is a connected subgraph of the unconnected graph.

Lemma 2. A graph is planar if and only if all its connected components are planar.

The connectedness algorithm outlined below requires the creation of a V * V matrix, where V is the number of vertices of the supplied graph. This sort of a matrix is called an adjacency matrix. The entries in such a matrix can be either 0 or 1.

For example an entry of 0 in the i'th row j'th column of this matrix would mean that there is no edge from the i'th to j'th vertex. An entry of 1 would likewise imply the the existence of an edge. This matrix is obviously symmetric for a undirected graph. For the example of Figure 1 the adjacency matrix is shown in Figure 3.

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	1

Figure 3

A detailed description of the algorithm for splitting a graph into its connected components ig given below.

2.2.1. Algorithm for connectedness :

/* To check whether the given graph is connected. If not find out the connected components */

```
Notconsidered : set of vertices .
component [] : array of sets of vertices
adj [v * v] : adjacency matrix of the graph
function    NOTADJACENT ;
input : A row number g (i.e. a vertex), the adjacency matrix adj.
output  : The number of vertices in the graph not adjacent to the
          given vertex.
k , i : integer ;
begin
      k := 0 ;
      for i := 1 to v do
         if ( adj [g,i] = 0 )
         then k := k + 1 ;
      notadjacent := k ;
end ;
/*   Main algorithm   */
Notconsidered  <---- all vertices ;
for j := 1 to v   do      /*  v is the number of vertices  */
component [j]  <---- NULL ;
j  <---- 1;
while  ( notconsidered ≠ NULL ) do
 begin
    component [j] = g  ∪  component [j]    /*    g  is  a  vertex
                              belonging to notconsidered  */
       /*   symbol ∪ stands for UNION   */
     repeat
        n_1  <----  notadjacent [g] ;
        for i := 1 to v do
```

```
            if ( adj [g,i] = 1 )
              then g := g OR i ;      /* OR the g'th and i'th rows  */
          $n_2$   <---   notadjacent [g] ;
      until ($n_1$ = $n_2$ ) ;
      for i := 1 to v  do
          if ( adj [g,i] = 1 ) then
          begin
             component [j] <---  component [j] ∪ {i} ;
             notconsidered <---  notconsidered - {i} ;
             j  <---   j + 1 ;    /* number of components  */
          end;
end.
```

After the above algorithm has been run. component [i] will contain the set of vertices belonging to the i'th component. Each of the connected components obtained by the above method is to be subjected next to a splitting into biconnected components.

2.3 Separating the self loops :

Self loops if exist will be separated in each of the connected components. Since the existence of a self loop has no contribution towards nonplanarity , for planarity testing these loops can be ignored. Finally during rectilinear plotting the loops will be added again to the original graph.

2.4 Finding the biconnccted components :

Before we explain what a biconnected graph is , it would be useful to know what an articulation point is.

An articulation point is a vertex in a graph such that all paths between two other vertices necessarily pass through this point. For example in the Figure 4 vertices 3 and 5 are the articulation points, since all paths between 1 and 4 pass through 3 and all paths between 4 and 6 pass through 5.

A biconnected graph is a graph which has no articulation points.

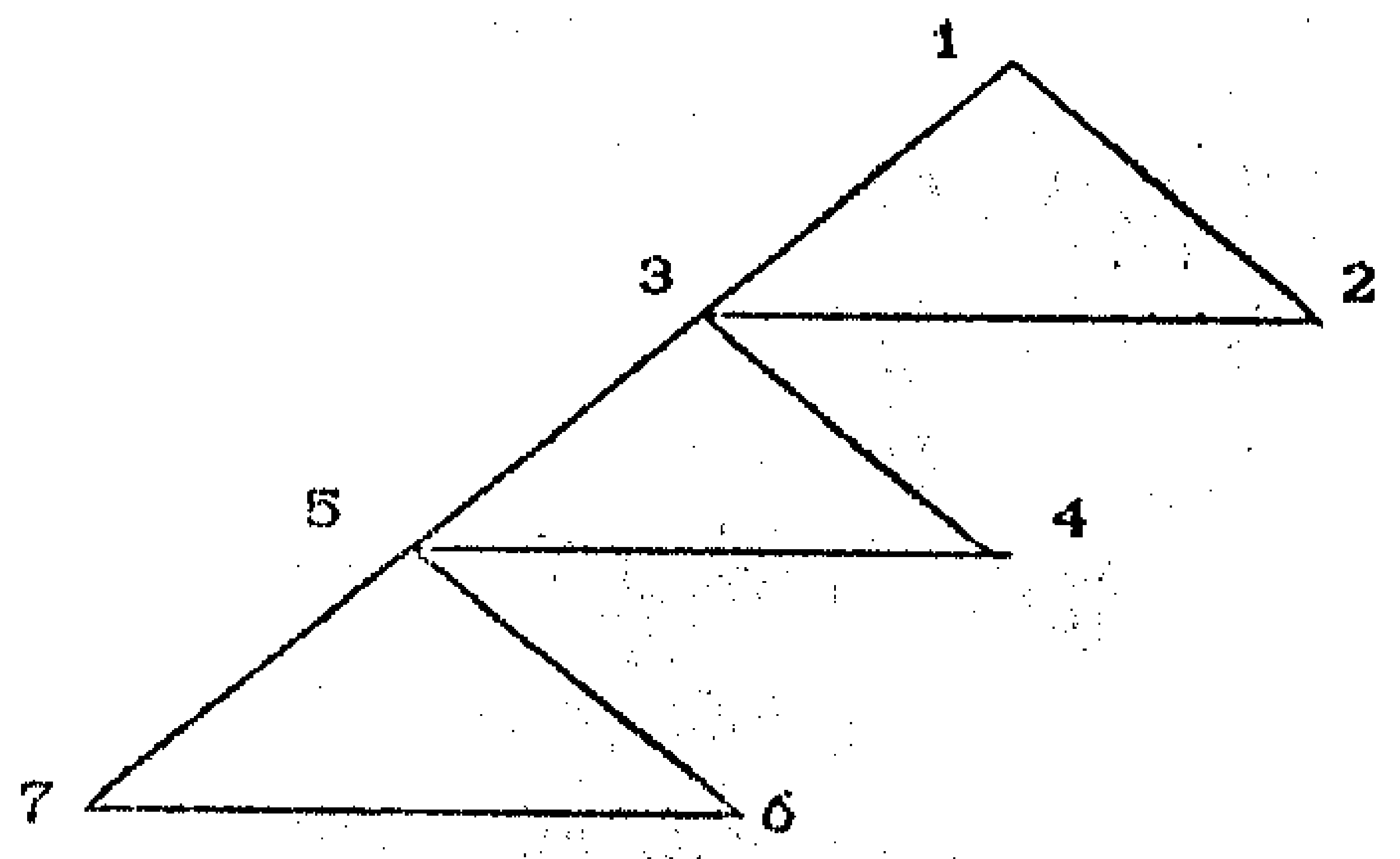


Figure 4

A few important properties of the biconnected components are given in the following lemmas.

Lemma 3. A connected graph is planar if and only if all its biconnected components are planar.

In this step all the connected components will be split into the biconnected components. The splitting into biconnected components provide an efficient preprocessing before the Hopcroft-Tarjan's algorithm is applied on the graph .

The algorithm used for splitting into biconnected components is a variant of the depth-first search technique. This is a technique used to scan a graph exhaustively. It ensures each vertex of the graph and each edge are scanned exactly once. The splitting into biconnected components uses the following algorithm.

2.4.1 Algorithm for biconnectedness :

Input : The input will be a connected component represented by an adjacency list structure, A(v).

Output : The biconnected components of the input component.

```
    /*  Initializations  */
```

```
mark all vertices 'new' ;
count := 0 ;    /*  used to give the numbering to vertices  */
DFS (v) ;
begin
  mark v 'old' ;    /*  the vertex v has been scanned  */
  DFNUMBER (v)  <— count ;  /*  the vertex has been numbered  */
  count  <— count + 1 ;
  LOW (v)  <— DFNUMBER (v) ;
  for each  w  ∈  L(v) do
    begin
        push edge (v,w) into a stack if it is not already in it and
        if it does not belong to any biconnected component
        previously found ;
        if w is 'new'
        begin
           father (w)   <— v ;     /*   father implies that v is
                      scanned just before w and is w'parent  */
           DFS (w) ;
           if ( LOW (w) ≥  DFNUMBER (v) ) ;
              then    /*  a biconnected component has been found  */
                 pop all edges in the stack upto and including (v,w)
                 i.e.  these  edges  constitute  the  biconnected
                       component
        end
        else  /*  if w is 'old'  */
        if ( w is not father (v) ) then
        LOW (v)  <—  min ( LOW (v) , DFNUMBER (w) ) ;
  end;
```

end;

Notations used :

A vertex a is said to belong to the list of vertex b (i.e. a ∈ L(b)) if vertex a is adjacent to b.

i.e. if the structure is as shown below,

a ———> b ———> c ———> d ———>NULL

Here b,c,d belong to the list of a.

In the above algorithm the depth-first search technique splits the edges of the graph into two nonintersecting sets of tree edges and back edges (i.e. fronds). It gives directions to the edges (the direction in which edges are scanned) and a numbering to the vertices (called DFNUMBER henceforth). The numbering indicates the order in which vertices are scanned. The algorithm invokes itself recursively. These properties are not very much used here though they will be used a lot during planarity testing. More on them will be discussed in the next chapter.

The output of the biconnectedness algorithm for a input graph of Figure 4 is shown in Figure 5.

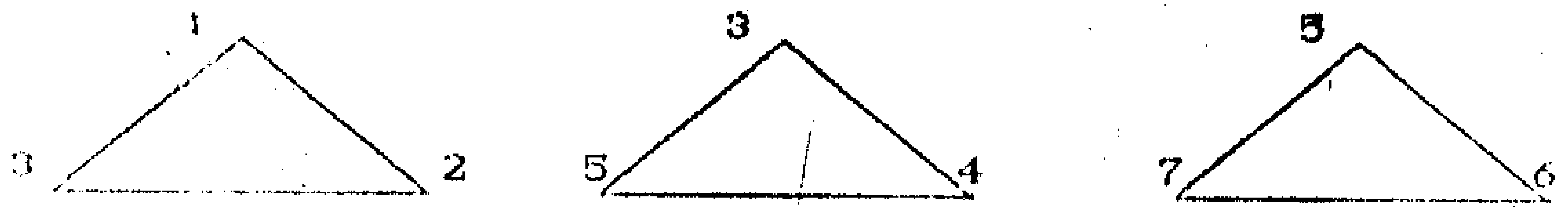


Figure 5

The next task to be accomplished is planarity testing. The output of the biconnectedness algorithm will be used as input for the planarity testing . The details of planarity testing will be discussed in the next chapter.

CHAPTER 3

PLANARITY TESTING

3.1 Aim : To determine whether a graph is planar. If not to get ~~a~~ maximal planar subgraph of it. A maximal planar subgraph of a nonplanar graph is a planar graph to which the addition of any more edge cause nonplanarity.

3.2 Method: The Hopcroft-Tarjan planarity algorithm [1]is used for this purpose. In addition to the planarity determination, it gives information about the order and the placement of different paths. This uses a depth-first search and has O(V) time and space bounds, where V is the number of vertices in the graph. Other planarity algorithms (i.e. by Kuratowski and by Auslander and Parter, Mondshein) require more time. Kuratowski[6]'s method requires an amount of time proportional to at least v^6. Auslander and Parter method was to construct a representation of a planar embedding of the given graph. If such a representation can be completed then the graph is planar; if not ,then the graph is nonplanar. But it requires a time complexity of O(v^3). The Hopcroft-Tarjan's algorithm is an improvement of the iterative version of the method originally proposed by Auslander and Parter, and correctly formulated by Goldstein. Mondshein[5] in his algorithm added one vertex at a time until the entire graph was constructed. But the order of vertex selection was crucial.Mondshein[5]'s implementation required O(v^2)time. Hence the Hopcroft-Tarjan algorithm whose complexity is the best is choosen here.

3.3 **Procedure :** First depth-first search (DFS) is applied on the biconnected component. It provides numbering to the vertices (say from 1 to N , where N is the total number of vertices) in the same order as in the search. The graph to be searched is represented by adjacency lists A(v) as described earlier.

3.3.1 DFS Algorithm

/* This is to be applied on the adjacency list A(v), where A(v) represents one biconnected component. The pseudo code algorithm for DFS is as follows. */

```
begin
   procedure    DFS (v,u) .    /* u,v,w are vertices & u is the
                                  father of v */
   begin
      n := NUMBER(v) := n+1;
      a : Statements corresponding to LOWPT calculations are to be
      inserted.
      for     w  ∈  A(v)  do
       begin
         if  NUMBER(w) = 0  then
           begin
             tree edge  <—  (v w) ;
             DFS (w,v) ;
             b : Statements corresponding to LOWPT calculations are
             to be inserted.
           end ;
         else if NUMBER(w)  <  NUMBER(v)   and   w ≠ u
```

```
                    begin
                        frond <—— (v w) ;
                        c   :   Statements   corresponding   to   LOWPT
                        calculations are to be inserted.
                    end ;
            end ;
        end;
    for i = 1 until v do NUMBER(i) := 0 ;
    n := 0 ;
    DFS (s,0) ;      /* s is the starting vertex */
end;
```

The Output of this algorithm will be the lists of tree and back edges (i.e. fronds). Also it gives the DFS number of each vertex according to the search order. So it is a transition point between the vertex numbers given by the user and the number in which all calculations have to be done for determining planarity.

3.3.2 Hopcroft-Tarjan planarity algorithm

Explanation : The DFS numbered graph will be split into an initial cycle C, and a number of paths with the help of pathfinding algorithm. Then a planar embedding of the initial cycle is constructed. Then the paths are added (in the same order as they are generated by the path finding algorithm) to their corresponding cycle. Now it is to be checked whether the cycle along with the path can be embedded on a plane without any intersection. It may require some transformations like one path which is previously placed inside the cycle to be transformed outside it. One example is shown in Figure 6. This process can be continued until all the paths are embedded without any intersection. Otherwise the graph is declared to be nonplanr.

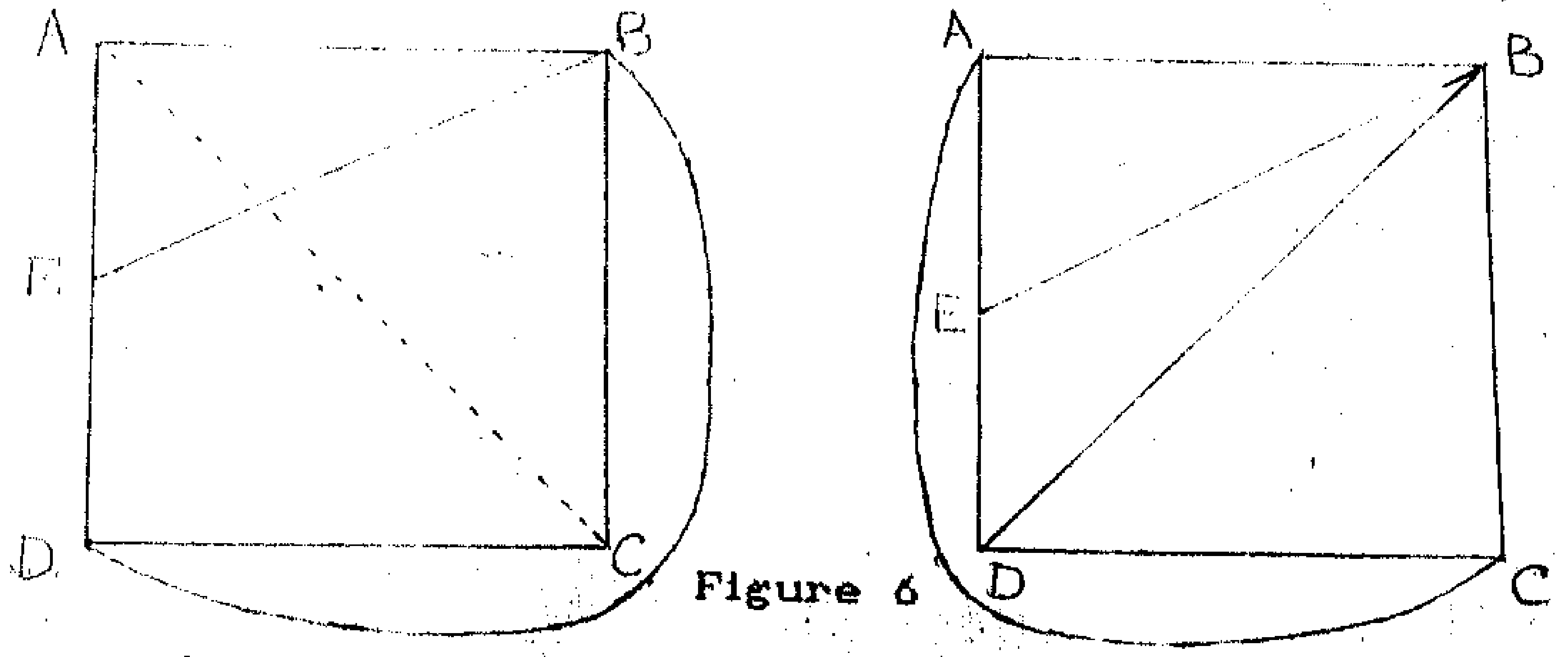


Figure 6

Steps of the planarity algorithm :

(1) Apply DFS to the biconnected graph to number the vertices and to get a palm tree.

(2) Create an ordered adjacency list in order to give the paths a perfect order.

(3) Apply algorithm pathfinder to find the initial cycle and other paths.

(4) Construct a planar embedding for initial cycle C.

(5) Now for each segment (i.e. a set of paths) e (G - C)

{ Check whether (C + newpath) is planar. If necessary change the path from inside to outside or from outside to inside.

If (C + newpath) is planar then add the newpath to the planar representation.

Else declare it to be nonplanar. }

Step 1 of the planarity algorithm is already explained. Step 2 is as follows.

3.3.2.1 Formation of ordered adjacency list :

The reason of finding such a list is to get the paths in a perfect order so that embedding can be done until a nonplanarity is reached. Because of this list the paths in one segment will be encountered at a time. To find such a list we have to calculate two values called LOWPT1 and LOWPT2. Let us define these values as

LOWPT1 : It is the lowest vertex below v reachable by a frond from a descendant of v.

LOWPT2 : It is the second lowest vertex below v reachable by a frond from a descendant of v.

For a complete graph with five vertices (i.e. K_5) DFS and LOWPT values are as follows.

vertex number	DFS number	LOWPT1 number	LOWPT2 number
1	1	1	1
2	2	1	2
3	3	1	2
4	4	1	2
5	5	1	2

Figure 7

These values can be calculated by inserting statements in DFS algorithm for different conditions. LOWPT values depend only on fronds those leaving v (i.e.the vertex for which LOWPT values to be calculated) and descendant of v. The statements to be added to the DFS algorithm are as follows.

```
a :    LOWPT1(v) := LOWPT2(v) := NUMBER(v) ;
b :    if LOWPT1(w)  <  LOWPT1(v)
```

```
      then begin
                 LOWPT2(v) := min ( LOWPT1(v) , LOWPT2(w) );
                 LOWPT1(v) := LOWPT1(w) ;
              end
       else if LOWPT1(w) = LOWPT1(v) then
                 LOWPT2(v) := min ( LOWPT2(v) , LOWPT2(w) );
       else     LOWPT2(v) := min ( LOWPT2(v) , LOWPT1(w) );
c :   if NUMBER(w)  <  LOWPT1(v)
        then begin
                   LOWPT2(v) := LOWPT1(v) ;
                   LOWPT1(v) := NUMBER(w) ;
               end
         else if NUMBER(w)  >  LOWPT1(v)
                then LOWPT2(v) := min ( LOWPT2(v) , NUMBER(w) );
```

The output of the DFS along with these statements is shown in Figure 7 for the input graph of K_5.

Now Θ(phi) values for each of the edges are to be calculated using the LOWPT values.

$$\Theta = \begin{cases} 2 * W & \text{if } v \dashrightarrow w \text{ (frond)} \\ 2 * \mathrm{LOWPT1}(w) & \text{if } v \dashrightarrow w \text{ (tree edge) \& } \mathrm{LOWPT2}(w) >= V \\ 2 * \mathrm{LOWPT1}(w)+1 & \text{if } v \dashrightarrow w \text{ (tree edge) \& } \mathrm{LOWPT2}(w) < V \end{cases}$$

Here W and V are the DFS values of the vertices w and v. Now an ordered adjacency list(with Θ value sorted in the increasing order)is to be formed which will be the input for the Hopcroft-Tarjan algorithm.

What is the necessity of ordered adjacency list ?

Pathfinding algorithm (to be explained next) generates the paths. The paths thus generated are also ordered so that we can embed them according to their order. Because of the ordered adjacency list (which is formed taking into consideration the LOWPT values) the paths are ordered.

3.3.2.2 Finding the paths :

This is the 3rd step in Hopcroft-Tarjan's algorithm. To generate a path we will be traversing the tree edges from the ordered adjacency list until a frond is reached. A single frond may also constitute a path. A path will be terminated when a vertex with DFS number lower than the previous one is encountered. The pathfinding algorithm is as follows.

PATHFINDING ALGORITHM

```
begin   /*  To generate paths in a biconnected palm tree with ordered
adjacency list A(v). Vertex s is the start vertex of the current
path  */

    procedure  PATHFINDER(v) ;

        for  w  ∈  A(v)  do

            if  v ⟶ w (tree edge) then

               begin  if s = 0 then

                        begin  s  :=  v ;

                                  start new path;

                        end;

                     add (v,w) to current path ;

                     PATHFINDER(w) ;

               end

               else begin  /*  v ⟶ w (frond)  */

               if s = 0 then

                  begin  s := v ;
```

```
                    start new path;
                end;
                add (v,w) to current path ;
                output current path ;
                s := 0 ;
            end;
        s := 0 ;
        PATHFINDER(1);
end;
```

The paths thus genarated have several important properties, which are summarized in the following lemmas.

Lemma 4. Let $p : s \Longrightarrow^{*} f$ be a generated path. If we consider the fronds which have not been used in any path when the first edge in p is traversed, then f is the lowest vertex reachable via such a frond from any descendant of s. If $v \neq s$, $v \neq f$, and v lies on p, then f is the lowest vertex reachable from a descendant of v via any frond (all fronds from descendants of v are unused when the first edge of p is traversed).

Lemma 5. Let $p : s \Longrightarrow^{*} f$ be a generated path. Then $f \longrightarrow^{*} s$ is in the spanning tree of p. If p is the first path, p is a cycle; otherwise p is simple. If p is not the initial path, p contains exactly two vertices (f and s) in common with previously generated paths.

Lemma 6. Let $p_1 : s_1 \Longrightarrow^{*} f_1$ and $p_2 : s_2 \Longrightarrow^{*} f_2$ be two generated paths. If p_1 is generated before p_2 and s_1 is an ancestor of s_2, then $f_1 \le f_2$.

Lemma 7. Let $p_1 : s \Longrightarrow^{*} f$ and $p_2 : s \Longrightarrow f^{*}$ be two generated paths with the same start and finish vertices. Let v_1 be the second vertex of p_1 and let v_2 be the second vertex of p_2. Suppose p_1 is generated before p_2, $v_1 \neq f$ and LOWPT2(v_1) < s. Then $v_2 \neq f$ and LOWPT2(v_2) < s.

For the proof of above lemmas [1] may be refered.

3.3.2.3 Embedding the paths :

This constitutes the 4th and 5th step of Hopcroft-Tarjan's algorithm. Here we test the planarity of the biconnected component, G by attempting to embed the paths(i.e.generated by the pathfinder) one at a time in the plane. First path will be the initial cycle C. This cycle consists of a set of tree arcs $1 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n$ followed by a frond $v_n \rightarrow 1$. The vertex numbering is such that $1 < v_1 < v_2 < \cdots < v_n$.Now initial cycle C is embedded on the plane. When C is removed, G falls into several connected pieces,called segments.

Segment (definition) : A segment consists of a group of paths. The starting vertex of a segment is a vertex on the initial cycle, C. The order of the path generation is such that all paths in one segment are generated before paths in any other segment.

According to the Jordan Curve theorem all the paths in one segment must be embedded completely on one side of C. So while embedding the paths if any one path results in a conflict then all the paths belonging to the same segment is to be transfered to the other side of C. The convention is as follows.

The segment S is embedded on the left of C if the orientation of edges (clockwise in the plane) around v_i is (v_{i-1},v_i), (v_i,w) , (v_i,v_{i+1}). The segment is embedded on the right if the orientation of edges around v is $(v_{i-1},v_i),(v_i,v_{i+1}),(v_i,w)$. A frond which enters C is embedded on the left if the segment to which it belongs is on the left of C. If (x,v_j) is a frond which enters C on the left , the orientation of edges around v_j is $(v_{j-1},v_j),(x,v_j),(v_j,v_{j+1})$. The embedding algorithm is given below.

```
procedure EMBED ;
begin      /* To embed a ordered biconnected graph in the plane */
   L := R := B := the empty stack ;
   find the initial cycle C ;
   while some segment is unexplored do
      begin
         initiate search for path in next segment S ;
         when tree arc v —> w is already considered delete entries
         on L & R and blocks on B containing vertices no smaller
         than v ;
         let p : s ==>* f be the first path found in S;
         while position of top block determines position of p do
         begin
           delete top block from B ;
           if block entries corresponds to left
           then switch block of entries from L to R and from R to L;
           if still an entry is left on the block conflicting p
           then declare nonplanar ;
        end ;
     if p is normal
      then add last vertex of p to L ;
     add new blocks to B corresponding to p and blocks just removed
     from B ;
     combine top two blocks on B ;
   end ;
end.
```

The above algorithm uses two stacks L and R. Whenever a path is embedded its end vertex is entered in either of the stacks. In this algorithm always stack L is used first. Once all the paths in one segment (say S_1) is embedded, L will contain the end vertices of all the paths belonging to S_1 (if at all there exists any such planar representation). Now the base vertex of the next segment S_2 will be compared with the existing stack entries. All entries which are greater than the base vertex of S_2 will be deleted from the stack because S_2 starts at a lower vertex and extends downwards. So vertices (of the former segment S_1) which are more than the base vertex of S_2 are no more useful.

The end vertex of the first path of S_2 will be placed on the stack L if it is less than the top entry of L. Otherwise it will be placed on stack R. Each time a vertex is changed from left to right or from right to left their relation will be stored in a different stack, B in terms of a pair (x,y) where x and y corresponds to vertex numbers in L and R. Certain important properties of procedure EMBED is listed in the following lemmas.

Lemma 8. Path p : $v_i \Rightarrow^* v_j$ may be added to the planar embedding by placing it on the left (right) of C if and only if no frond (x,v_k) previously embedded on the left (right) satisfies $v_j < v_k < v_i$.

Lemma 9. Procedure EMBED runs to completion if and only if G is planar. Otherwise the procedure leads towards nonplanarity.

For proof [1] may be refered.

After all the above steps are carried out, a plot of the maximal planar subgraph has to be obtained. Then the remaining nonplanar edges are to be added taking into consideration the other aesthetics. Detailed explanation is given in the next chapter.

CHAPTER 4

RECTILINEAR PLOTTING OF THE PLANAR GRAPH

In this chapter how maximal planar subgraph can be plotted rectilinearly is discussed. Here plotting is done using heuristics with the information collected from the planarity algorithm. A hierarchy of squares is used over which the vertices can be placed. The convention is as follows :

The i'th square will contain (2i-1) * 4 vertices.

i.e. 1st square ⟶ 4 vertices.

2nd square ⟶ 12 vertices.

3rd square ⟶ 20 vertices.

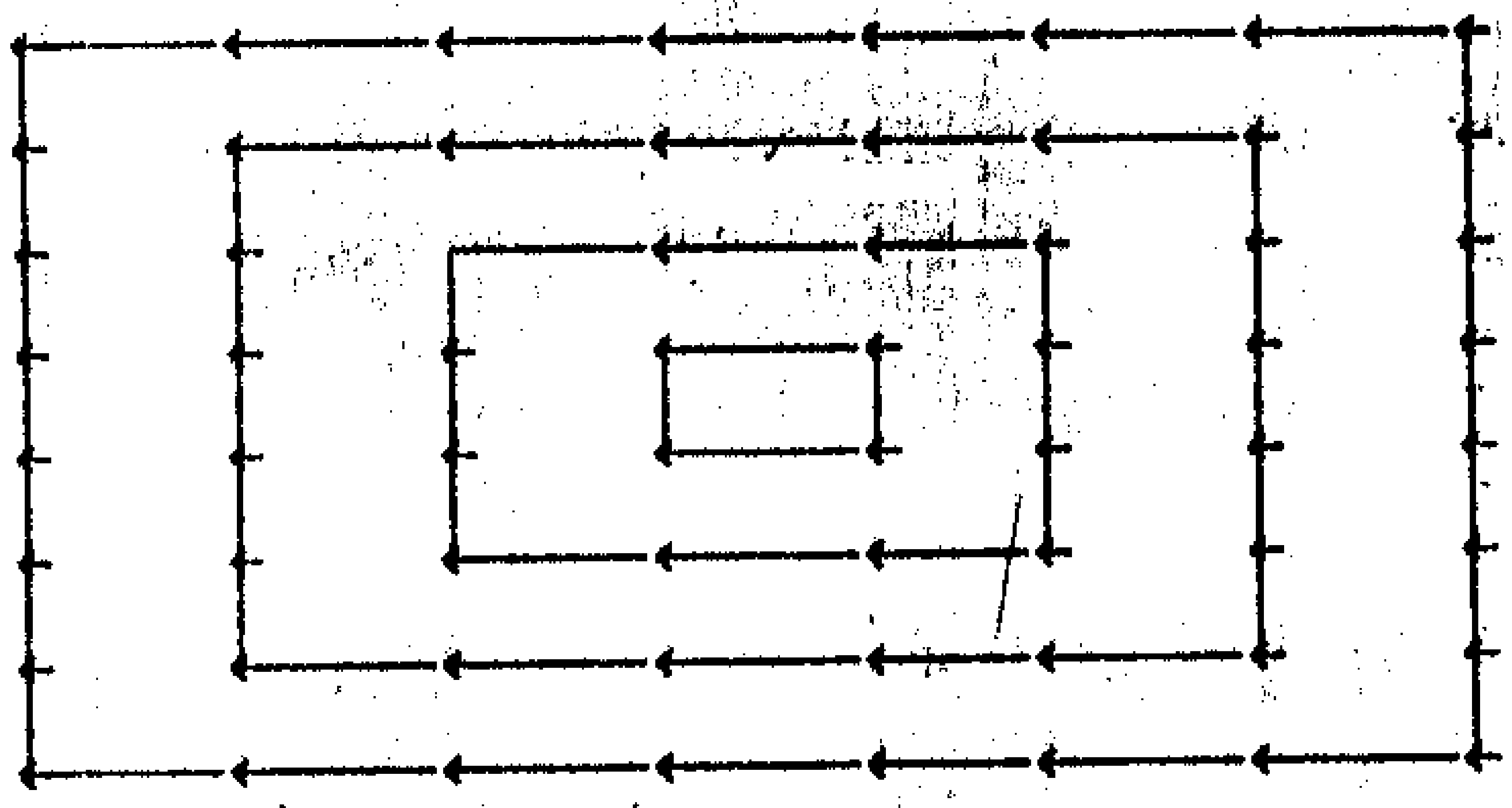


Figure 8

In figure 8 above vertices can be placed over the positions marked by '← '. The unit distance between the positions of two vertices is called the 'scale'.

4.1. To place the initial cycle C :

From the embedding algorithm we know what the segments to be placed inside C are ,and which are to be placed outside. The set having more vertices will be plotted outside C.

i) Smallest i satisfing the two conditions given below will be selected for initial cycle C.

c-1 : N_C (the number of vertices on C) $\leq$ (2i-1) * 4

c-2 : $\sum_{p=1}^{i-1} (2p-1) * 4 \geq N_{set}$.

where N_{set} is the number of vertices in the set of segments to be placed inside C and

p is the number of paths to be placed inside C.

ii) If the i'th cycle is choosen, then the separation between two vertices will be k times the scale.

$N_C * k \leq (2i-1) * 4$

Smallest k satisfying the above equation will be taken.

iii) The first vertex will be placed on the top left corner of the i'th cycle. The remaining vertices will be placed with a gap of k * scale from the first vertex in the clockwise direction.

4.2. To place cycle C_1 for segment S_1:

Such a cycle which is the first path of the segment S_1 will be plotted in the anticlockwise direction. It starts from the initial cycle C (i.e. at base vertex) and terminates at a vertex (i.e. end vertex) which is again on C. The reason for giving the paths anticlockwise direction is that , the other paths in segment S_1 will be started and terminated at vertices which are more than the end vertex and less than the base vertex of C_1. The rest paths of segment S_1 will be plotted with respect to C_1 in the same manner as C_1 is plotted with respect to initial cycle C. Also the other segments can be plotted in the same manner as S_1.

4.3. Plotting the fronds :

A frond has only the base vertex and the end vertex. So the direction is dependent on other aesthetics.

All numbers used for vertices in planarity testing are DFS numbers (i.e. this is different from the numbering used for the original graph) . Now the numbers can be replaced by their original value. Different biconnected components which were decomposed previously in chapter 2 will be joined and placed together ignoring the nonplanar edges (How to add nonplanar edges with minimum crossing will be discussed in the next chapter.)

Similarly different connected components are to be placed together constructing the original graph. Self loops which were deleted at the beginning will be added back. The outputs for a number of inputs are shown in the concluding pages. The addition of nonplanar edges is discussed in the next chapter.

CHAPTER 5

ADD CROSSINGS

As long as the graph is planar there is no need to add any crossings. But in the case of nonplanar graphs, the nonplanar edges are to be added to the maximal planar subgraph of the original graph, in such a way that the number of edge-crossings are minimized. The edge-crossings are represented as fictitious vertices.

From the embedding algorithm we can conclude about the nonplanarity and the edges contributing towards nonplanarity. So deleting the nonplanar edges from the input graph we will obtain the maximal planar subgraph of the original graph. In order to know how the nonplanar edges should be added with minimum crossing , duality of the maximal planar subgraph is necessary.

Before finding the duality we have to convert the maxima planar subgraph into a form where the number of nested faces are

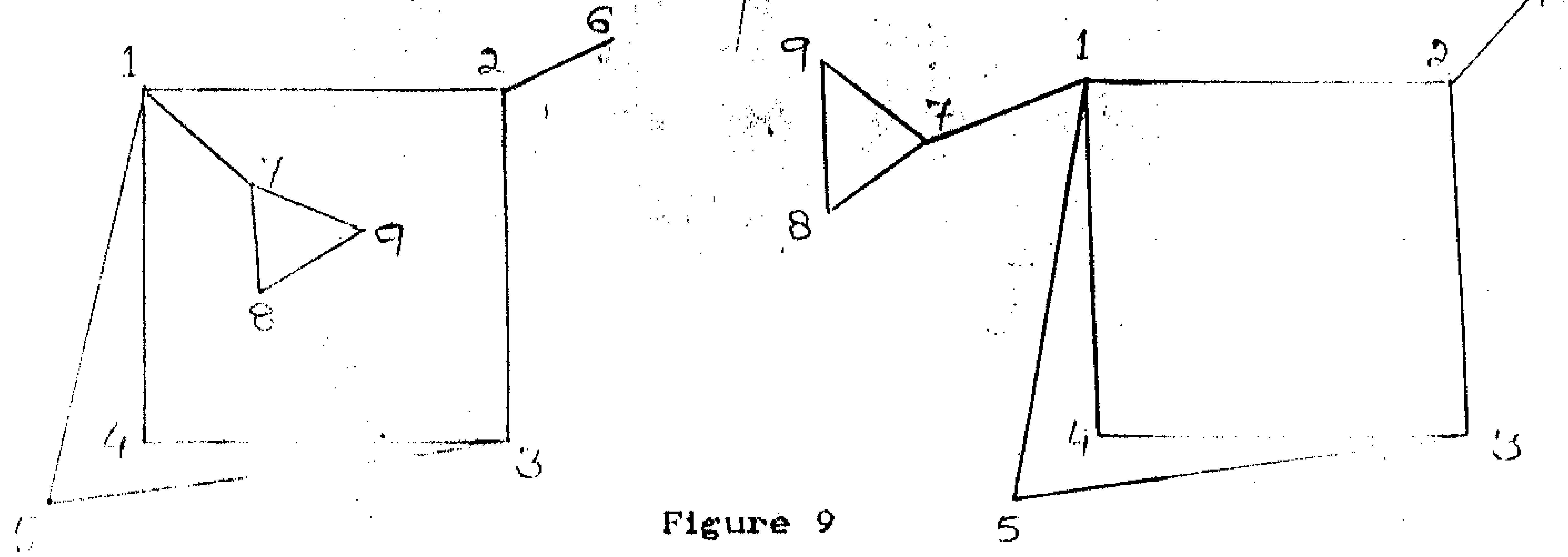


Figure 9

minimum As shown in Figure 9 (same graph though the left one contains one nested face).

How the duality finding helps in minimizing the edge-crossings can be explained with the help of an example say K_5 (the complete graph with five vertices). The paths are generated by the pathfinder algorithm in such a order that when path 4 —> 2 is embedded, nonplanarity will be indicated. So deleting 4 —> 2 the maximal planar subgraph of K_5 will look like Figure 10. Now in the graph of

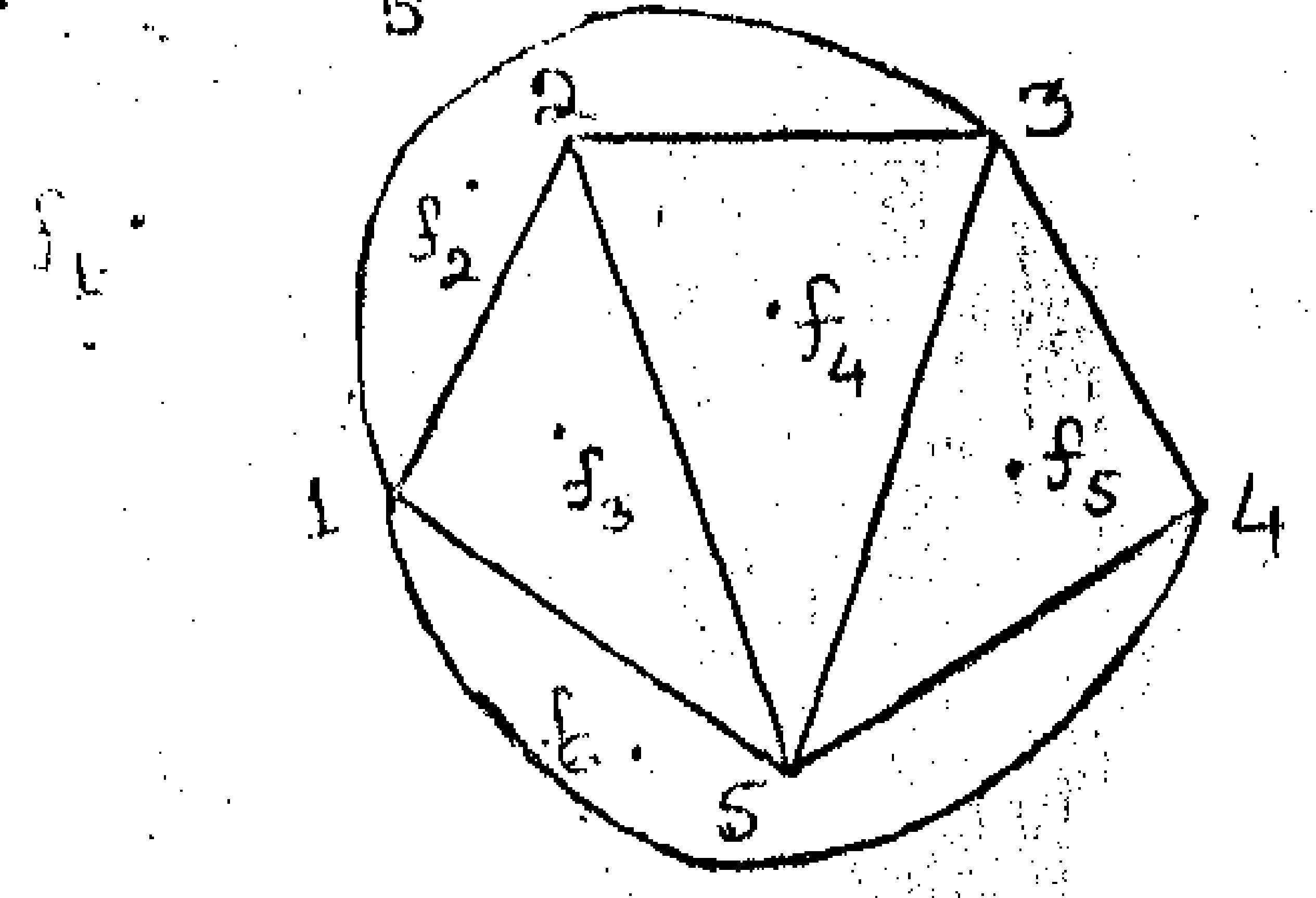


Figure 10

Figure 10 one point will be placed for each face (including the external face) and some numbers are assigned to these points. These will be the vertices for the dual graph.

If face1 and face2 of the maximal planar subgraph are adjacent then the vertices f_1 and f_2 will be joined in the dual graph. In the same manner other edges will be joined in the dual graph for the corresponding adjacent faces of the maximal planar graph . So for our maximal graph of Figure 10 dual graph will look like figure 11.

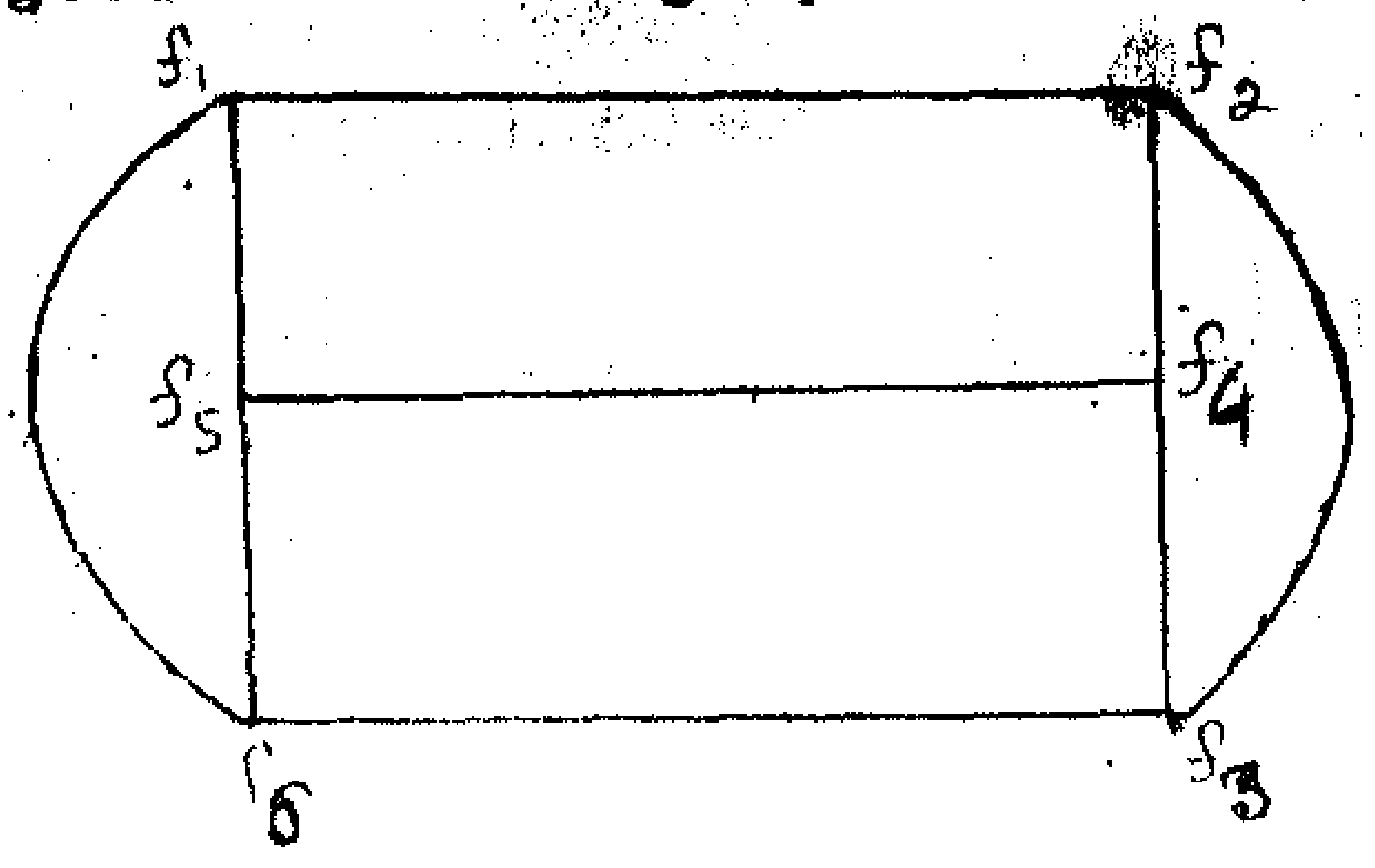


Figure 11

Next our aim is to add the edge 4 —> 2. Since in the maximal planar graph vertex 4 is adjacent to faces f_1 ,f_5 ,f_6 we can place 4 in the face surrounded by vertices f_1 ,f_5 ,f_6 in the dual graph. Now 4 will be joined with f_1 ,f_5 ,f_6 in the dual graph of Figure 12. Similarly 2 will be placed and joined to f_2 ,f_3,f_4.Minimum distance between 4 and 2 is found out in the graph of Figure 12.

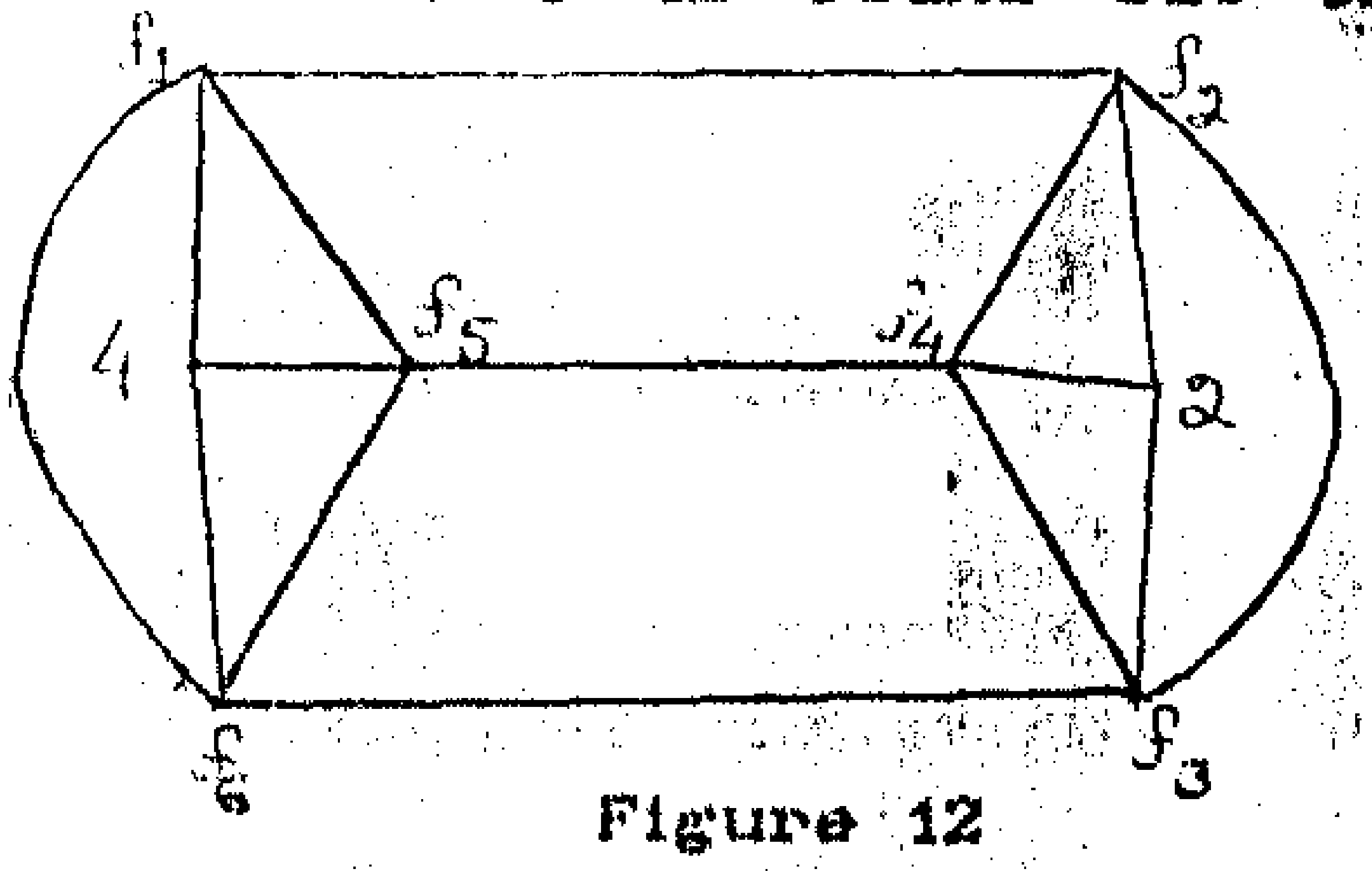


Figure 12

Other cases are 4 —> f_5—> f_4—>2 or 4 —> f_1 —> f_2 —> 2 or 4 —> f_6—>f_3—> 2. So in the maximal planar graph we can add the crossings between either the faces f_5 - f_4 or f_1 - f_2 or f_6 - f_3 so that representation of K_5 is complete. Out of the above cases

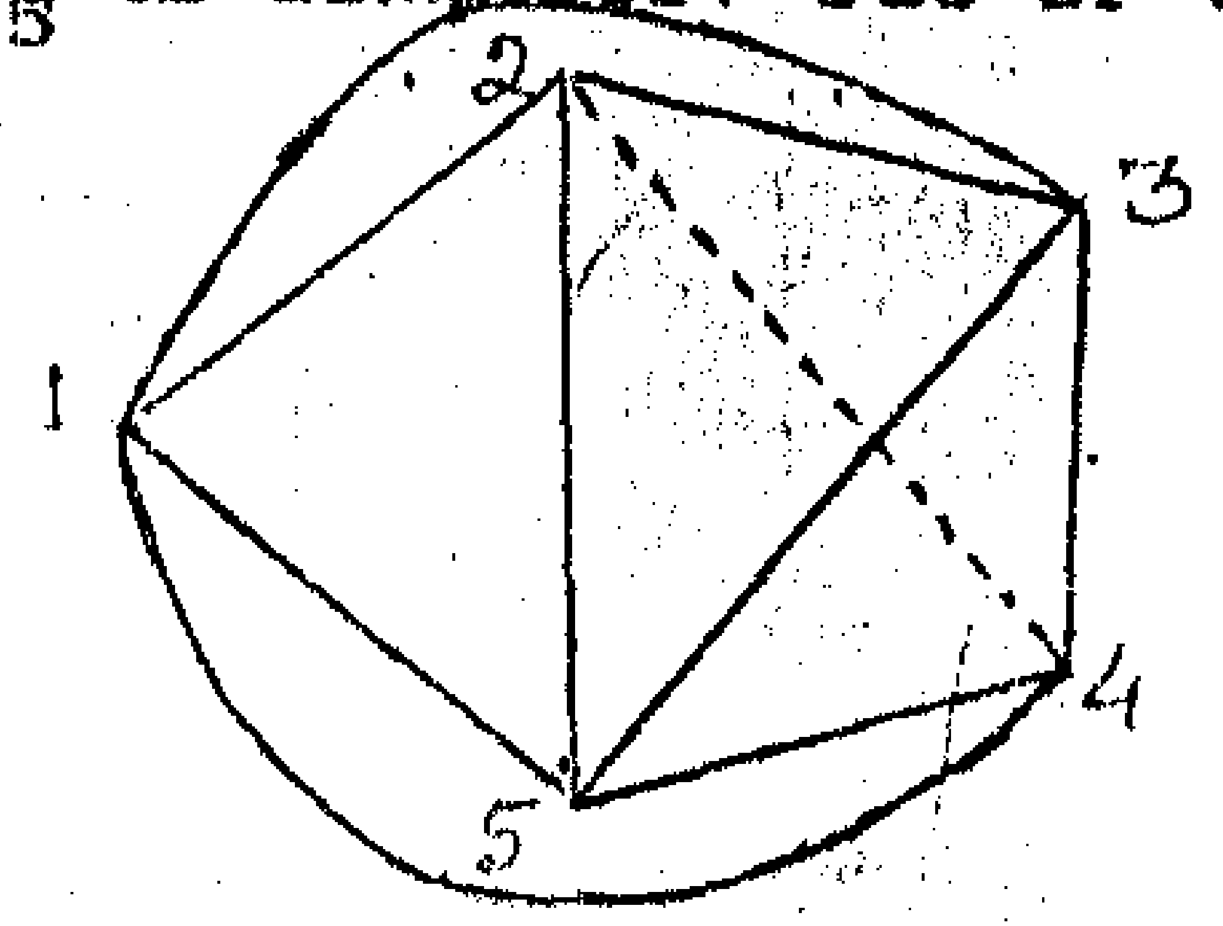
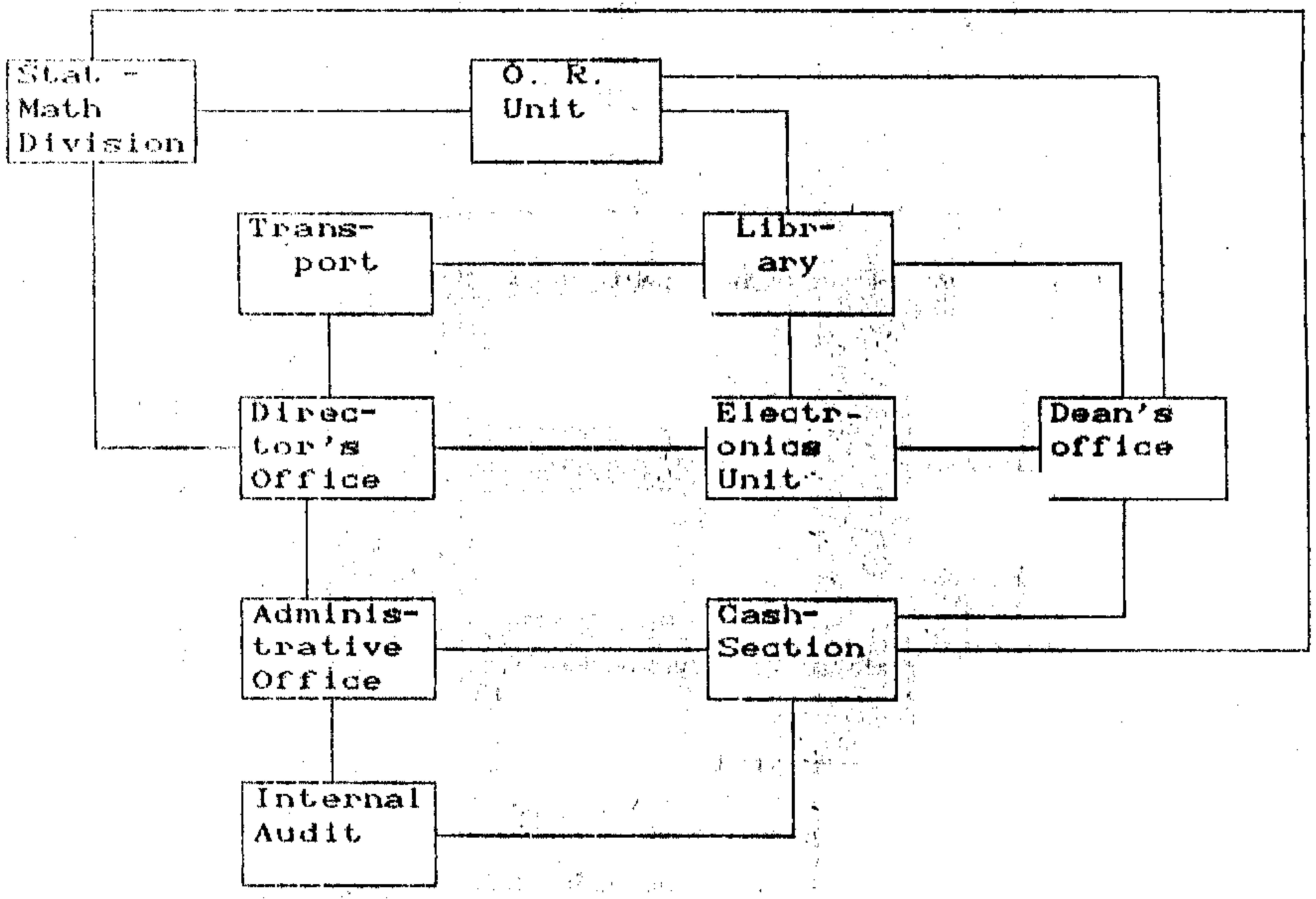


Figure 13

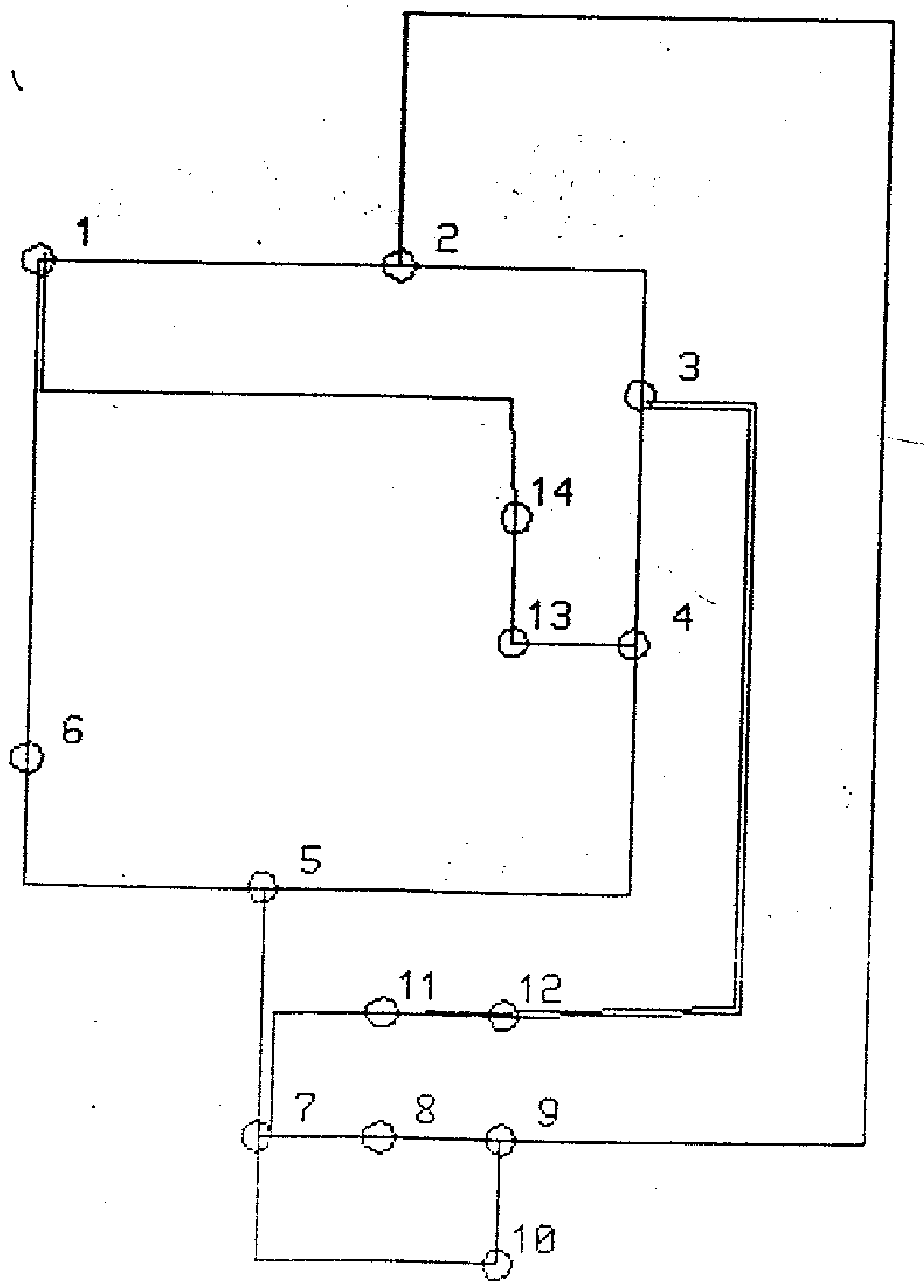
4—> f_5—> f_4—> 2 is considered and the complete graph is shown figure 13. Out of the three possible cases , the choice of a particular case is dependent on other aesthetics. For further bend minimization [4] may be referred.

Figure given below is an example of 'Office Automation' using rectilinear layouting.

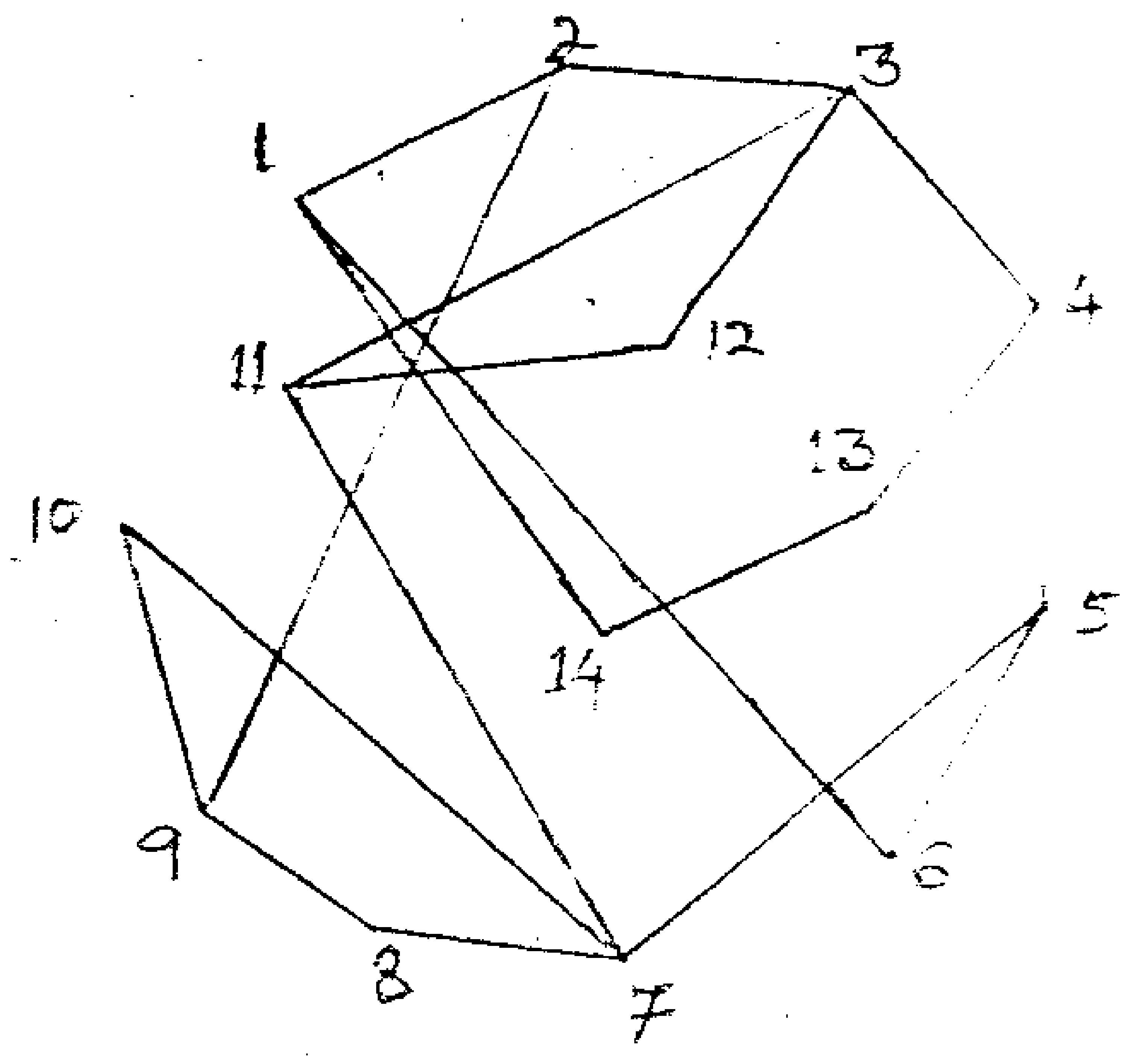


```
E THE SCALE:

E THE INPUT FILE NAME:
ut1.dat
```

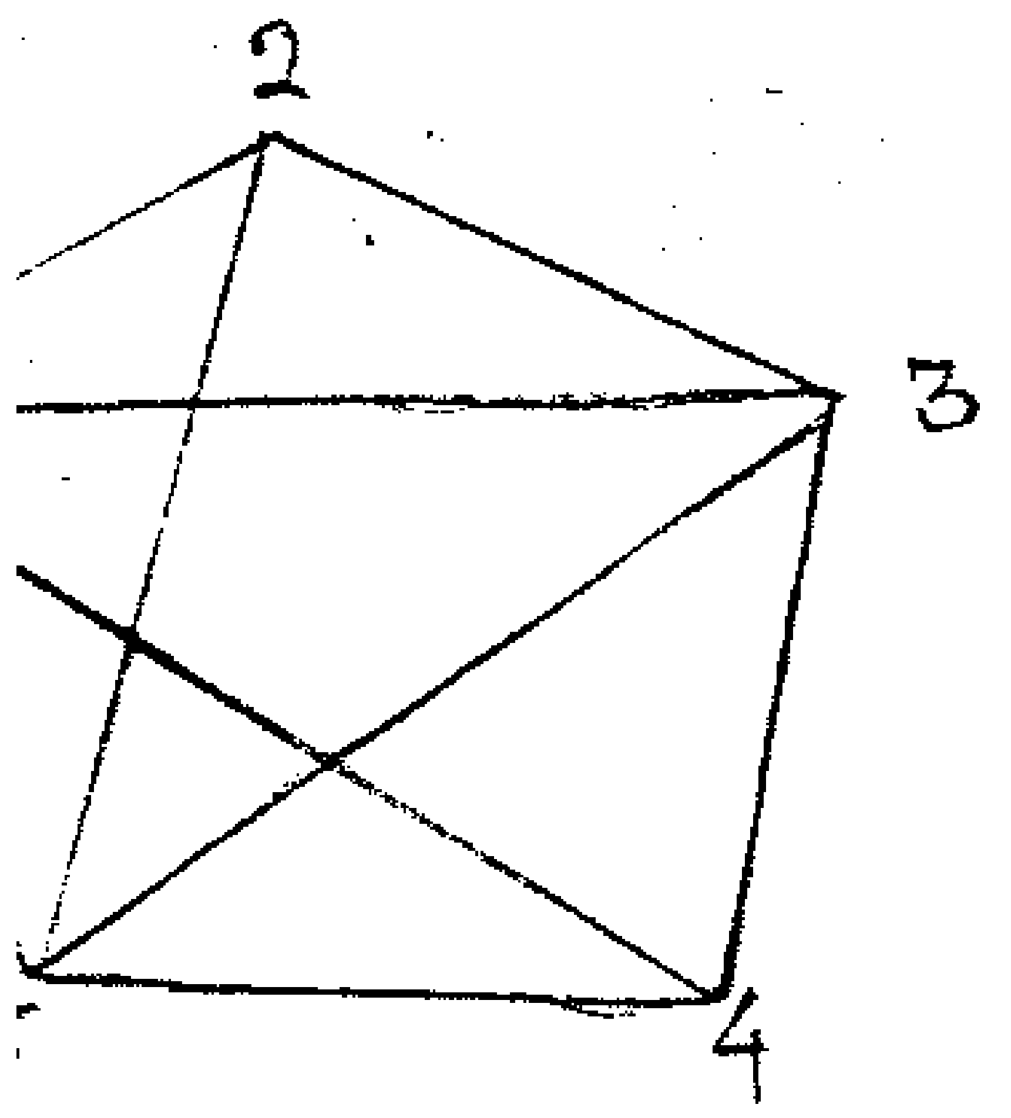


OUTPUT 1

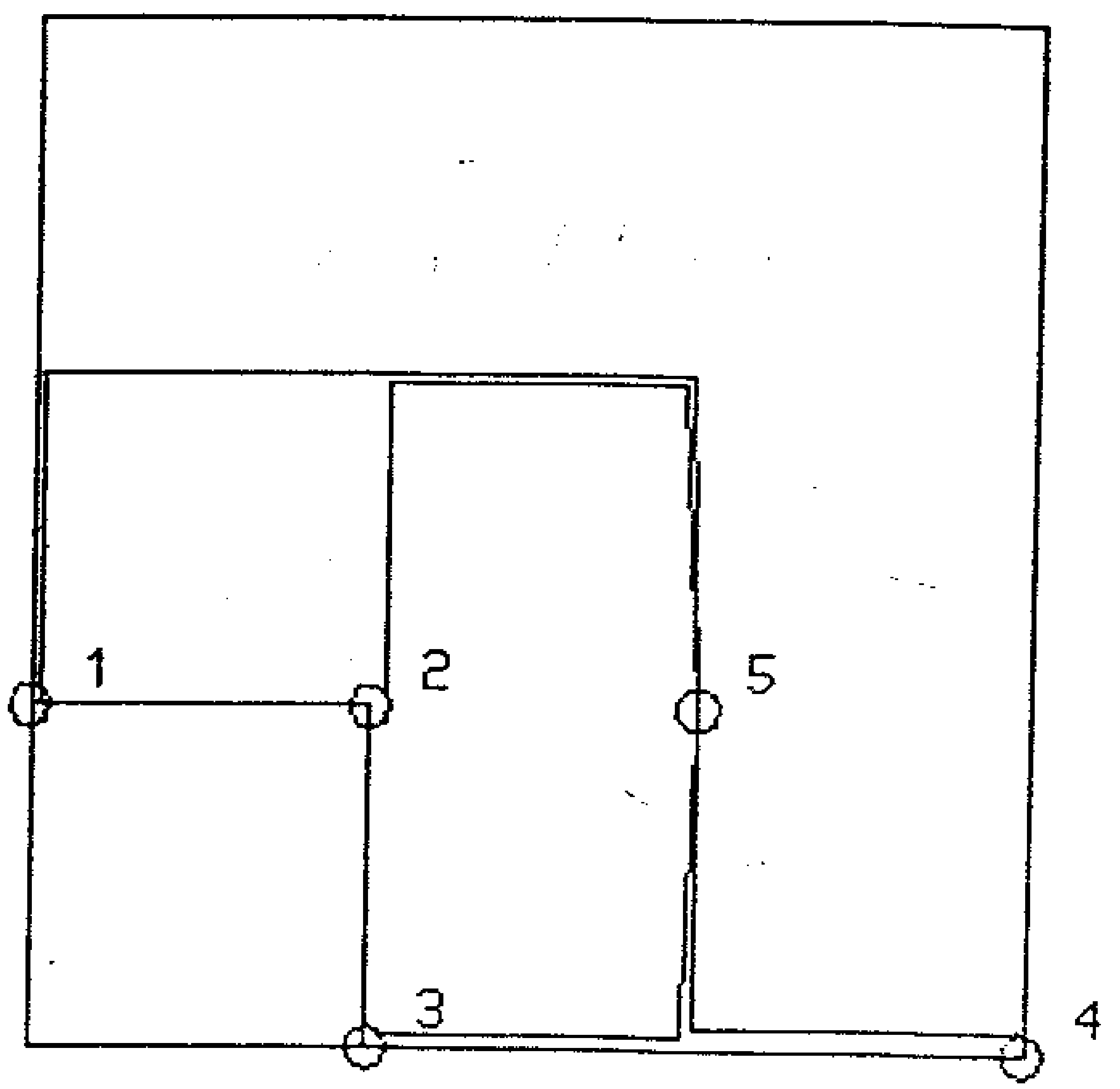


INPUT 1

```
GIVE THE SCALE:
16
GIVE THE INPUT FILE NAME:
input2.dat
$
```

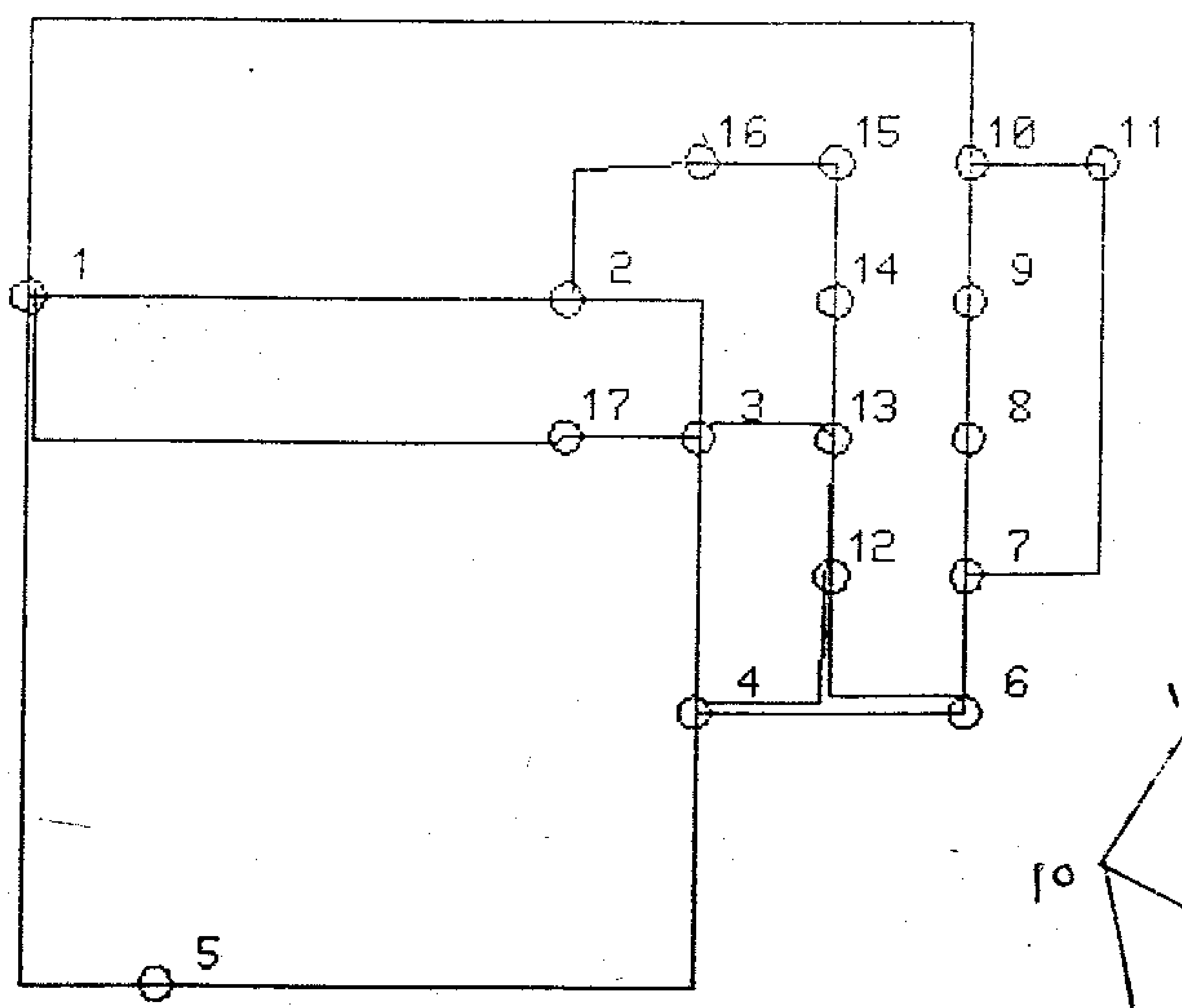


VPUT 2

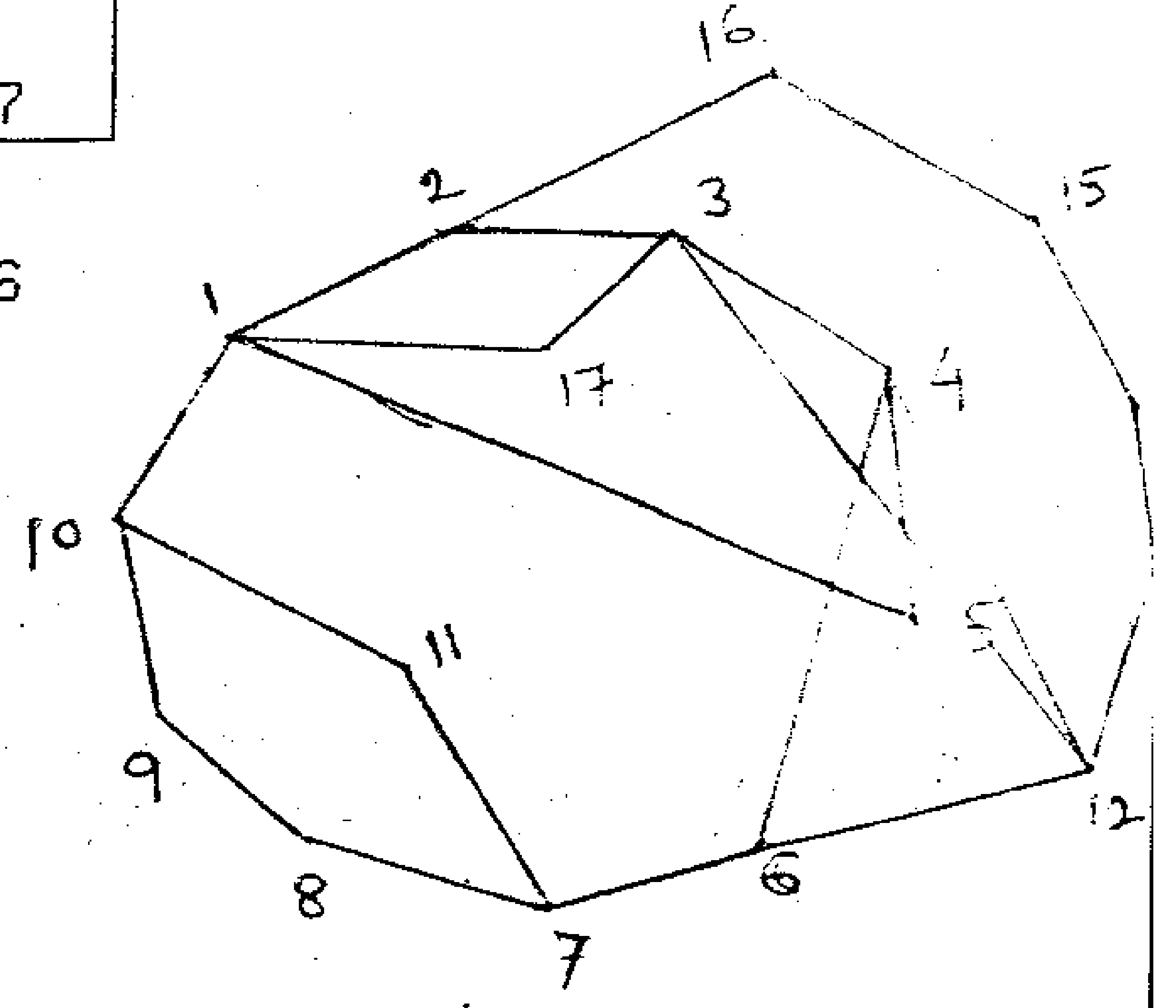


OUTPUT 2

```
GIVETHE SCALE:
8
GIVETHE INPUT FILE NAME:
inpu8.dat
$
```

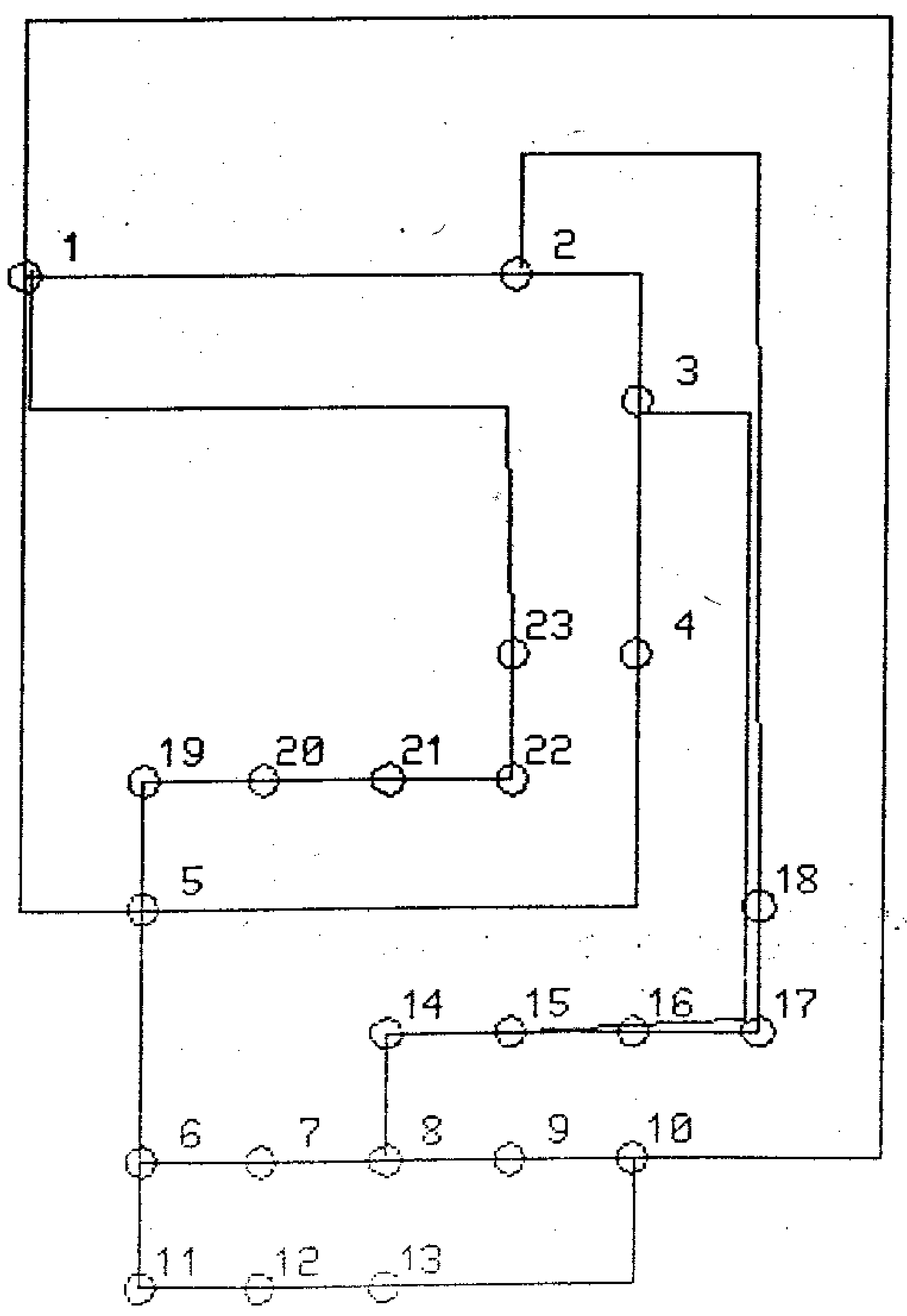


OUTPUT 3

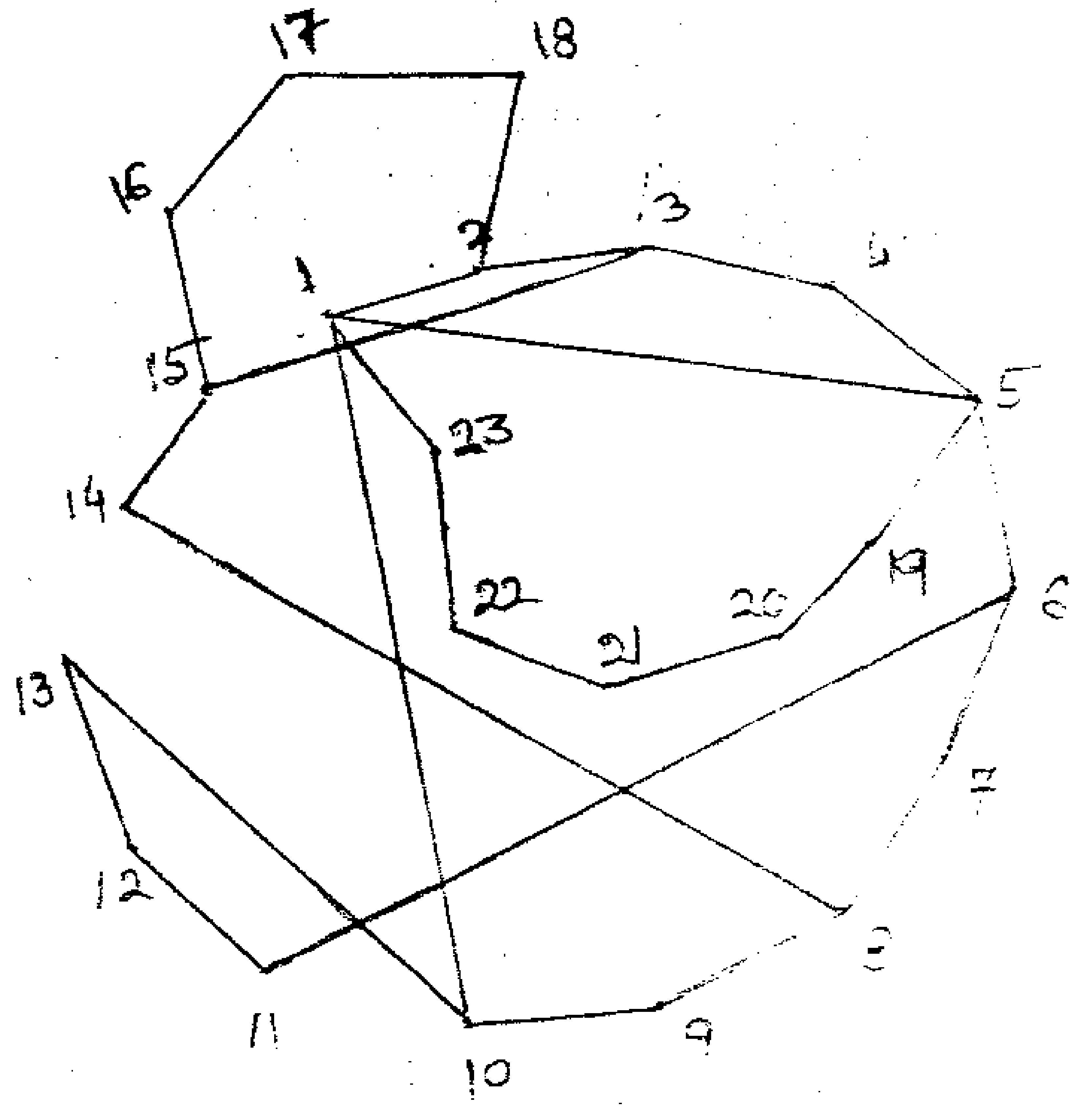


INPUT 3

```
GIVE THE SCALE:
8
GIVE THE INPUT FILE NAME:
input4.dat
$
```

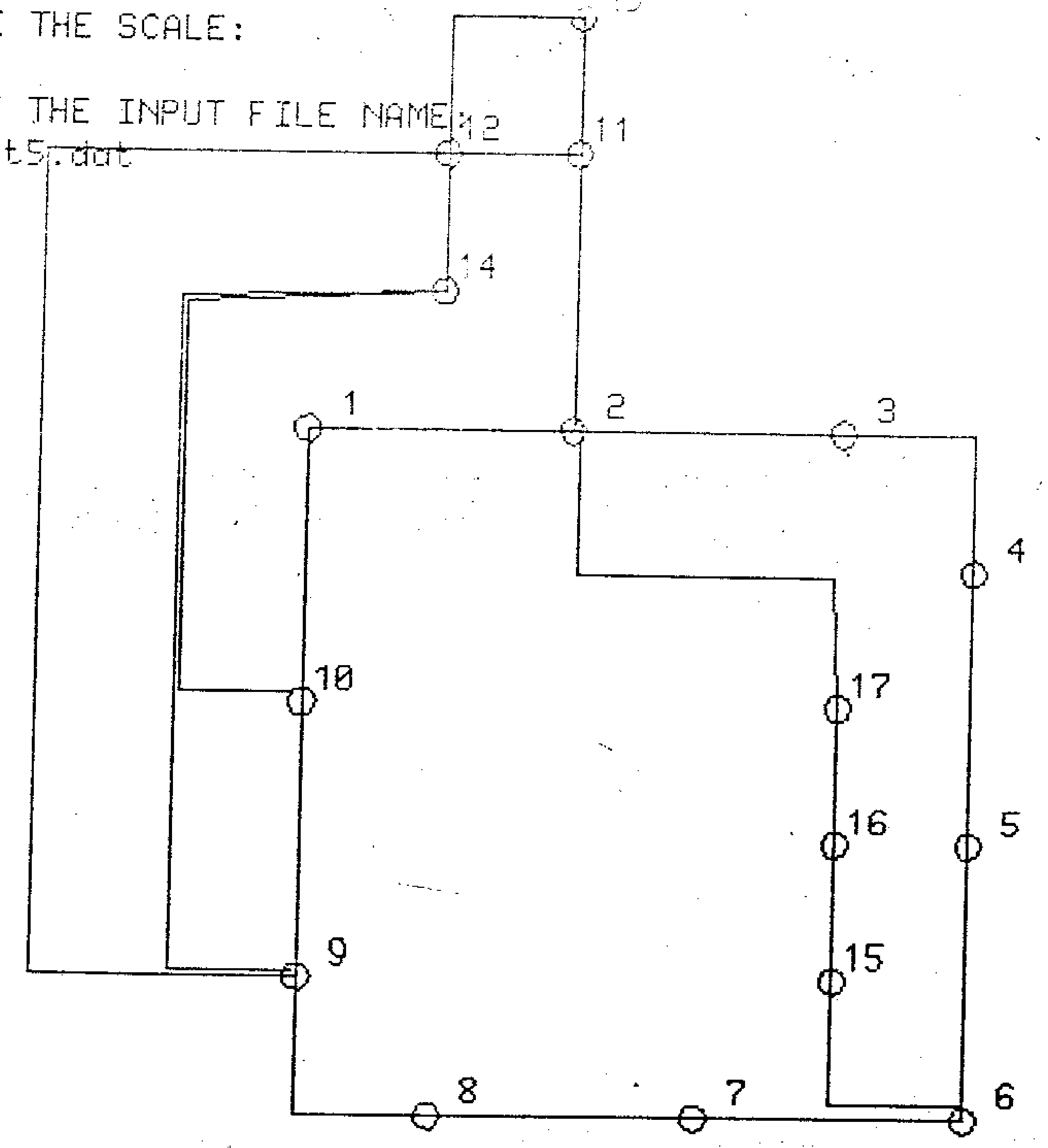


OUTPUT 4

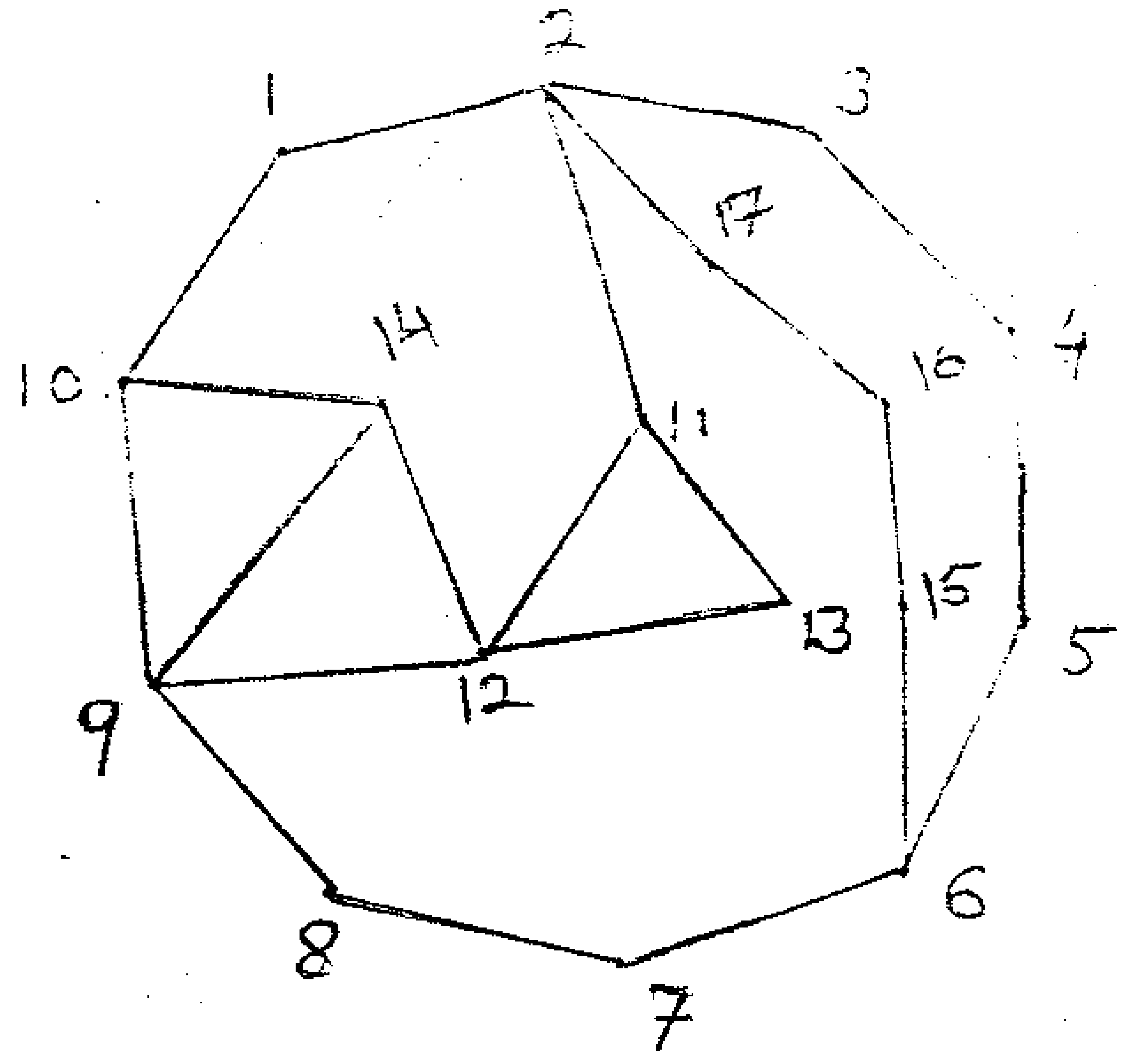


INPUT 4

```
GIVE THE SCALE:
10
GIVE THE INPUT FILE NAME:
input5.dat
$
```



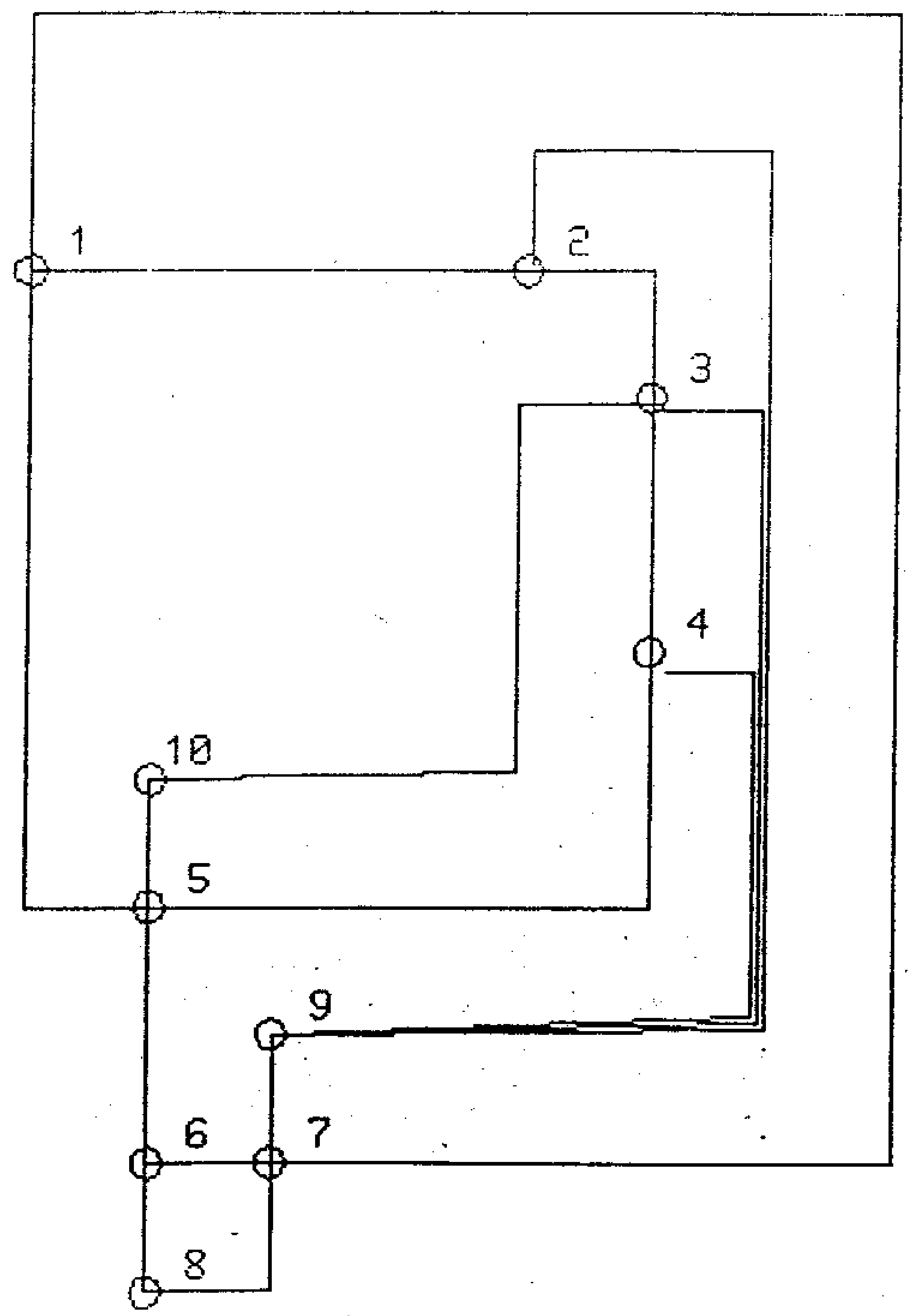
OUTPUT 5



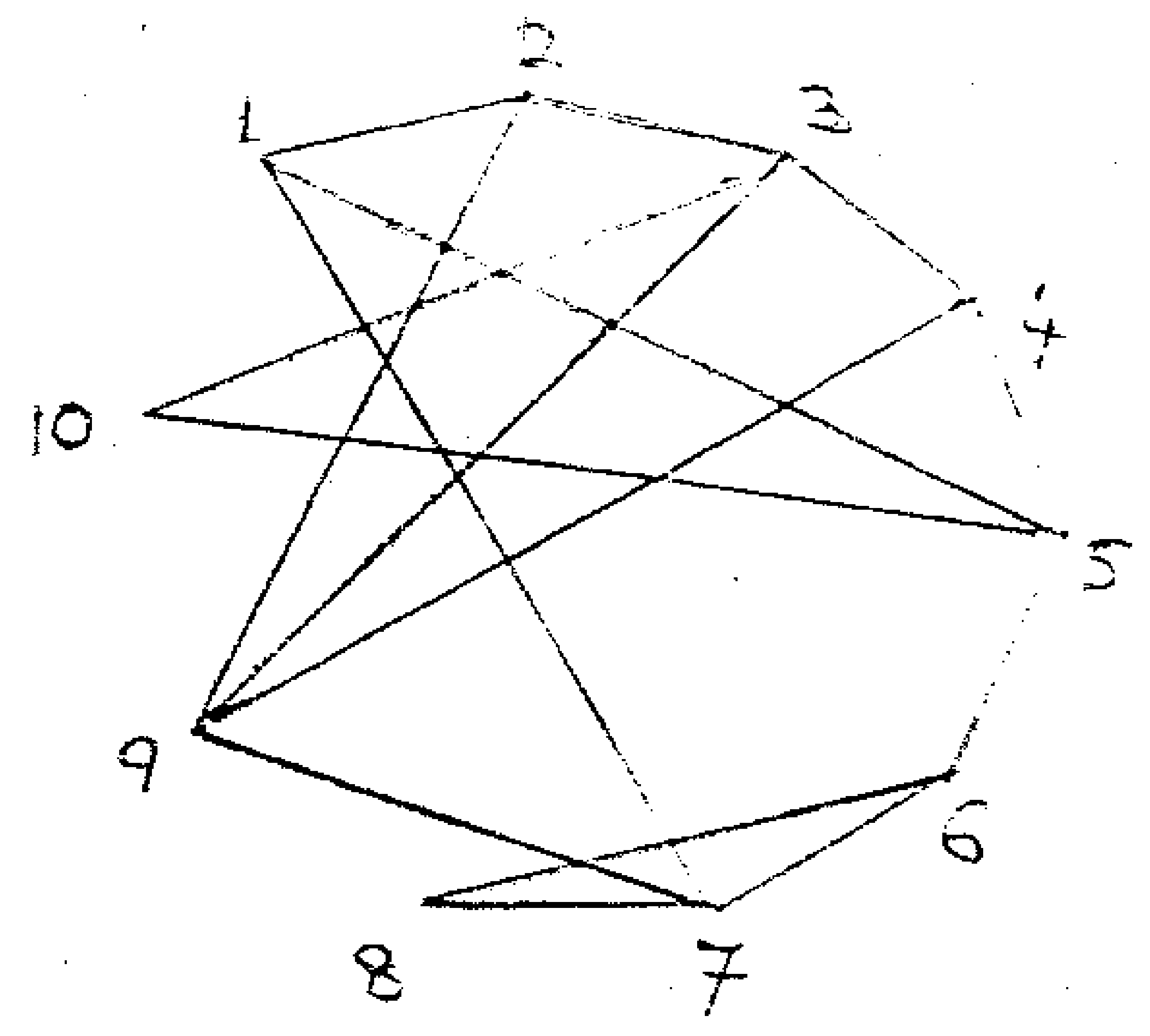
INPUT 5

```
GIVE THE SCALE:
8
GIVE THE INPUT FILE NAME:
input6.dot
$
```

This is the corresponding graph of layout given in Page 28.



OUTPUT 6



INPUT 6

REFERENCES

1. Hopcroft, J. and Tarjan, R., Efficient planarity testing. J. ACM,21, 4(Oct. 1974),pp.549-568.

2. Tamassia, R.,Batini, C. and Tamalo, M., An algorithm for automatic layout of entity relationship diagram, in Entity-Relationship Approach to Software Engineering, Proc. 3rd Internal Conf. on Entity-Relationship Approach, C.G.Davis,S. Jajodia, P.A. Ng and R.T. Yeh, Eds.,North-Holland, Amsterdam , New-York,pp.421-439.

3. Batini Carlo, Enrico Naderlli and Robert Tamassia, A layout algorithm for data flow diagrams, IEEE Transactions on Software Engineering,Vol-12,No-4(April 1986).

4. Roberto Tamassia , On embedding a graph in the grid with the minimum number of bends, SIAM J. COMPUT,Vol 16,No-3(June 1987),pp.421-444.

5. Mondshein, L., Combinatorial orderings and embedding of graphs Tech. Note 1971-35, Lincoln Lab., M.I.T., Aug. 1971.

6. Kuratowski, C., Sur le probleme des corbes gauches en topologie. Fundamenta Mathematicae 15 (1930),271-283.

7. Narsingh Deo, Graph Theory with Applications to Engineering and Computer Science., Prentice Hall of India, 1990.

8. Harary, F., Graph Theory , Addison-Wesley Publishing Company,1988.

9. Aho A.V., Hopcroft J.E., Ullman J.D., Design and Analysis o Algorithms, London, Addison-Wesley, 1974.