CHAPTER IV

TOPOLOGY FOR TERMINAL LAYOUT PROBLEM

4.1 Introduction:

As mentioned in Chapter-I, this stage of the computer communication network is concerned with the topology of the local area network. The topology of connecting customer sites to terminal controllers is decided at this level. The volume of traffic at this stage of the network is much less; and major design criteria in deciding the topology are low cost and high operational flexibility, in the sense that a customer site may enter or leave the network without disturbing the performance of the network to any significant extent.

As discussed in Chapter-3, currently available algorithms for determining the topology of the local area network [10], [22], [28], [34], [102] aim at the generation of a minimal spanning tree; and thus, each of them guarantees a minimal cost connected network. But, they do not consider the much-desired modularity in the structure - if a customer site enters/leaves the network, the topology has to be re-designed once again. Moreover, in the process of generating a minimal cost network, the capacity constraint (in terms of the maximum number of nodes that may be connected) of the terminal controller is left out of consideration; and hence, there is a tendency of some of the terminal controllers becoming over-loaded, in spite of the fact that the average load on the local area network is much less.
In this Chapter, an algorithm is being proposed for optimal connection of terminals to terminal controllers in the terminal layout problem of the local area network. In order to ensure maximum modularity (and hence operational flexibility) in design without any loss of generality, it is assumed that (i) there is no direct connection amongst the terminals, and (ii) each terminal controller has some definite upper bound on the number of terminals it can accommodate. Minimal cost is taken to be the optimality criterion.

The objective of this topological design is to connect customer sites to controllers at a minimal cost, i.e. minimal cost is taken to be the optimality criterion.

4.2 An algorithm for terminal layout design:

4.2.1 A brief description:

In brief, the algorithm works in the following steps:

a) It accepts the cost matrix and the capacity of the terminal controllers as inputs.

b) 'Penalty' in each row of the cost matrix is calculated. Here, the term 'penalty' in any row means the difference between the minimum element and the next minimum element of the corresponding row. So, physically, the term 'penalty' indicates the minimum additional cost to be incurred if a terminal cannot be connected to its nearest terminal controller, due to the restrictions on the capacity constraint.
c) The penalties, sorted in the descending order, are scanned sequentially starting with the penalty having largest value; and the position of the minimum element is checked at first, for possible assignment. If it is not possible, the position of the next minimum is checked. If that is also not available for possible assignment, next minimum in the current row is found out, and backtracking is done so that some of its previous assignments may be altered in order to accommodate the present assignment; the primary idea being to keep the cost incurred in making all the assignments up to any stage of the algorithm, to a minimum.

In view of the fact that the terminals are practically expected to be clustered around some terminal controller, and that the capacity of the terminal controller is expected to be quite high, the backtracking will be necessitated only in very rare situations.

d) The final solution matrix is printed out.

This algorithm does not generate a minimal-cost spanning tree, but it does develop a minimal cost tree satisfying the upper bound on the number of terminals that may be accommodated by any terminal controller.
The algorithm has been implemented in MicroVAX - II using PASCAL. It has extensively been tested for various sets of data. Average execution time taken by MicroVAX - II for various sets of input data, has been listed in the subsequent section of this chapter.

4.2.2 The Algorithm :

Input :
- \( n \) : The number of nodes in the network.
- \( m \) : The number of terminal controllers (excepting the central one).
- \( m_t \) : Maximum number of nodes that can be attached to a terminal controller, except the central one, which has infinite capacity.
- \( CM_{[1:n,0:m]} \) : The cost matrix with \( n \) rows and \( m+1 \) columns, where \( CM_{[i,j]} \) represents the cost of connecting the \( i \)th terminal to the \( j \)th terminal controller.

Output :
- \( INC_{[1:n,0:m]} \) : The incidence matrix, where
  \[ INC_{[i,j]} = \begin{cases} 
  1 & \text{if } i \text{th terminal is assigned to } j \text{th terminal controller} \\
  0 & \text{otherwise} 
  \end{cases} \]
begin

// Initialize the auxiliary cost matrices CP and CI with the values in CM

for a = 1 to n
    for j = 0 to m
        CP [i, j] ← CM [i, j] ;
        CI [i, j] ← CM [i, j] ;
    next j ;
next i ;

// Compute penalty for each row, where the term penalty is defined as the difference between the first minimum and second minimum elements of the corresponding row in the cost matrix. Also, keep track of the columns in which these first and second minimum elements occur. */

for s = 1 to n
    minimum (s, c1, min1) ; // Module minimum is being explained later */
    CP [s, c1] ← max ; // max is greater than any element of CM */
    minimum (s, c2, min2) ;
    row [s] ← s ;
    col1 [s] ← c1 ;
    col2 [s] ← c2 ;
    value [s] ← min2 - min1 ;
next s ;
// Re-initialize the auxiliary matrix CP with the values
in CM ;

for s = 1 to n
    for t = 0 to m
        CP [s, t] ← CM [s, t] ;
next t ;
next s ;

// Description of the module minimum ;
.
module minimum (row, col, min);
begin
    min ← CP [row, 0] ;
    col ← 0 ;
    for i = 1 to m
        if ( CP [row, i] < min ) then
            begin
                min ← CP [row, i] ;
                col ← i ;
            end ;

next i ;
end ;
Sort the array VALUE in the descending order, and arrange the arrays ROW, COL1 and COL2 accordingly. 

\[
\text{Max } \text{VALUE}[i] \geq \text{VALUE}[i+1] ; \ i = 1,2,\ldots,n-1 ; \text{ and in the process, if } \text{VALUE}[s] \leftrightarrow \text{VALUE}[t] , \text{ then make}
\]

\[
\begin{align*}
\text{ROW}[s] & \leftrightarrow \text{ROW}[t] \\
\text{COL1}[s] & \leftrightarrow \text{COL1}[t] \\
\text{and } \quad \text{COL2}[s] & \leftrightarrow \text{COL2}[t]
\end{align*}
\]

where \( s, t \in \{1,2,\ldots,n\} \) and the notation ' \( \leftrightarrow \)' implies interchange.

Next part deals with the assignment. It starts with initialization of the array \( \text{CAP1}[1:m] \) with zero, where \( \text{CAP1}[i] \) stores the total number of assignments to the \( i \)th terminal controller at any instant. The variable \( \text{max} \) stores a large value - larger than a feasible value in the cost matrix. 

\[
\text{for } i = 1 \text{ to } m \quad \text{CAP1}[i] \leftarrow 0 \quad \text{next } i ;
\]

\[
\text{for } i = 1 \text{ to } n
\]

\[
\text{for } j = 0 \text{ to } m \quad \text{INCL}[i,j] \leftarrow 0 \quad \text{next } j ;
\]

next \( i \);
for $i = 1$ to $n$

if ( COL1 $[i] = 0$ ) then // Assign to $i$th terminal controller. 

begin

INC [ ROW $[i]$, COL1 $[i] ] \leftarrow 1$
CP [ ROW $[i]$, COL1 $[i] ] \leftarrow \text{max}$
CI [ ROW $[i]$, COL1 $[i] ] \leftarrow \text{max}$

end; else

begin

if ( CAP1 [ COL1 $[i] ] \neq 0$ ) then // Assign to $i$th terminal controller if its capacity is not filled up.

begin

INC [ ROW $[i]$, COL1 $[i] ] \leftarrow 1$
CAP1 [ COL1 $[i] ] \leftarrow \text{CAP1 [ COL1 $[i] ] + 1}$
CP [ ROW $[i]$, COL1 $[i] ] \leftarrow \text{max}$
CI [ ROW $[i]$, COL1 $[i] ] \leftarrow \text{max}$

end; else // Find 2nd minimum position //

begin

if ( COL2 $[i] = 0$ ) then

begin // Assign to $i$th terminal controller //

INC [ ROW $[i]$, COL2 $[i] ] \leftarrow 1$
CP [ ROW $[i]$, COL2 $[i] ] \leftarrow \text{max}$
CI [ ROW $[i]$, COL2 $[i] ] \leftarrow \text{max}$

end;
else  
   // Assign to i\textsuperscript{th} terminal 
   controller if its capacity is not 
   filled up.  
begin

   if ( \texttt{CAP1[COL2[i]]} \texttt{mt} ) then
      begin
         \begin{verbatim}
         \texttt{INC[ROW[i],COL2[i]]} \texttt{= 1 ;}
         \texttt{CAP1[COL2[i]]} \texttt{=}
         \texttt{CAP1[COL2[i]]} + 1 ;
         \texttt{CP[ROW[i],COL2[i]]} \texttt{= max ;}
         \texttt{CH[ROW[i],COL2[i]]} \texttt{= max ;}
         \end{verbatim}
      end ;
      else
         reallocate ;  \texttt{/** Explained later. **/}
      end ;
end ;

next i ;

\texttt{/** Module reallocate uses sub-modules \texttt{initialize}, \texttt{checl} and 
\texttt{readjust}.**/}

\texttt{Module initialize initializes each element of DIFF [1:n] 
to a very high value - higher than any feasible value of 
cost matrix, and each element of COLUMN [1:n] to zero.}

\texttt{Module checl uses submodules pass and findnewp.}
Module pass searches for any previous assignment and if there is any assignment, it finds out next minimum and its column position in the corresponding row k (say) and assigns the difference between next minimum value and cost of current assignment to DIFF [i].

Module findnewp finds out the minimum of the DIFF array and its position p.

Module check tests whether the minimum value in the DIFF array is less than the difference between 3rd minimum and 1st minimum (or 2nd minimum and 1st minimum), and if the condition is satisfied, it marks the respective terminal and terminal controller.

Module readjust makes readjustment to the incidence matrix if reallocation is necessary.

// Description of the module reallocate

begin

penal1 ← max; penal2 ← max;
initialize;

CP [ ROW [1], COL1 [1] ] ← max;
CP [ ROW [1], COL2 [1] ] ← max;
minimum (ROW [1], col3, min3);
pen1 ← min3 - CM [ ROW [1], COL1 [1] ];
for t = 0 to m do CP [ROW [1], t] ← CI [ROW [1], t] ;
check (col1, rw1, cw1, pen1, penal1);
initialize:

\[
pen2 \leftarrow \min - CM (ROW [1], COL2 [1]) ;
\]

\[
check (col2, rw2, cw2, pen2, penal2);
\]

api \leftarrow penal1 - pen1 ;

ap2 \leftarrow penal2 - pen2 ;

\[
if ( (api \leq ap2) \text{ and } (api \neq 0)) \text{ then}
\]

// Allocate in the 1st minimum position of 1th row and
cancel allocation in minimum position of row rw1
\[
readjust (rw1, cw1, col1);
\]

\[
if ( (ap2 \leq api) \text{ and } (ap2 \neq 0)) \text{ then}
\]

// Allocate in the 2nd minimum position of 1th row and
cancel allocation in 2nd minimum position of row rw2
\[
readjust (rw2, cw2, col2);
\]

\[
if ( (api \geq 0) \text{ and } (ap2 \geq 0)) \text{ then}
\]

begin // Re-allocation is not possible, and hence
assign 1th row to the column corresponding
to the next minimum value.

\[
if (col3 = 0) \text{ then}
\]

begin

\[
if (\neg\text{API}[col3] \text{ mt}) \text{ then}
\]

begin

\[
repeat
\]

\[
CP [ROW [1], col3] \leftarrow \max ;
\]

\[
\text{minimum} (ROW [1], col3, min3) ;
\]

\[
\text{until} (\neg\text{CAP}[col3] \text{ mt}) \text{ or } (col3=0) ;
\]

49
for \( l = 0 \) to \( m \)

\[
CP \text{[ROW [1],t]} \leftarrow CI \text{[ROW [1],t]} ;
\]

next \( t \);

end;

end;

INC \text{[ROW [1], col3]} \leftarrow 1 ;

CP \text{[ROW [1], col3]} \leftarrow \text{max} ;

CI \text{[ROW [1], col3]} \leftarrow \text{max} ;

if (col3 < 0) then CAP1 [col3] \leftarrow \text{CAP1 [col3]} + 1;

end;

end;

// Description of module initialize used in module reallocate //
initialize:

begin

for \( s = 1 \) to \( n \)

\[
\text{DIFF [s]} \leftarrow \text{max} ;
\]

\[
\text{COLUMN [s]} \leftarrow 0 ;
\]

next \( s \);

\[
i \leftarrow i - 1 ;
\]

end;

50
// Description of module checl used in module reallocate //
checl (1, rw, cw, pen, penal); begin
  while (i ≥ 0) do pass (1); findnewp;
  if (flag) then begin
    if (DIFF [p] * pen) then begin
      begin
        rw ← rowindex[1];
        cw ← colindex[1];
        penal ← DIFF [p];
      end;
    end;
  end;
/*
Description of the module pass used in module checl */
pass (col); begin
  if (INC [ROW [1], COL [1]] = 1) then begin
    repeat
      minimum (ROW [1], COLUMN [1], min4);
      CP [ROW [1], COLUMN [1]] ← max;
      until ( (CAP1 [COLUMN [1]] < m) or
        (COLUMN [1] = 0) );
  end;
for \( t = 0 \) to \( m \)

\[
\text{CP} \left[ \text{ROW} \left[ i \right], t \right] \leftarrow \text{CH} \left[ \text{ROW} \left[ i \right], t \right] ;
\]

next \( t \) :

\[
\text{DIFF} \left[ i \right] \leftarrow \text{MIN} \left( 4 - \text{CM} \left[ \text{ROW} \left[ i \right], \text{COL} \left[ i \right] \right] \right) ;
\]

end :

\[
i \leftarrow i - 1 ;
\]

end :

// Description of the module findnewp used in the module thernl

findnewp :

begin

\[
i \leftarrow 1 + 1 ;
\]

flag \leftarrow \text{false} ;

if ( \( i \) \neq 1 ) then

begin

\[
\text{minm} \leftarrow \text{max} ;
\]

for \( s = 1 \) to \((i-1)\)

if ( \( \text{DIFF} \left[ s \right] \), \( \text{minm} \) ) then

begin

\[
\text{minm} \leftarrow \text{DIFF} \left[ s \right] ;
\]

\[
p \leftarrow s ;
\]

flag \leftarrow \text{true} ;

\[
\text{rowindex} \leftarrow \text{ROW} \left[ s \right] ;
\]

\[
\text{colindex} \leftarrow \text{COLUMN} \left[ s \right] ;
\]

end :

next \( s \) ;

end :

end ;
4.3 Analysis of the algorithm:

In order to analyse the algorithm in terms of storage space requirements and computation time requirements, it may be noted that the algorithm basically works in the following steps - stepwise execution of the algorithm is being explained in section 4.4 with the help of an example.

Step 1:
Read the variables $n$, $m$ and $m_1$; and the matrix:

$CM[i,j] ; i = 1,2,\ldots,n$ and $j = 0,1,2,\ldots,m$
Step 2:

Compute penalty for each row, and store in the array VALUE [1:n]

Step 3:

Sort the values of the penalties in the descending order, and accordingly rearrange the rows of the cost matrix so that those rows can subsequently be traversed in accordance with the descending order of the penalties.

Step 4:

Next, starting with the first row of the rearranged cost matrix, repeat for each of the rows:

\[
\text{if (the first minimum occurs at the 0th terminal controller position w.r.t. the ith row)}
\]

\[
\text{then allocate the ith terminal to the 0th terminal controller since the 0th terminal controller is the central one with infinite capacity}
\]

\[
\text{else test whether the total number of assignment of terminals to that terminal controller exceeds the capacity m}
\]

\[
\text{if (it does not exceed)}
\]

\[
\text{then}
\]

\[
\text{allocate the ith terminal to that terminal controller}
\]

\[
\text{else}
\]
If (the second minimum occurs at the 0th terminal controller position)

then
allocate 1th terminal to 0th terminal controller

else

  test whether the total number of assignment of terminals to that terminal controller exceeds M;
  if (it does not exceed)
  then
allocate the 1th terminal to that terminal controller
  else
re-allocate (as mentioned below).

Re-allocation:

Let there be a necessity of re-allocation for the 1th terminal.

Then the rows numbered as 1-1, 1-2, ... , 2, 1 are traversed to find whether there is any improvement with respect to cost if the allocation in 1th row is made to its 1st minimum position (say, s) as well as a previous assignment in 1st minimum position (i.e., s) is altered to the next available minimum position.
The same procedure is repeated for allocating the ith row to its 2nd minimum position as well as altering a previous assignment in 2nd minimum position to the next available minimum position.

If there is no improvement, the ith row is assigned to its next available minimum position, and if there is any improvement, the best of the two alternatives is considered for reallocation.

The complexity analysis of the algorithm in terms of storage space requirements and computation time requirements are now being discussed below.

4.3.1 Storage Requirements

In this context, we consider only the arrays used in the algorithm, since these are the main contributors towards the total space requirements. Individual requirements are being mentioned in Table 4.1.

Table 4.1

<table>
<thead>
<tr>
<th>Array</th>
<th>Dimension</th>
<th>Space Required (in units of storage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>1:n,0:m</td>
<td>(m+1)n</td>
</tr>
<tr>
<td>ROW</td>
<td>1:n</td>
<td>n</td>
</tr>
<tr>
<td>COL1</td>
<td>1:n</td>
<td>n</td>
</tr>
<tr>
<td>COL2</td>
<td>1:n</td>
<td>n</td>
</tr>
<tr>
<td>VALUE</td>
<td>1:n</td>
<td>n</td>
</tr>
<tr>
<td>DIFF</td>
<td>1:n</td>
<td>n</td>
</tr>
<tr>
<td>NC</td>
<td>1:n,0:m</td>
<td>(m+1)n</td>
</tr>
</tbody>
</table>
Hence, the total storage requirements are:

\[ m^n + n + n + n + n + n + n + n + n + n \]

In practical situations, the number of terminals are much more compared to the number of terminal controllers, i.e. \( m > n \). Thus, the storage requirements of the algorithm have an upper bound of \( O(n^2) \).

Table 4.1 depicts only major storage space required by the algorithm. Any particular implementation of the algorithm may require some additional memory space, depending on the methodology used for implementation. However, the order of the storage space requirements is unaffected by the differences in implementation.

4.5.2 Computation time requirements:

The computation time requirements of the algorithm may be analysed stage-wise, as discussed below:

Calculation of penalties:

The term 'penalty' denotes the difference between two minimums, and the execution time required for finding minimum of \( m \) elements is \( O(m) \). Also, since penalty is to be calculated for each of the \( n \) terminals, the execution time required at this stage is \( n \times O(m) \).

Sorting the penalties:

The execution time required to sort \( n \) number of penalties in the descending order is \( O(n \log n) \).

Assignment of terminals:

In the best-case situation, backtracking is not required at all, and hence computation time requirement for scanning all the \( n \) terminals in one pass is \( O(n) \).

57
If backtracking is required for the $i$th terminal, each of the $m$ terminal controllers is to be searched, the associated computation time requirement being $O(m)$. Thus, in the worst case situation, computation time requirement at this stage is

$$\sum_{i=1}^{n} O(m) = O(m) \times O(n^2)$$

Now, considering the execution times of all the three stages and taking care of the fact that in practical situations $m \leq n$, it follows that the worst-case execution time requirement of the algorithm is

$$n \times O(m) + O(n \log n) + O(m) \times O(n^2) \quad \text{i.e.} \quad O(n^3).$$

4.4 An Example:

Step 1 (Inputs):

- Number of terminals ($n$) = 7
- Number of terminal controllers ($m$) = 3
- Capacity of each terminal controller ($mt$) = 2
  
  [Capacity of the central terminal controller is infinite]

The cost matrix $CM[1:7, 0:3]$ is as follows:

<table>
<thead>
<tr>
<th>Term. controller</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term. 1</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>8</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Step 1:

Contents of
row [1:n] : 1 2 3 4 5 6 7
value [1:n] : 3 3 4 5 1 1 1
col1 [1:n] : 1 1 1 1 2 3 0
col2 [1:n] : 2 2 3 2 1 0 1

Step 2:

Contents of
value [1:n] : 5 4 3 3 1 1 1
row [1:n] : 4 3 1 2 5 6 7
col1 [1:n] : 1 1 1 1 2 3 0
col2 [1:n] : 2 3 2 2 1 0 1

Subsequently we shall across the cost matrix in descending order of the values in the array value [1:n], and, hence logically the cost matrix now becomes:

<table>
<thead>
<tr>
<th>Term.</th>
<th>Row</th>
<th>Term. controller</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>0 1 2 3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8 3 9 7</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>8 3 6 11</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>14 3 6 7</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9 3 2 4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6 8 9 5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4 5 7 9</td>
<td>1</td>
</tr>
</tbody>
</table>
Step 4:

1st iteration:
row no. = 1; Term. = 4; col1[1] = 1; col2[1] = 2
No. of assignment to col1[j] = 0
and this is less than ml.
Hence, assign row 1 to terminal controller 1 and
increase the no. of assignment to terminal controller 1 by 1.

2nd iteration:
row no. = 2; Term. = 3; col1[2] = 1; col2[2] = 3
No. of assignment to col1[2] = 1
and this is less than ml.
Hence, assign row 2 to terminal controller 1 and
increase the no. of assignment to terminal controller 1 by 1.

3rd iteration:
row no. = 3; Term. = 1; col1[3] = 1; col2[3] = 2
No. of assignment to col1[3] = 2
and this is equal to mt.
No. of assignment to col2[3] = 0
and this is less than mt.
Hence, assign row 3 to terminal controller 2 and
increase the no. of assignment to terminal controller 2 by 1.
4th Iteration:
No. of assignment to col1[4] = 2 and
this is equal to mt.
No. of assignment to col2[4] = 1 and
this is less than mt.
Hence, assign row 4 to terminal controller 2 and
increase the no. of assignment to terminal
controller 2 by 1.

5th Iteration:
No. of assignment to col1[5] = 2 and
this is equal to mt.
No. of assignment to col2[5] = 2 and
this is equal to mt.
Hence, there is a necessity of re-allocation.
The present state of allocation is clearly
understood from the following cost matrix CM where
circled entries indicate respective allocations.

<table>
<thead>
<tr>
<th>Term.</th>
<th>Row</th>
<th>Term. controller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>
Re-allocation:

row no : 5 : nextmin : 4 (for term. controller 3)

a) pen1 = (nextmin - 1st minimum value) in 5th row
   = 4 - 2 = 2

The DIFF array is initialized as:

```
DIFF [1:n] : MAX MAX MAX MAX MAX MAX MAX MAX
```

where MAX is a very large value.

Different states of l, nextmin, DIFF [1] etc. during execution are being shown below:

<table>
<thead>
<tr>
<th>Term</th>
<th>Controller</th>
<th>nextmin</th>
<th>DIFF [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2nd terminal controller</td>
<td>MAX</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>MAX</td>
</tr>
</tbody>
</table>

Hence, new values of the DIFF array are:

```
DIFF [1:n] : MAX MAX 3 1 MAX MAX MAX MAX
```

Minimum element of this array is:

```
```

Now, DIFF [p] = pen1 = -1, i.e. DIFF [p] = pen1 and hence, rw1 = 2 and cw1 = 3.

b) pen2 = (nextmin - 2nd minimum value) in 5th row
   = 4 - 2 = 2

The DIFF array is initialized as:

```
DIFF [1:n] : MAX MAX MAX MAX MAX MAX MAX MAX
```
The different states of \( I, \text{nextmin}, \text{DIFF} [i] \) etc. during execution are shown below:

<table>
<thead>
<tr>
<th>( L )</th>
<th>( \text{nextmin} )</th>
<th>( \text{DIFF} [i] )</th>
<th>Term controller corresponding to nextmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>No previous assignment in terminal controller</td>
<td>MAX</td>
<td></td>
</tr>
</tbody>
</table>
| 3     | "              | MAX              | ----
| 2     | 7              | 4                | 3 |
| 1     | 16             | 8                | 3 |

Hence, new values of the DIFF array are:

\[
\text{DIFF} [1:n] : 8 \quad 4 \quad \text{MAX} \quad \text{MAX} \quad \text{MAX} \quad \text{MAX} \quad \text{MAX}
\]

Now, \( \text{DIFF} \ [p2] = 4 \) and

\( \text{DIFF} \ [p2] - \text{pen}2 = 4 - 1 = 3 \)

Now, since \( (\text{DIFF} \ [p1] - \text{pen}1) \quad (\text{DIFF} \ [p2] - \text{pen}2) \)

and \( \text{DIFF} \ [p1] \cdot \text{pen}1 \),

allocate to row no 4 (i.e. term 2) the terminal controller no 3; cancel the previous allocation in row no 4 (i.e. term 2) and terminal controller 2; and allocate to row no 5 (i.e. term 5) the terminal controller 2.
The present status of assignments in the cost matrix is being shown below, with the circled entries representing the allocated elements:

<table>
<thead>
<tr>
<th>Term.</th>
<th>Row</th>
<th>Term. controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

**6th iteration:**

Row no: 6; Term: 6; col[6]: 7; col[2][6]: 0

No. of assignment to col[6] = 1, and this is less than mt.

Hence, assign row no 6 to the term. controller 7;
and increase the total no. of assignments to the term. controller 7 by 1.

**7th iteration:**

Row no: 7; Term: 7; col[7]: 0; col[2][7]: 1

Thus, the first minimum occurs with the 0th terminal controller, i.e. with the terminal controller having infinite capacity.

Hence, assign row no. 7 (i.e. terminal no. 7) to the 0th terminal controller.
Step 5:

Print the incidence matrix representing the allocated elements (obtained as above) as follows:

\[
\begin{array}{c|cccc}
\text{Term.} & 0 & 1 & 2 & 3 \\
\hline
1 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 \\
6 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 \\
\end{array}
\]

4.5 Implementation results:

The algorithm has been implemented in MicroVAX - 11 using PASCAL and it has extensively been tested for various sets of input data. The execution times taken by the algorithm (averaged over 10 runs for each set of data) for various sets of input data, are being listed in Table - 4.1:
4.6 Conclusion:

In this chapter, we have considered the terminal layout problem which deals with the problem of connecting terminals to terminal controllers. Currently available algorithms for generating such a topology primarily aims at obtaining a minimal spanning tree. Thus, though a minimal cost solution is achieved, yet the inherent problems related to modularity in design and controller capacity are not included in the design. The algorithm proposed and analysed in this chapter aims at producing a design taking into account such problems often encountered in practice. Herein, instead of generating a minimal cost spanning tree, a minimal cost tree satisfying the constraint on upper bound on the number of terminals connected to the controller is obtained. The method has been tested with problems of differing magnitudes and the associated data on execution time have been included.