CHAPTER 31

GRAPHS

GRAPHS PRELIMINARIES (Reading)
APPLICATION OF GRAPHS
SHORTEST PATH ALGORITHM
FLOW PROBLEM (Reading)
IMPROVING A FLOW FUNCTION
ALGORITHM TO PRODUCE AN OPTIMAL FLOW FUNCTION (Reading)
C ROUTINE FOR MAXFLOW (Reading)
LINKED REPRESENTATION OF GRAPHS
GRAPH TRAVERSALS
MINIMUM SPANNING TREES
PRIM'S ALGORITHM FORMINIMUM SPANNING TREES

CHAPTER – 31

GRAPHS

GRAPHS PRELIMINARIES

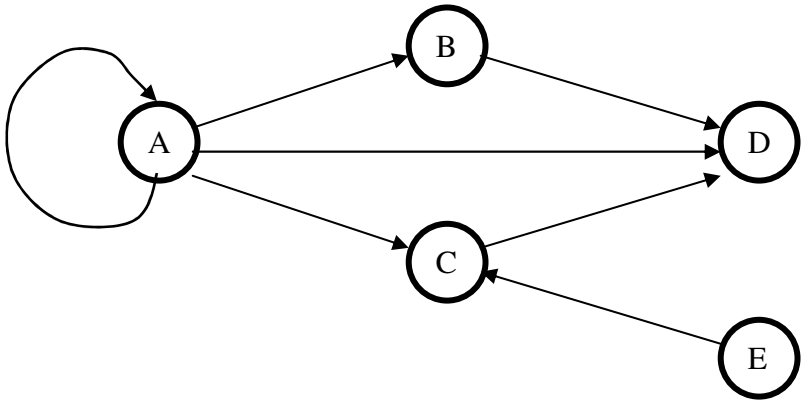
A ***graph*** consists of a set of ***nodes*** (***vertices***) and a set of ***arcs*** (***edges***). A pair of nodes specifies each arc in a graph. The set of nodes is *{A, B, C, D, E, F, G, H}*, and the set of arcs is *{(A,B), (A,D), (A,C), (C,D), (C,F), (E,G), (A,A)}*. If the pairs of nodes that make up the arcs are ordered pairs, the **graph** is said to be a ***directed graph*** (***digraph***). The arrow between nodes represents arcs. The head of each arrow represents the second node in the ordered pair.

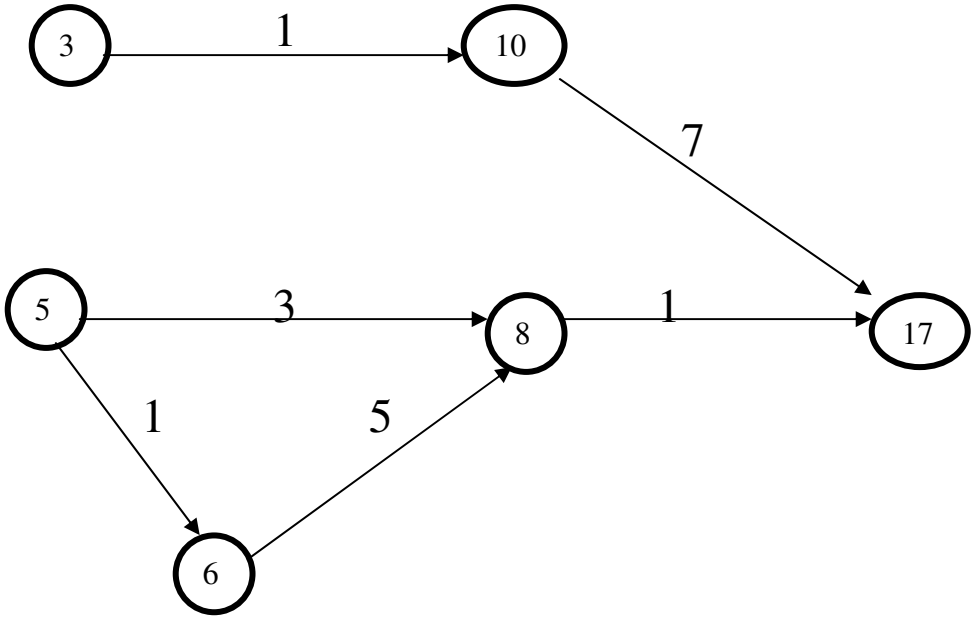
A node *n* is incident to an arc ***x*** if ***n*** is one of the two nodes in the ordered pair of nodes that constitute *x*. The ***degree*** of a node is the number of arcs incident to it. The ***indegree*** of a node *n* is the number of arcs that have *n* as the head, and ***outdegree*** of *n* is the number of arcs that have *n* as the tail. A node *n* is ***adjacent*** to a node m if there is an arc from *m* to *n*. If *n* is **adjacent** to *m*, *n* is ***successor*** of *m*, and *m* a ***predecessor*** of *n*.

A ***relation R*** on a set ***A*** is a set of ordered pairs of elements of *A*. If <*x,y*> is a member of a relation *R*, *x* is said to be ***related*** to *y* in *R*. The above relation *R* may be described by saying that *x* is related to *y* if *x* is less than *y* and the remainder obtained by dividing y by *x* is odd. A relation may be represented by a **graph** in which the nodes represent the underlying set and the arcs represent the ordered pairs of the relation. Such a **graph**, in which a number is associated with each arc, is called a ***weighted graph*** or a network. The number associated with an arc is called its ***weight.***

A ***path of length k*** from node *a* to node *b* is defined as a sequence of *k + 1* nodes *n1, n2, …nk, nk+1* such that *n1 = a*, *nk+1 = b* and adjacent (*ni, ni+1*) is true for all *i* between *1* and *k*. If for some integer *k*, a ***path*** of length *k* exists between *a* and *b*, there is a path from *a* to *b*. A path from a

node to itself is called a ***cycle***. If a graph contains a cycle, it is ***cyclic***; otherwise it is ***acyclic***. A directed **acyclic** graph is called ***dag***.





FLOW PROBLEM

Assume a water pipe system with each arc representing a pipe and the number above each arc represents the capacity of that pipe in gallons per minute. The nodes represent point at which pipes are joined and water is

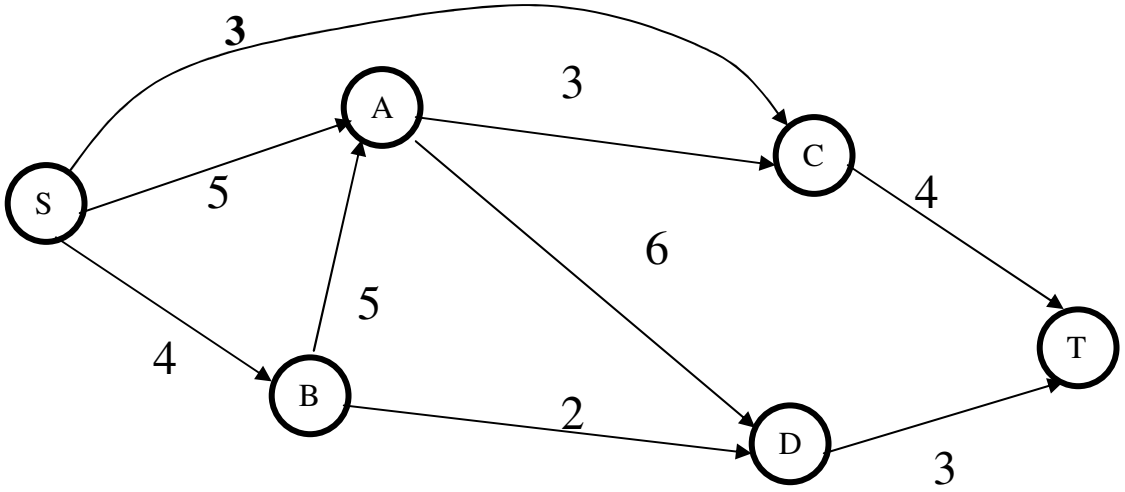
transferred from one pipe to another. Two nodes *S* and *T*, are designated as a ***source*** of water and a ***user*** of water, respectively. We would like to maximize the amount of water flowing from the source to the user. Although the source may be able to produce water at prodigious rate and the user may be able to consume water at a comparable rate

Define a ***capacity function***, ***c***(a, b) where *a* and *b* are nodes. If ***isadjacent***(*a, b*) is true, ***c***(*a, b*) is the capacity of the pipe from *a* to *b*. If there is no pipe from a to b, ***c***(a,b) = 0. At any point in the operation of the system, a given amount of water flows through each pipe. Define a ***flow function***, ***f***(*a, b*), where *a* and *b* are nodes, as *0* if *b* is not adjacent to *a*, and as the amount of water flowing through the pipe from *a* to *b* otherwise. Define the ***inflow*** of a node *x* as the total flow entering *x* and the ***outflow*** as the total flow leaving *x*. The foregoing conditions may be written as:

outflow (S) = ***inflow*** (T) = v

inflow (x) = ***outflow*** (x) for all x != S,T

No node other than *S* can produce water and no node other than *T* can absorb water. Thus the amount of water leaving any node other than *S* or *T* is equal to the amount of water entering the node.

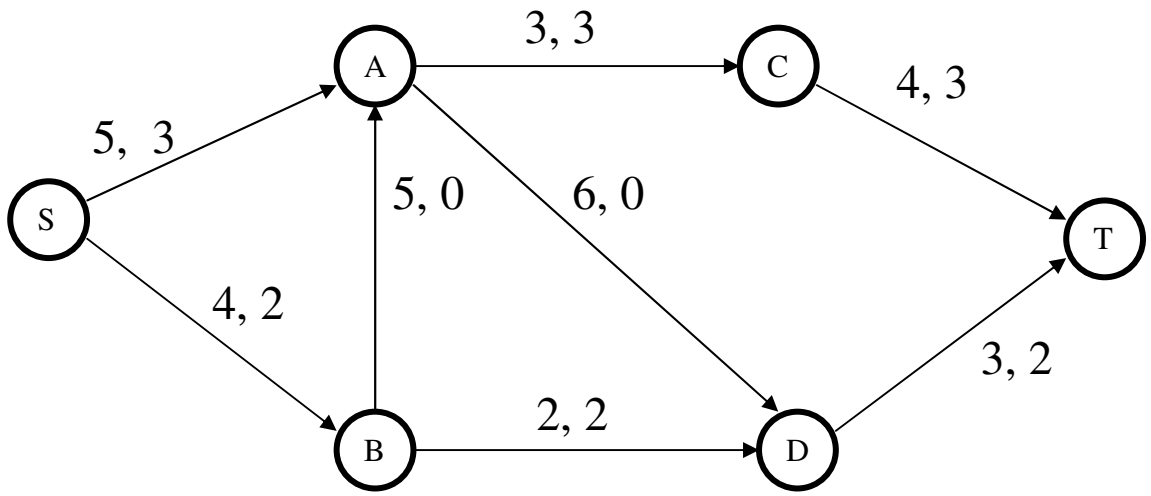


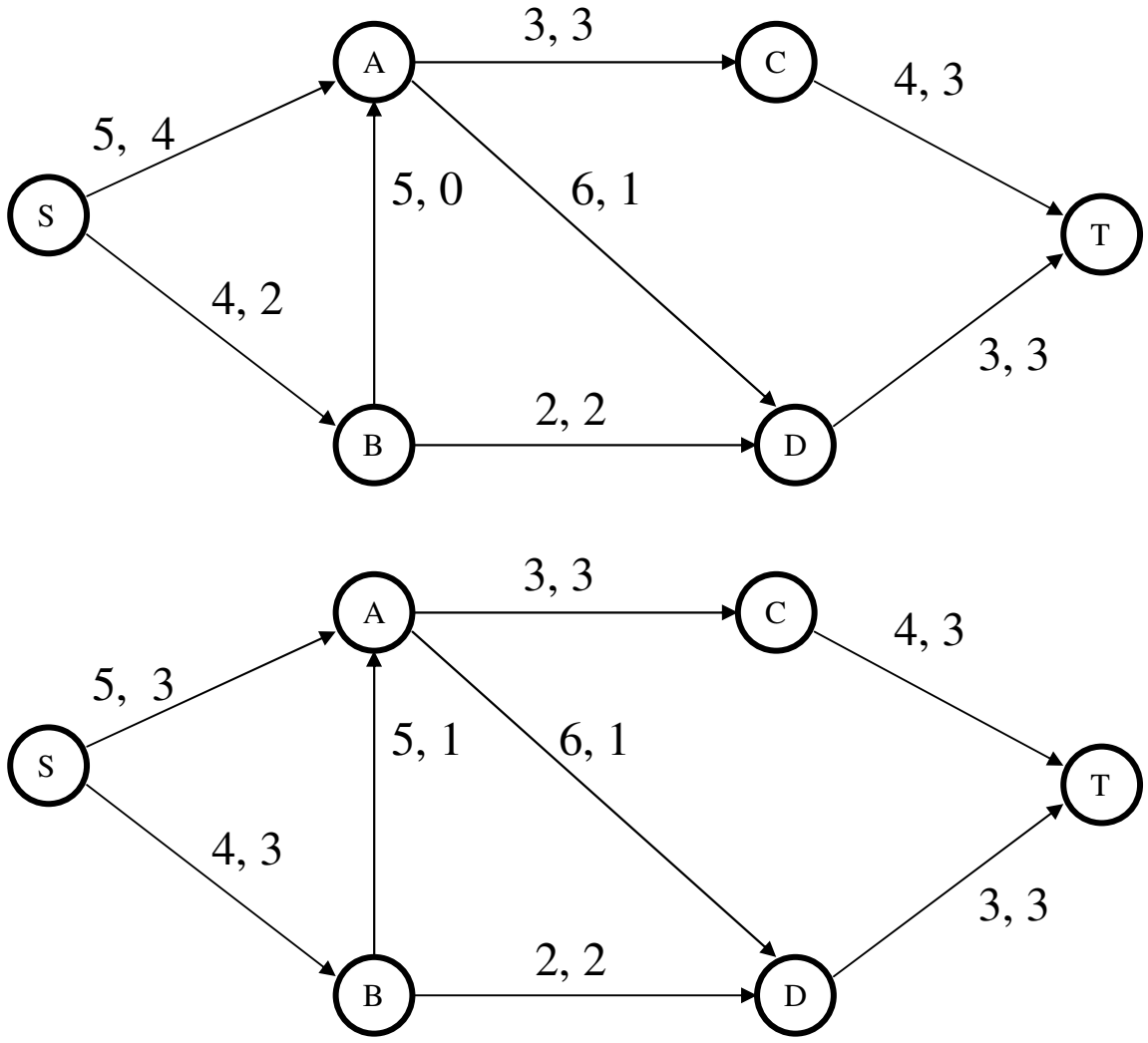
Several flow functions may exist for a given graph and **capacity functions.** The intention is to find a flow function that maximizes the value of *v*, which is the amount of water going from *S* to *T*. Such a flow function is called ***optimal.***

One valid **flow function** can be achieved by setting ***f*** (*a, b*) to *0* for all nodes *a* and *b*. Of course this **flow function** is least optimal, since no water flows from *S* to *T*. Given a flow function, it can be improved so that the flow from *S* to *T* is increased. However, the improved version must satisfy all the conditions for a valid **flow function**. In particular, if the flow entering any node is increased or decreased, the flow leaving that node must be increased or decreased correspondingly. The strategy for producing an optimal **flow function** is to begin with the zero **flow function** and to improve upon it successively until an optimal **flow function** is produced.

IMPROVING A FLOW FUNCTION

Given a **flow function** ***f*** there are two ways to improve upon it. One way consists of finding a path *S = x1, x2, …., xn = T* from *S* to *T* such that the flow along arc in the path is strictly less than the capacity. The flow can be increased on each arc in such a path by the minimum value of ***c***(*xk-1, xk*) - ***f***(*xk-1, xk*) for all *k* between *1* and *n – 1* so that when the flow has been increased along the entire path there is at least one arc <*xk-1, xk*> in the path for which ***f***(*xk-1, xk*) = ***c***(*xk-1, xk*) and through which the flow may not be increased.





There are two paths from *S* to *T* with positive flow (*S, A, C, T*) and (*S, B, D, T*). However each of these paths contains one *arc*<*A, C*> and <*B, D*> in which the flow equals the capacity. Thus the flow along these paths may not be improved. However, the paths and (*S, A, D, T*) is such that the capacity of each arc in the path is greater than its current flow. The maximum amount by which the flow can be increased along this path is *1*, since the flow along *arc* <*D, T*> cannot exceed *3*. The total flow from *S* to *T* has been increased from *5* to *6*.

Define a ***semipath*** from *S* to *T* as a sequence of nodes *S = x1, x2, xn = T* such that, for all *0 < i < = n – 1*, either <*xi-1, xi*> or <*xi, xi-1*> is an arc. Using the foregoing technique, we may describe an algorithm to discover

a **semipath** from *S* to *T* such that the flow to each node in the **semipath** may be increased. This is done by building upon already discovered partial **semipaths** from *S*. If the last node in a discovered **partial semipath** from *S* is *a*, the algorithm considers extending it to any node *b* such that either <*a, b*> or <*b, a*> is an arc. The ***partial semipath*** is extended to *b* only if the extension can be made in such a way that the inflow to *b* can be increased. Once a **partial semipath** has been extended to *a* node *b*, that node is removed from consideration as an extension of some other **partial semipath**. The algorithm of course keeps track of the amount by which the inflow to *b* may be increased and whether its increase is due to consideration of the arc <*a, b*> or <*b, a*>.

This process continues until some **partial semipath** from *S* has been completed by extending it to *T*. The algorithm then proceeds backward along the **semipath** adjusting all flows until *S* is reached. The entire process is then repeated in an attempt to discover yet another such semipath from *S* to *T*. When no **partial semipath** may be successfully extended, the flow cannot be increased and the existing **flow is optimal**.

ALGORITHM TO PRODUCE AN OPTIMUM FLOW FUNCTION

Input is a weighted graph (an adjacency matrix and a capacity matrix) with a source *S* and sink *T*.

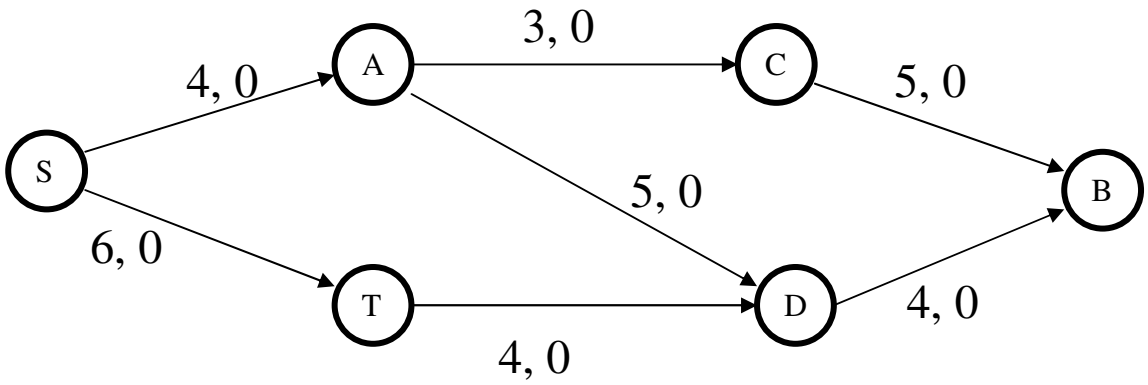
```
Initialize the flow function to 0 at each arc;
canimprove = TRUE;
do
{
    find a semipath from S to T that increases the flow to T by x > 0;
    if (a semipath cannot be found)
                        canimprove = FALSE;
    else
        increase the flow to each node in the semipath by x;
}  while (canimprove == TRUE);
```

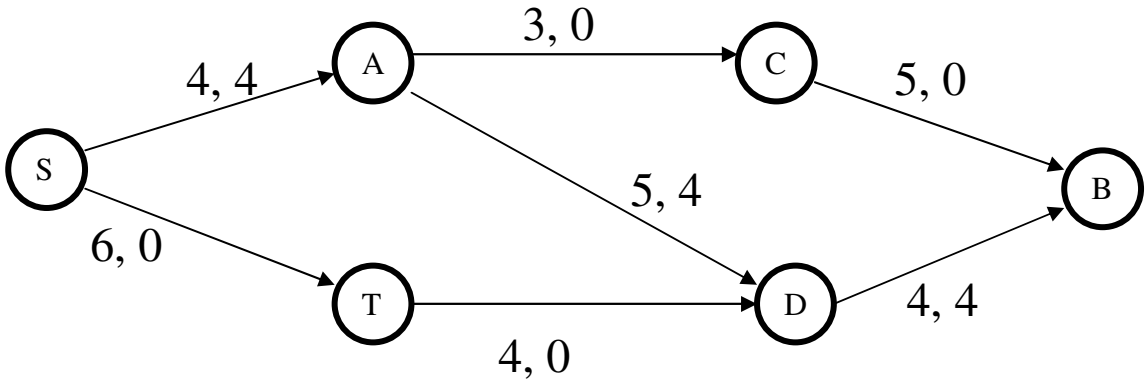
Once a node has been placed on a partial **semipath**, it can no longer be used to extend a different **semipath**. The algorithm that attempts to find a **semipath** from *S* to *T* uses an array of flags **onpath** such that *onpath[node]* indicates whether or not node is on some semipath. It also needs an indication of which nodes are at the ends of partial **semipaths** so that such partial **semipaths** can be extended by adding adjacent nodes. *endpath[node]* indicates whether or not node is at the end of a partial **semipath**. For each node on a **semipath** the algorithm must keep track of what node precedes it on that semipath and the direction of the arc. *precede[node]* points to the node that precedes node on its semipath, and *forward[node]* has the value ***TRUE*** if and only if the arc is from *precede[node]* to node**.** *improve[node]* indicates the amount by which the flow to node may be increased along its semipath. ***c****[a][b]* is the capacity of the pipe from *a* to *b* and ***f*** *[a][b]* is the current flow from *a* to *b*.

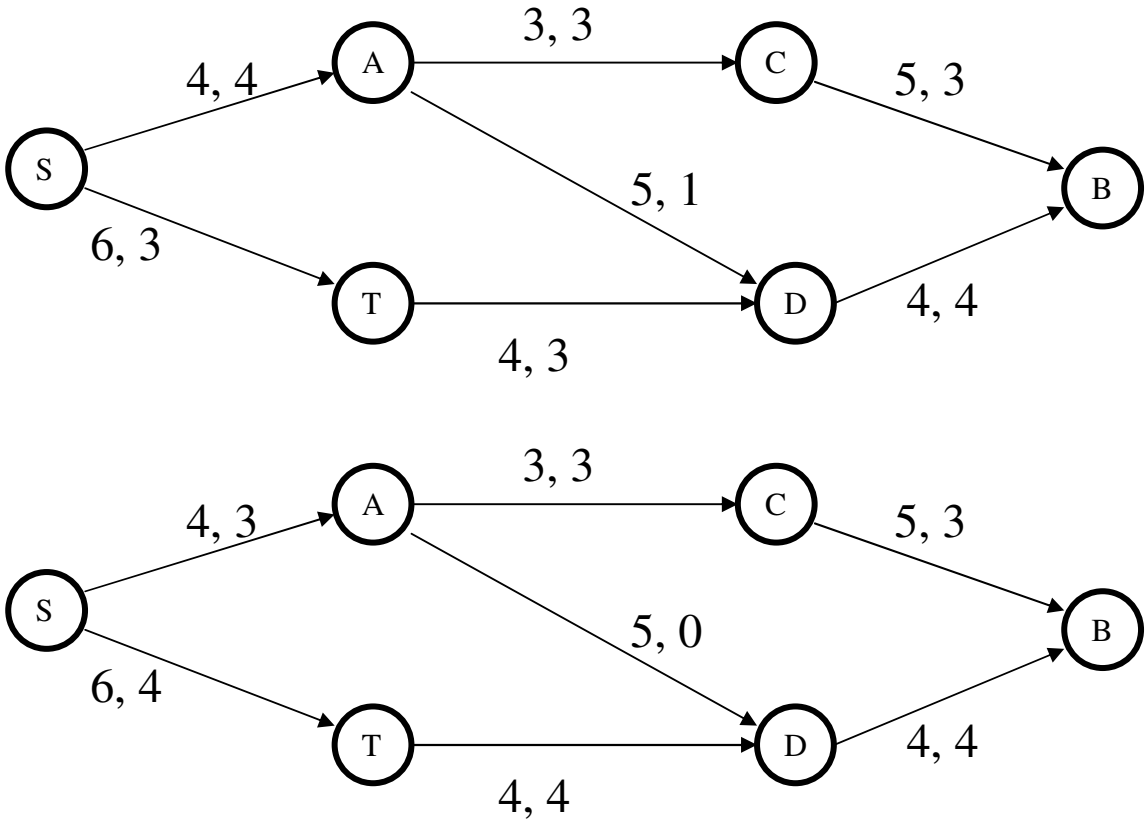
algorithm to find a semipath from S to T along which flow is increased:

```
set endpath[node], onpath[node] to FALSE for all nodes;
endpath[S]  =  TRUE;
onpath  [S]  =  TRUE;
/* compute maximum flow from S that pipes can carry */
improve[S]  =  sum of c[S][node] over all nodes node;
while ( (onpath[T] == FALSE) &&
          (there exists a node i such that endpath[nd] == TRUE) )
{
   endpath [nd] = FALSE;
   while ( there exists a node i such that
             (onpath[i] == FALSE) && (adjacent (nd, i) == TRUE)
             && (f[nd] < c[nf][i]) )
   {  /* flow from nd to i may be increased , place i on semipath */
      onpath   [i]   =  TRUE;
      endpath [i]   =  TRUE;
      precede [i]   =  TRUE;
      forward [i]   =  TRUE;
```

```
            x = c[nd][i] – f[nd][i];
            improve [i]  =  (improve[nd] < x) ? improve[nd] : x;
        }       /* end of while there exists  -- 1  */
        while ( there exists a node i such that
(onpath[i] == FALSE) && (adjacent (i, nd) == TRUE) && (f[nd] > 0) )
        {  /*flow from i to nd may be decreased, place i on semipath */
            onpath   [i]   =  TRUE;
            endpath [i]   =  TRUE;
            precede [i]   =  nd;
            forward[i] = FALSE;
           improve[i] = (improve[nd] < f[i][nd] ? improve[nd] : f[i][nd];
        }       /* end of while there exists  -- 2  */
    }   /* while ( (onpath[T] == FALSE)  * /
    if (onpath(T) == TRUE
        semipath from S to T is found;
    else
        the flow is already optimum;
```







once a semipath is found, algorithm to increase flow along semipath

Once a semipath from *S* to *T* has been found, the *flow* may be increased along the semipath by the following algorithm:

```
x  = improve[T];
nd = T;
while (nd != S)
{
    pred = precede[nd];
    (forward[nd] == TRUE) ? (f[pred, nd] += x)  : (f[nd, pred] -= x);
    nd = pred;
} /*  end while  */
```

C ROUTINE FOR MAXFLOW

C routine ***maxflow*** (*cap, s, t, flow, totflow*) has cap as an input parameter representing a capacity function defined on a weighted graph. ***S*** and ***T*** are input parameters representing the source and sink, flow is an output parameter representing the maximum ***flow*** function, and totflow is the amount of ***flow*** from *s* to *t* under the flow function ***flow***. ***any*** is a function that accepts an array of logical values and returns ***TRUE*** if ***any*** element of the array is ***TRUE***. If none of the elements of the array is ***TRUE***, ***any*** returns ***FALSE***.

```
#define MAXNODES   50
#define INFINITY   ….

int  any (int []);

void maxflow ( int cap[] [MAXNODES], int s, int t,
                              int flow [][MAXNODES], int *ptotflow)
{
    int pred, nd, i, x, onpath[MAXNODES];
    int precede[MAXNODES], improve[MAXNODES];
    int endpath[MAXNODES], forward[MAXNODES];

    for (nd = 0;  nd < MAXNODES;  ++nd)
        for (i = 0; nd < MAXNODES; ++i)
            flow[nd][i] = 0;
    *ptotflow = 0;
    do
    {
        /* attempt to find a semipath from s to t  */
        for (nd = 0;  nd < MAXNODES;  ++nd)
        {
            endpath[nd] = FALSE;
             onpath[nd] = FALSE;
```

```
 }        /* end of for  */
 endpath[s]   = TRUE;
   onpath[s]   = TRUE;
 improve[s]   = INFINITY;
/* we assume that s can provide infinite flow  */
while ( (onpath[t] == FALSE && (any(endpath) == TRUE) )
{
     /* attempt to extend an existing path  */
      for (nd = 0; endpath[nd] == FALSE;  nd++)
           ;
       endpath[nd] = FALSE;
       for (i = 0;  i < MAXNODES;  ++i)
       {
            if ((flow[nd][i] < cap[nd][i])&& (onpath[i] == FALSE) )
            {
                 onpath[i]   = TRUE;
                endpath[i]   = TRUE;
                 precede[i]  = nd;
                forward[nd] = TRUE;
                x                = cap[nd][i] – flow[nd][i];
                improve[i]   = (improve[nd] < x) ? improve[nd] : x;
            }   /* end of if ((flow[nd][i]     */
           if ((flow[i][nd] > 0) && (onpath[i] == FALSE) )
           {
                onpath[i]   = TRUE;
                endpath[i]   = TRUE;
                precede[i]   = nd;
                forward[nd] = FALSE;
                improve[i]   = (improve[nd] < flow[i][nd]) ?
                                    improve[nd] :  flow[i][nd];
           }   /* end of if ((flow[nd][i]     */
     }    /* end of for   */
}   /* end of while  */
```

```
        if (onpath[t] == TRUE)
           {   /* flow on semipath to t can be increased  */
               x              = improve[t];
               *ptotflow += x;
               nd             = t;
               while (nd != s)
               {
                   /* travel back along path  */
                   pred = precede[nd];
                   /* increase or decrease flow from pred  */
                   forward[pred] == TRUE) ? (flow[pred][nd] += x) :
                                            (flow[pred][nd] += x);
                   nd = pred;
               }     /* end of while (nd != s)     */
           }         /* if (onpath[t]  == TRUE                */
        }  while (onpath[t] == TRUE);          /* end of do  while   */
}                     /* end of maxflow             */
```