Mod 6

Algorithms

- Used to solving a certain mathematical problems.
- Consists of a set of instructions that when followed step by step will lead to the solutions of the problem.
- Every step in algorithm must be precisely and unambiguously defined
- Algorithm must terminate after having solved the given problem in a finite number of steps.

An algorithm can be expressed in different forms: (1) the steps may be written in English; (2) it may be in the form of a computer program written in complete detail in the language understandable by the machine in use; or (3) the algorithm may be expressed in a form between these two extremes, such as a flow chart.

Usually, an algorithm is first expressed in ordinary language, then converted into a flow chart, and finally written in the detailed and precise language so that a machine can execute it.

Efficiency of algorithms

An algorithm must not only do what it is supposed to do, but must do it efficiently. The two main criteria for efficiency of an algorithm are the memory and computation-time requirements as a function of the size of the input. In our case the input is a graph, and its size is the number of vertices, n, and the number of edges, e.

Input : Computer representation of graphs

- An algorithm start working based on some input values.
- Here, input will be one or more graphs (or digraphs).

Adjacency matrix

- The most popular form in which a graph or digraph is fed to a computer is its **adjacency matrix.**
- Each n vertices in G is assigned a number and n by n binary matrix X(G) is created.
- X(G) is representing G during input, storage & output.
- X(G) contains n^2 entries , which are 1 or 0.
- X(G) matrix needs n^2 bits of computer memory.

- Bits can be packed into words
- Let w be word length & n number of vertices in G
- Each row of adjacency matrix may be written as a sequence of n bits in $\lceil n/w \rceil$ machine words
- Therefore, the number of words used to store adjacency matrix is $n\lceil n/w \rceil$.

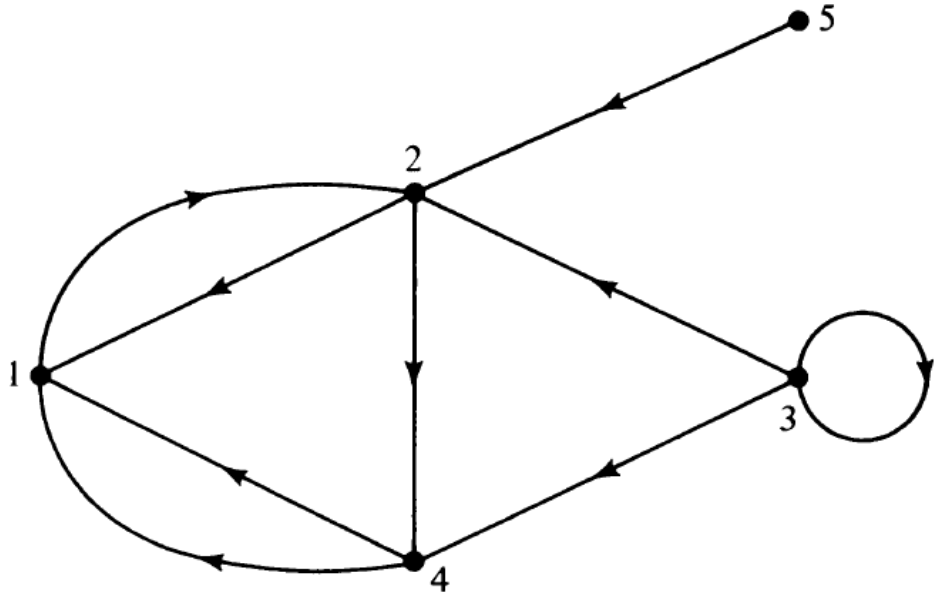
- Adjacency matrix of a undirected graph is symmetric, so storing upper triangle is sufficient.
- This requires n(n-1) bits for storage. Thus space complexity reduced.
- But it costs in increased complexity in computation time.

Incidence matrix

- Since adjacency matrix is defined for graphs without parallel edges, we use **incidence matrix** for storing and manipulation of a graph.
- Incidence matrix needs **n*e** bits for storage which will be more than n^2, because number of edges can be greater than number of vertices.

Edge listing

- This representation used is to list all edges of the graph as vertex pairs, having numbered n vertices in order.
- (1, 2), (2, 1), (2, 4), (3, 2), (3, 3), (3, 4), (4, 1), (4, 1), (5, 2).



- This representation can include both self loops and parallel edges.
- The number of bits needed to store each vertex *b :*

$$2^{b-1} < n \leq 2^b.$$

- And since each edge e requires storing two such numbers, total storage required is :

2e*b bits

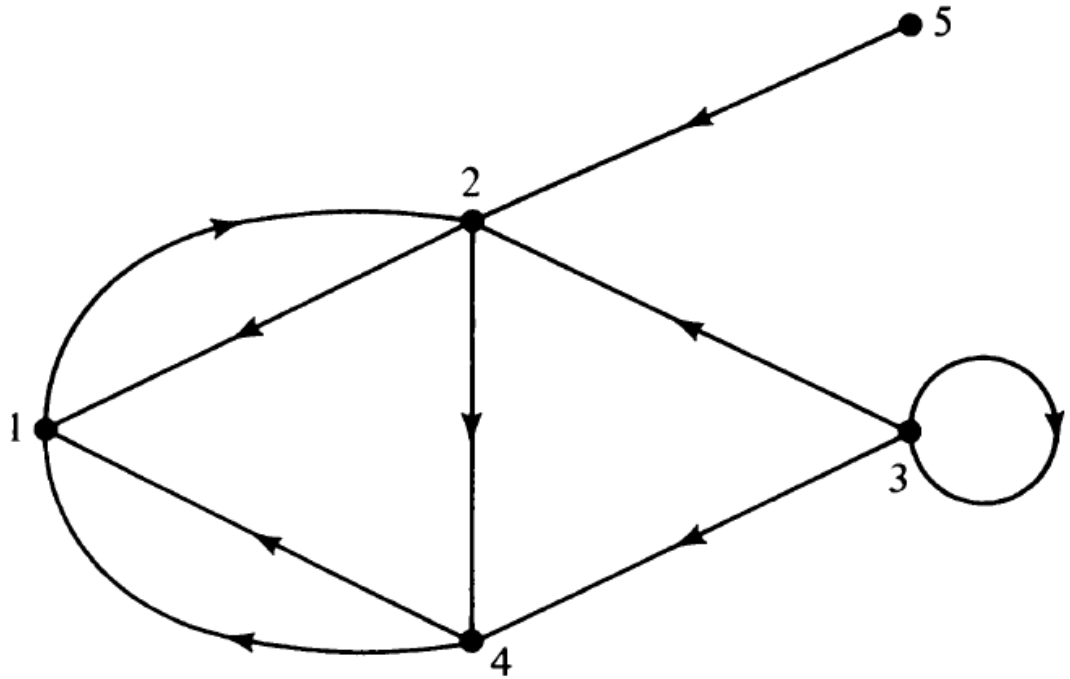
- Edge listing is more economical than adjacency matrix if :

$$2e*b < n^2$$

- Edge listing is very convenient form for inputting a graph into the computer, but storage and retrieval & manipulation of graph within the computer become quite difficult.
- Ex : Searching operation whether a graph is connected or not

Two Linear Arrays

- Variation of edge listing
- Represent graph in two linear arrays
- F=(f1,f2……f_e) & H=(h1,h2,…..h_e)
- Each entries in these array is a vertex label.
- The i^{th} edge e_i is from vertex f_i to vertex h_i if G is a digraph.
- If G undirected, then consider e_i is in between vertex f_i & h_i
- Storage requirements is same as edge listing

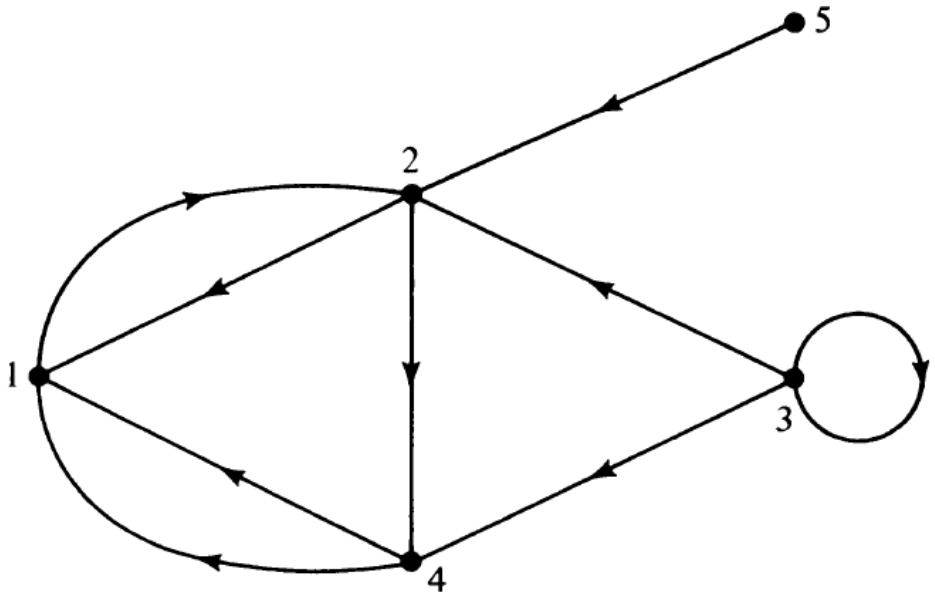


$$F = (5, 2, 1, 3, 2, 4, 4, 3, 3),$$

$$H = (2, 1, 2, 2, 4, 1, 1, 4, 3).$$

Successor Listing

- Another efficient way for representing graphs
- Each vertices are assigned in any order, 1….n, represent each vertex k by a linear array, where first element is k and remaining elements are immediate successors of k(ie, vertices that have directed path of length one from k).



1 : 2

2 : 1, 4

3 : 2, 3, 4

4 : 1, 1

5 : 2

Output

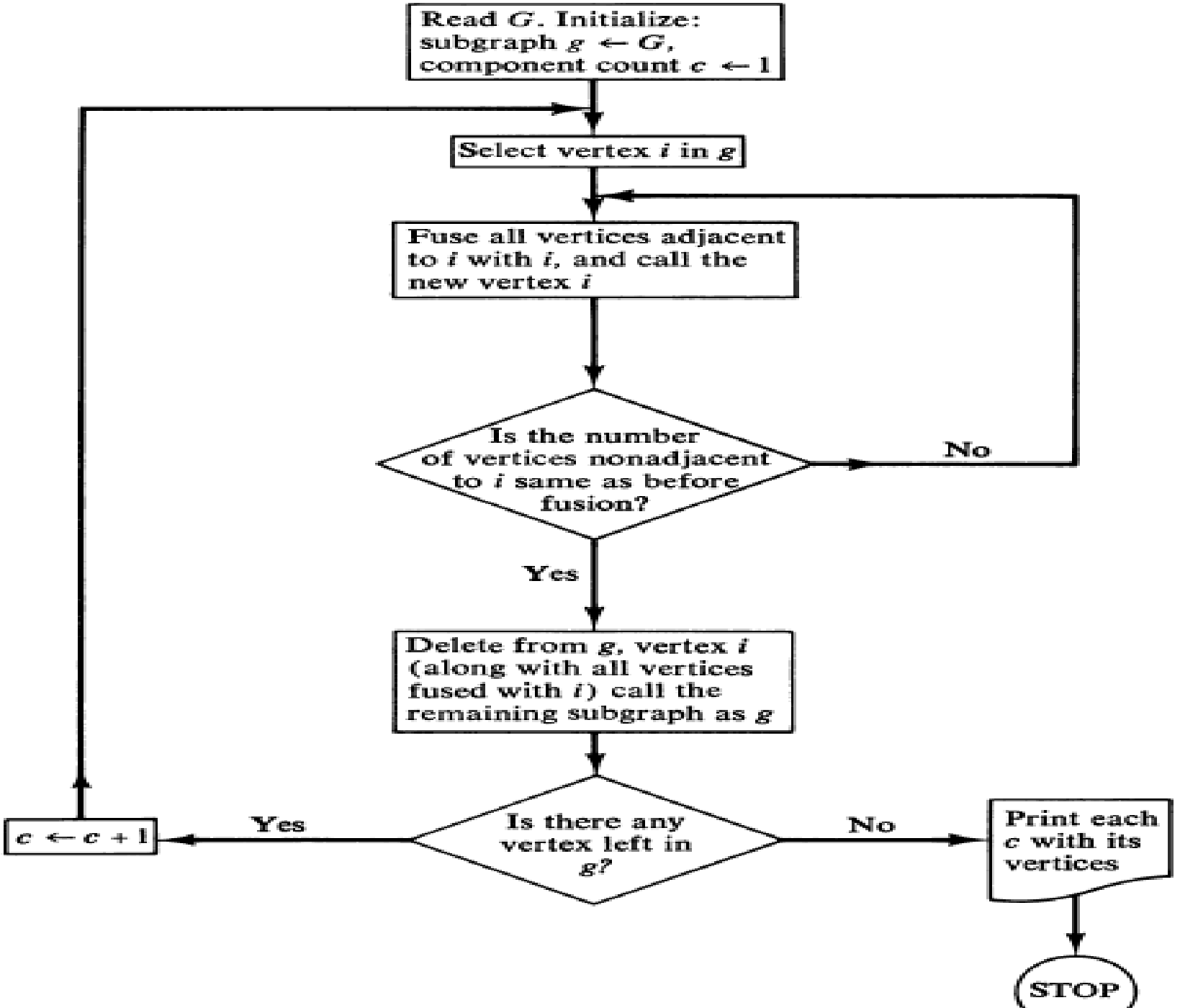
- Input – one or more graphs
- Output depends up on the problem
- If the output consists of subgraphs, the corresponding adjacency matrix is used to represent matrix.
- Ex. If to find a particular graph is connected or not, ie, its an yes or no question, then ask program to just print YES or NO.

Basic algorithms

Algorithm 1: Connected ness and components

- G connected or not. If not, how many components.
- Basic step is the fusion of adjacent vertices.
- Start with some vertices in graph and fuse all vertices that are adjacent to it.
- Then take fused vertex and again fuse it with adjacent vertices.
- This process of fusion should be repeated until no more vertices can be fused.

- This indicates that a connected component has been fused to a single vertex.
- Otherwise, start with new component and continue fusing operation.
- In adjacency matrix, fusion of jth vertex to ith vertex is accomplished by logical OR- operation
- Maximum number of fusions that may be performed in this algorithm is n-1, n is the number of vertices.
- And in each fusions one performs at most n logical additions , upper bound on execution time is proportional to n(n-1)


Read G. Initialize: subgraph g ← G, component count c ← 1
Select vertex i in g
Fuse all vertices adjacent to i with i, and call the new vertex i
Is the number of vertices nonadjacent to i same as before fusion?
No
Yes
Delete from g, vertex i (along with all vertices fused with i) call the remaining subgraph as g
Is there any vertex left in g?
Yes
c ← c + 1
No
Print each c with its vertices
STOP

- Efficiency of algorithm can be increased by proper selection of initial vertex that is to be fused with adjacent vertices.
- It should not cost much for selecting initial vertex.

- Example:
- Input : 20x20 matrix
- Output :Number of components and the list of vertices in each component

N = 20

$$X = \begin{bmatrix}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}$$

OUTPUT

```
COMP 1; VERT:  6  7   8   10  16  19  20
COMP 2; VERT:  1  2   3   5   11
COMP 3; VERT:  9  12  13  17
COMP 4; VERT:  14 15
COMP 5; VERT:  4
COMP 6; VERT:  18
```

Algorithm 2 : Spanning Tree

- Minimum a single spanning tree can be yield from this algorithm, if G is connected.
- If G is disconnected, a spanning forest containing n-p edges can be formed, ***where p > 1*** is the number of components in the disconnected graph.
- Algorithm can be used to find the shortest spanning tree.
- This algorithm can also be used to ***find connected ness of a graph.***

Description: Consider G with n vertices and e edges, is connected and have no self loops

- G is represented using two linear arrays F and H
- At each stage of algorithm, new edge is tested to see if either or both of its end vertices appear in any tree formed so far.

- At kth stage, 1 < =k < = e, in examining the edge (f_k, h_k) five different conditions may occur:

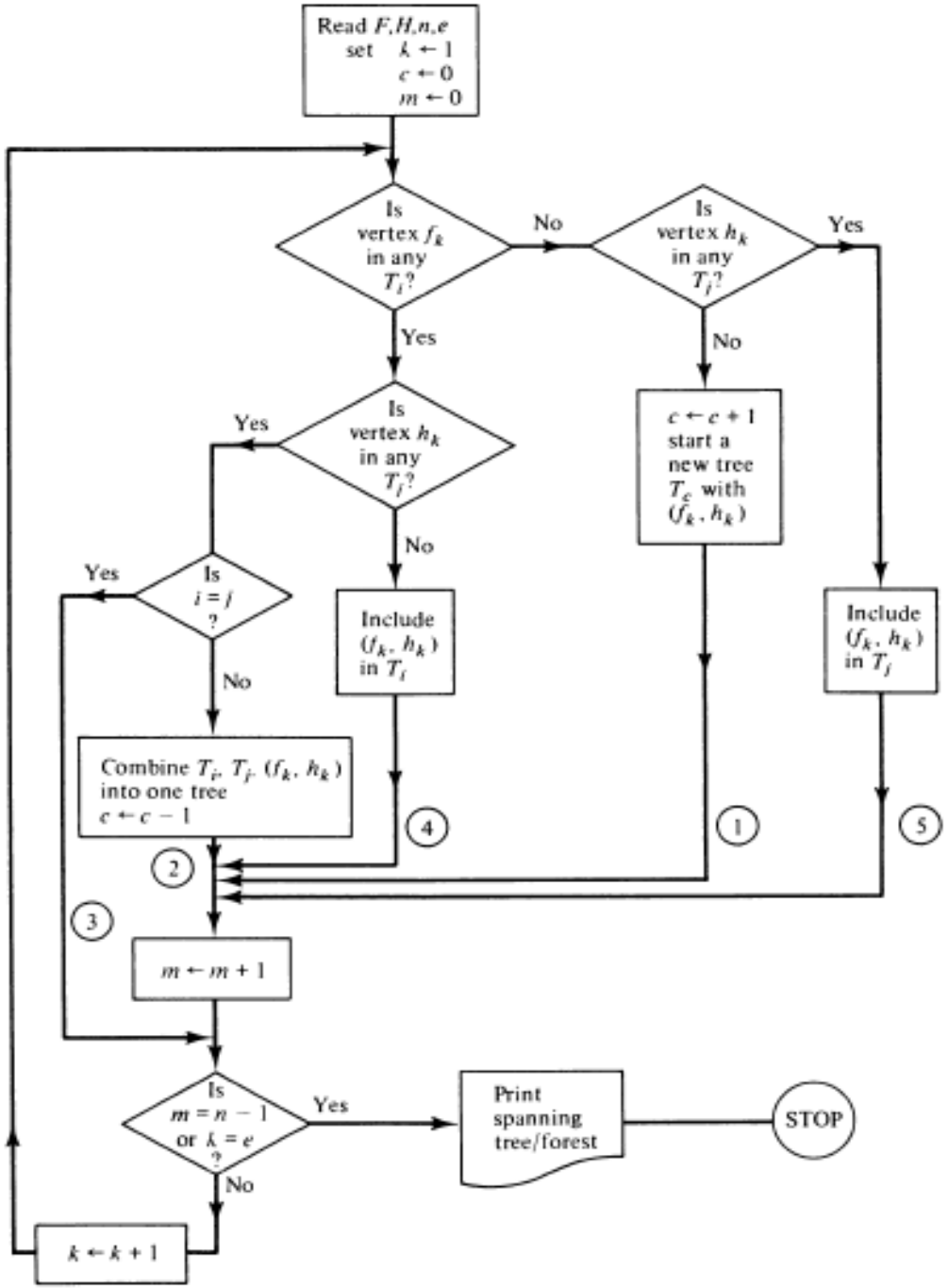
1. If neither vertex f_k nor h_k is included in any of the trees constructed so far in G, the kth edge is named as a new tree and its end vertices f_k, h_k are given the component number c, after incrementing the value of c by 1.
2. If vertex f_k is in some tree T_i ($i = 1, 2, \ldots, c$) and h_k in tree T_j ($j = 1, 2, \ldots, c$, and $i \neq j$), the kth edge is used to join these two trees; therefore, every vertex in T_j is now given the component number of T_i. The value of c is decremented by 1.
3. If both vertices are in the same tree, the edge (f_k, h_k) forms a fundamental circuit and is not considered any further.

4. If vertex f_k is in a tree T_i and h_k is in no tree, the edge (f_k, h_k) is added to T_i by assigning the component number of T_i to h_k also.

5. If vertex f_k is in no tree and h_k is in a tree T_j, the edge (f_k, h_k) is added to T_j by assigning the component number of T_j to f_k also.

- Efficiency of algorithm depends on the speed with which we can test whether or not the end vertices of the edge under consideration have occurred in any tree formed so far.
- Maintain an array **VERTEX** of size n to test it.
- When an edge (i, j) is included in c^{th} tree, the i^{th} and j^{th} entries of array are set to c.
- A zero in the q^{th} position of the array indicates vertex q is not so far added in any tree.
- At end of execution, this array **VERTEX** identifies the components of G

- Output will be a linear array EDGE which contains the edges of spanning tree.
- If kth edge is in cth tree, then EDGE(k)=c, otherwise its zero
- All zero entries correspond to chords
- Main loop of this algorithm works for e times.
- The time required to test whether or not the end vertices have appeared in any tree is constant – independent to e & n
- Execution time is proportional to e.
- If e/n ration is high, to reduce the execution time we maintain a variable to count the number of edges in tree.
- If the value of the variable reaches n-1, we can terminate execution of algorithm.


Read F,H,n,e
set k ← 1
c ← 0
m ← 0
Is vertex f_k in any T_i?
No
Is vertex h_k in any T_i?
Yes
Yes
No
Is vertex h_k in any T_i?
$c \leftarrow c + 1$ start a new tree T_c with (f_k, h_k)
Yes
No
Yes
Is $i = j$?
Include (f_k, h_k) in T_i
Include (f_k, h_k) in T_j
No
Combine T_i, T_j, (f_k, h_k) into one tree
$c \leftarrow c - 1$
4
1
5
2
3
$m \leftarrow m + 1$
Is $m = n - 1$ or $k = e$?
Yes
Print spanning tree/forest
STOP
No
$k \leftarrow k + 1$

- ***Minimal Spanning Tree Algorithms:***

Kruskal's algorithm is used to find shortest spanning tree in G.

This is done using the above described algorithm (algorithm to find spanning tree) in addition with inequality :

Wt of edge $(f_i, h_i) <=$ ***wt of edge*** (f_{i+1}, h_{i+1})

Due to sorting in Kruskal's algorithm , Prim's also is used.

Spanning trees with desired properties:

- Same algorithm that is used to find spanning tree of G can be used.
- An additional sorting is only needed.

- ***Generating all spanning trees:***

Effective algorithm to find all spanning trees is reducing the graph G by deleting edges and fusion of end vertices.

From reduced graph spanning trees, we can obtain spanning trees of original graph.

Shortest Path Algorithms

- Different shortest path problems:
  - . Shortest path between two specified vertices
  - Shortest path between pair of vertices
  - Shortest path from a specified vertex to all others
  - Shortest path between two specified vertices that passes to a specified vertex
  - Second , third etc shortest paths

1. *Shortest path between one specified vertex to another specified vertex*

A simple weighted digraph† G of n vertices is described by an n by n matrix $\mathbf{D} = [d_{ij}]$, where

d_{ij} = length (or distance or weight) of the directed edge from vertex i to vertex j, $d_{ij} \geqslant 0$,

$d_{ii} = 0$,

$d_{ij} = \infty$, if there is no edge from i to j

- Dijkstra’s algorithm

- Label all vertices of given digraph.
- At each stage in algorithm, some vertices have permanent labels and some have temporary labels.
- Algorithm starts by assuming a permanent label 0 to the starting vertex s & temporary labels ∞ to remaining n-1 vertices.
- After each iteration another vertex gets permanent label according to following rules:

1. Every vertex j that is not yet permanently labeled gets a new temporary label whose value is given by

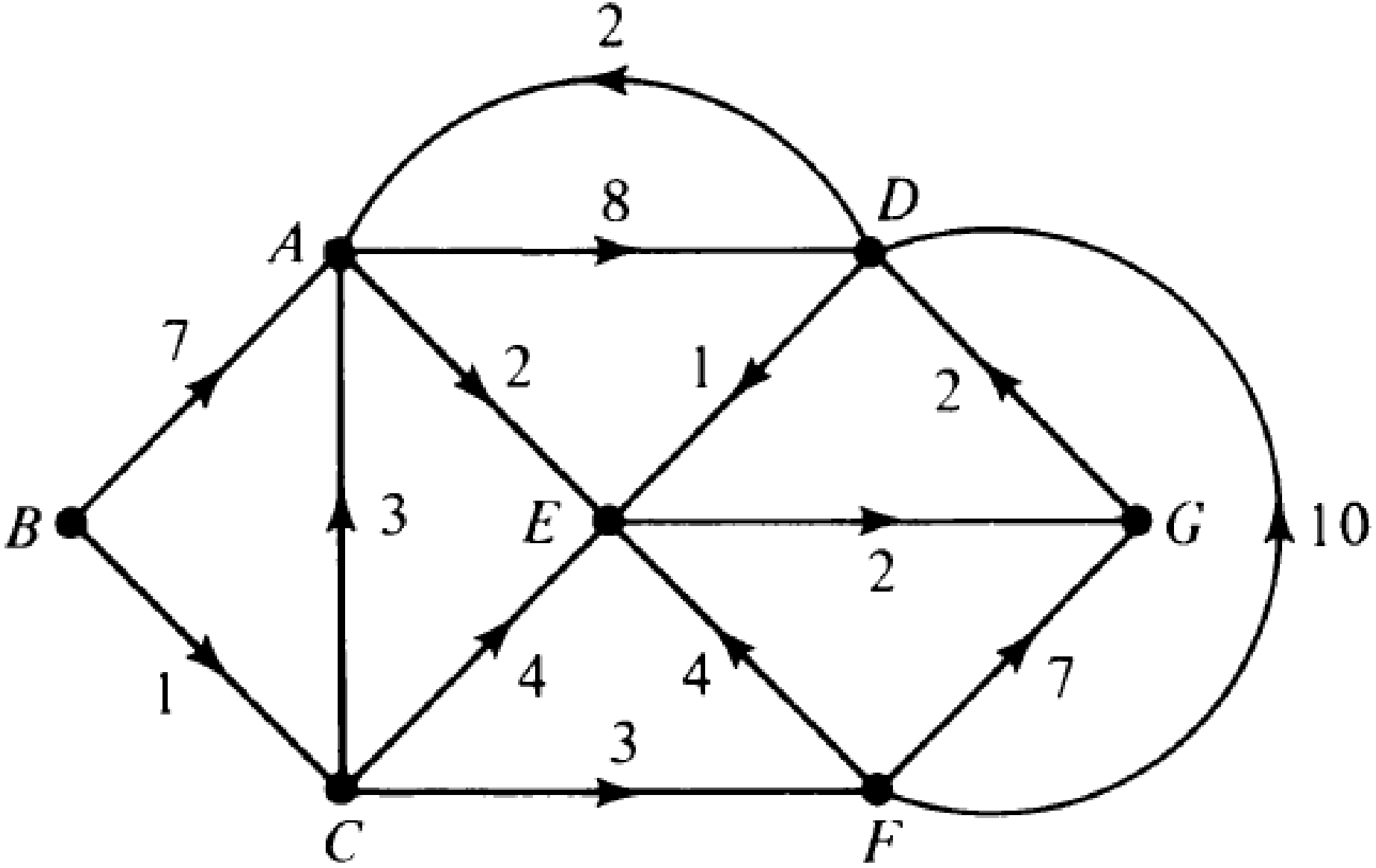
$$\min [\text{old label of } j, (\text{old label of } i + d_{ij})],$$

where i is the latest vertex permanently labeled, in the previous iteration, and d_{ij} is the direct distance between vertices i and j. If i and j are not joined by an edge, then $d_{ij} = \infty$.

2. The smallest value among all the temporary labels is found, and this becomes the permanent label of the corresponding vertex. In case of a tie, select any one of the candidates for permanent labeling.

Steps 1 and 2 are repeated alternately until the destination vertex t gets a permanent label.

The first vertex to be permanently labeled is at a distance of zero from s. The second vertex to get a permanent label (out of the remaining $n - 1$ vertices) is the vertex closest to s. From the remaining $n - 2$ vertices, the next one to be permanently labeled is the second closest vertex to s. And so on. The permanent label of each vertex is the shortest distance of that vertex from s.

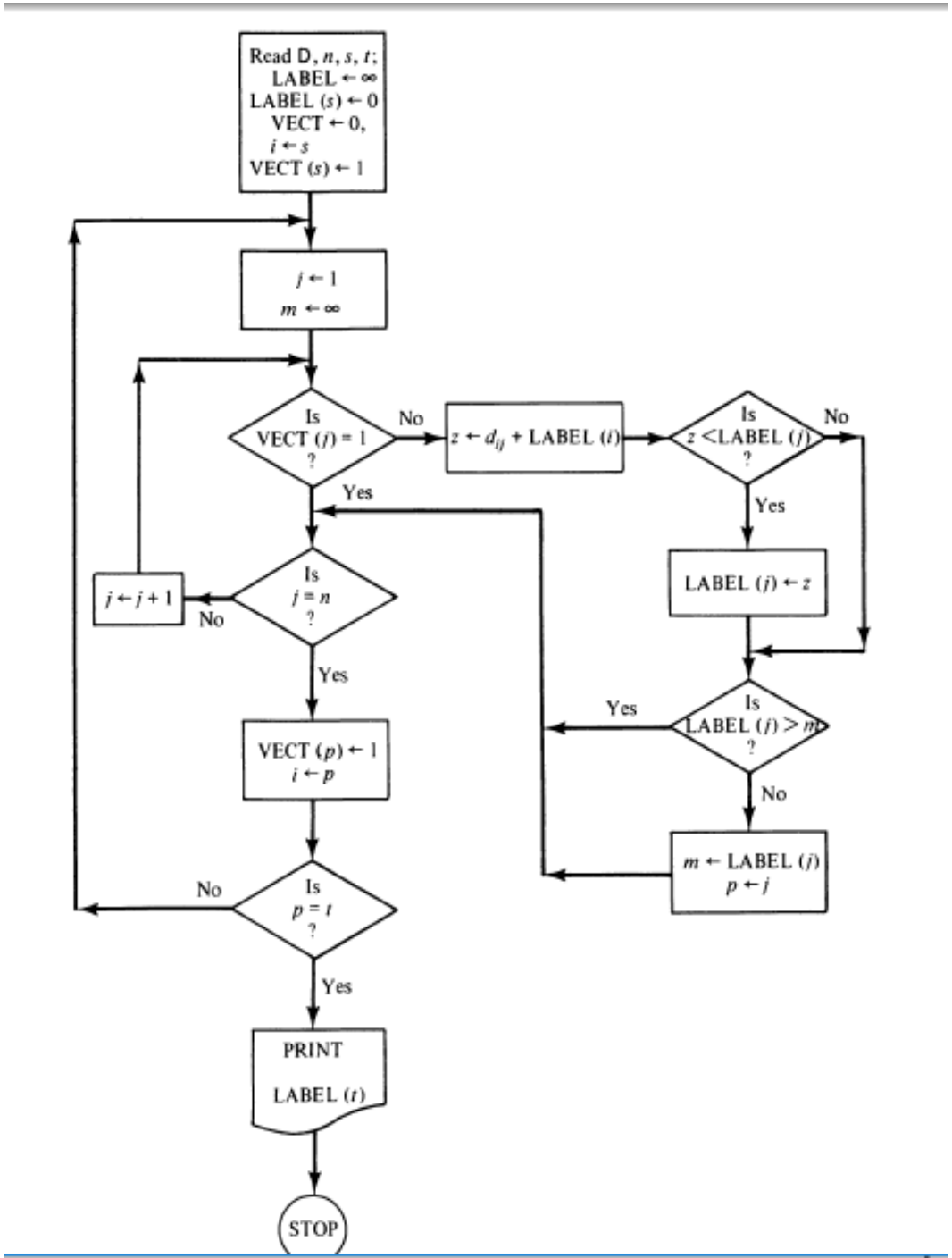

2
8
D
A
7
2
1
2
B
3
E
G
10
2
1
4
4
7
3
C
F

A	B ✓	C	D	E	F	G	
∞	[0]	∞	∞	∞	∞	∞	: Starting Vertex B is labeled 0.
7	[0]	1 ✓	∞	∞	∞	∞	: All successors of B get labeled.
7	[0]	[1]	∞	∞	∞	∞	: Smallest label becomes permanent.
4	[0]	[1]	∞	5	4 ✓	∞	: Successors of C get labeled.
4	[0]	[1]	∞	5	[4]	∞	
4 ✓	[0]	[1]	14	5	[4]	11	
[4]	[0]	[1]	14	5	[4]	11	
[4]	[0]	[1]	12	5 ✓	[4]	11	
[4]	[0]	[1]	12	[5]	[4]	11	
[4]	[0]	[1]	12	[5]	[4]	7 ✓	
[4]	[0]	[1]	12	[5]	[4]	[7]	: Destination vertex gets permanently labeled.

- Here shortest distance is 7.

- We should traverse through B-C-E-G to get total distance travelled as 7.

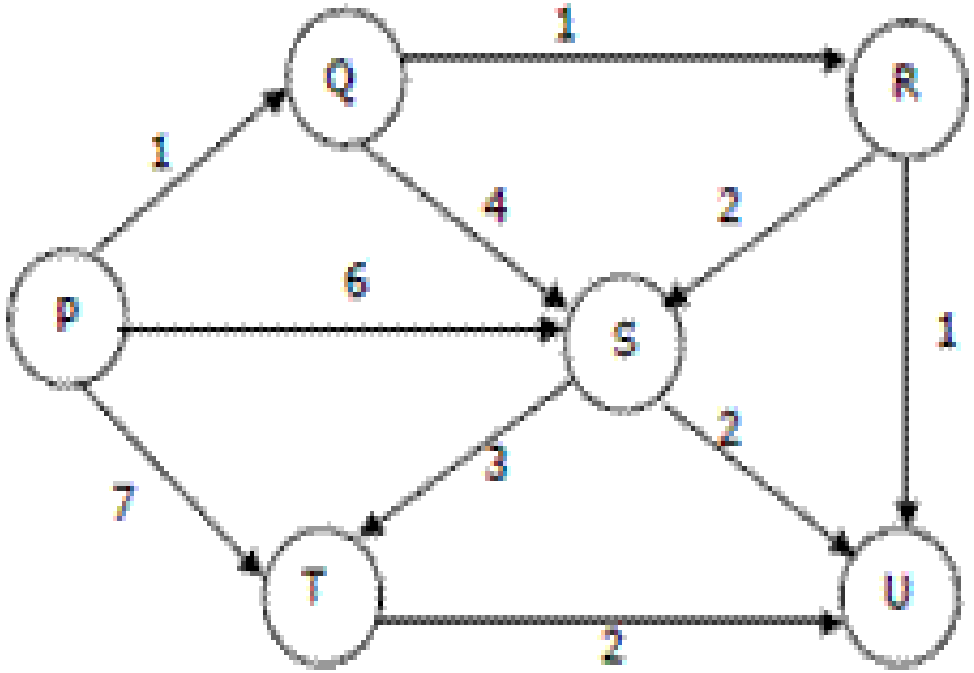
- ***Permanent label is the shortest distance to that particular vertex from source vertex.***
- Main difficulty of this algorithm is to distinguish between permanently and temporarily labelled vertices.
- An efficient method to distinguish permanent and temporarily labelled vertices is to maintain a binary vector VECT of order n.
- When ith vertex becomes permanently labelled, then corresponding ith element in VECT changes from 0 to 1.


Read D, n, s, t;
LABEL ← ∞
LABEL (s) ← 0
VECT ← 0,
i ← s
VECT (s) ← 1
j ← 1
m ← ∞
Is VECT (j) = 1 ?
No
z ← d_ij + LABEL (i)
Is z <LABEL (j) ?
No
Yes
Yes
j ← j + 1
Is j = n ?
No
LABEL (j) ← z
Yes
Yes
Is LABEL (j) > m ?
VECT (p) ← 1
i ← p
No
m ← LABEL (j)
p ← j
No
Is p = t ?
Yes
PRINT LABEL (t)
STOP

The algorithm described does not actually list the shortest path from the starting vertex to the terminal vertex; it only gives the shortest distance. The shortest path can be easily constructed by working backward from the terminal vertex such that we go to that predecessor whose label differs exactly by the length of the connecting edge. (A tie indicates more than one shortest path.) Alternatively, the shortest path can be determined by keeping a record of the vertices from which each vertex was labeled permanently. This record can be maintained by another linear array of length n, such that whenever a new permanent label is assigned to vertex j, the vertex from which j is directly reached is recorded in the jth position of this array.

Find shortest path between P & U……

Using Shortest Path Algorithms



Algorithm : Shortest Path between all pairs of vertices

- Aim is to find shortest paths between n(n-1) ordered pairs of vertices in a digraph.
- In case of undirected graph, find distance between n(n-1)/2 unordered pair of vertices.
- Algorithm used to find shortest path between two specified vertices can be used here. But the computation time required will be proportional to n^4.

Steps:

- Algorithm works by inserting one or more vertices into path, whenever it is good.
- Start with n by n matrix $D=[d_{ij}]$of direct distances
- n different matricies D_1, D_2,….D_n are constructed sequentially
- Matrix D_k,1<=k<=n, whose (i, j) entry gives the length of the shortest directed path among all directed paths from i to j.

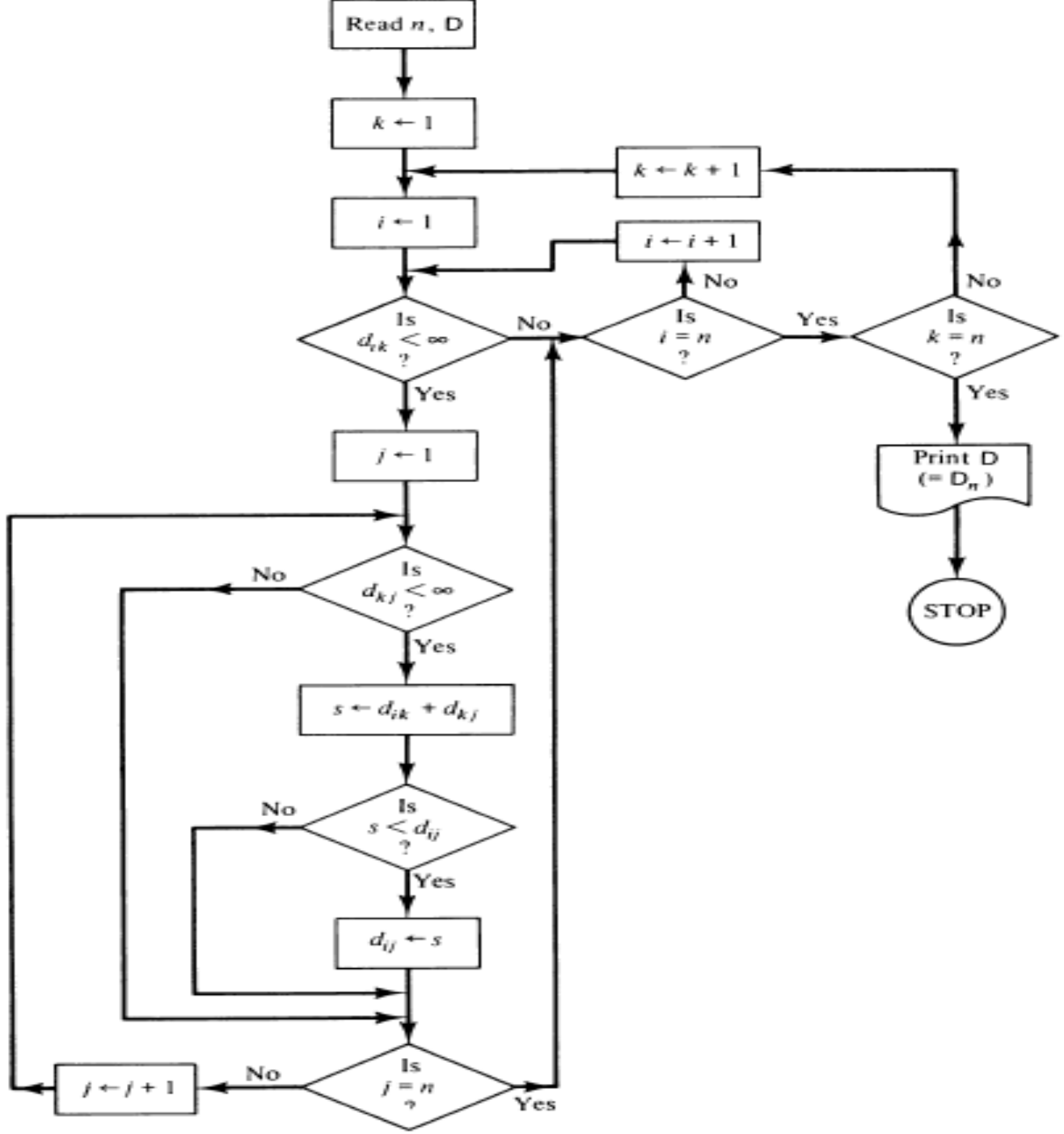
- Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$.
- When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all, hence $d_{ij}^{(0)} = w_{ij}$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \ , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \ . \end{cases}$$

Floyd-Warshall(W)

1 $n \leftarrow rows[W]$
2 $D^{(0)} \leftarrow W$
3 **for** $k \leftarrow 1$ **to** n
4 **do for** $i \leftarrow 1$ **to** n
5 **do for** $j \leftarrow 1$ **to** n
6 **do** $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7 **return** $D^{(n)}$

Running time $O(V^3)$


Read n, D
k ← 1
i ← 1
j ← 1
Is $d_{ik} < \infty$?
No
Yes
Is $d_{kj} < \infty$?
No
Yes
$s \leftarrow d_{ik} + d_{kj}$
Is $s < d_{ij}$?
No
Yes
$d_{ij} \leftarrow s$
Is $j = n$?
No
Yes
$j \leftarrow j + 1$
Is $i = n$?
No
Yes
$i \leftarrow i + 1$
Is $k = n$?
No
Yes
$k \leftarrow k + 1$
Print D (= D_n)
STOP