DIRECTED GRAPHS

UNIT – III

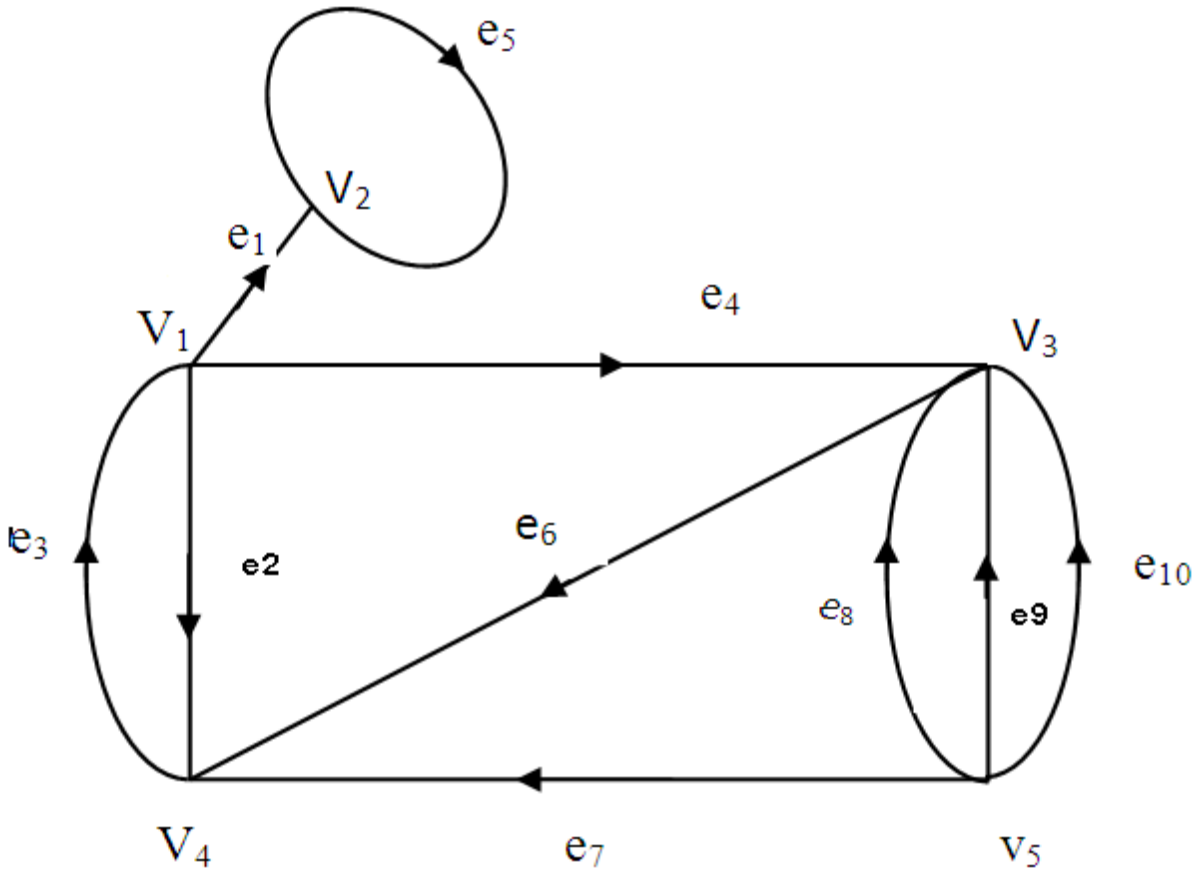
Prof. K.Maheswaran

- Directed graphs
  - ✓ In-degree
  - ✓ Out – degree
  - ✓ Isolated , pendant digraphs
  - ✓ Isomorphic
- Types of di-graphs
- Tree with directed edges

What is a directed graph?

A directed graph (*or a digraph*) G consist of a set of vertices $V=\{v_1, v_2, \ldots\ldots\}$, a set of edges $E=\{e_1, e_2, \ldots\ldots\}$,and a mapping ψ that maps every edge onto some ordered pair of vertices (v_i, v_j).

As in the case of undirected graphs, a vertex is represented by a point and an edge by a line segment between v_i to v_j. For example, the following shows the digraph which is an oriented graph.



Directed graph with five vertices

In a digraph an edge is not only incident on a vertex, but is also **incident out** of a vertex and **incident into** a vertex.

The vertex v_i, which edge is incident out of, is called the **initial vertex** of e_k. The vertex v_j, which e_k is incident into, is called the **terminal vertex** of e_k .

In the above example v_5 is the initial vertex and v4 is the terminal vertex of e7 . An edge for which the initial and terminal vertices are the same forms a self-loop, such as e5.

The number of edges incident out of a vertex vi, is called the out-degree (or out-valence or outward demidegree) of vi, and is written as d^+ (vi). The number of edges incident into vi is called the in-degree (or a in-valence or inward demidegree) of vi and is written as d^- (vi).

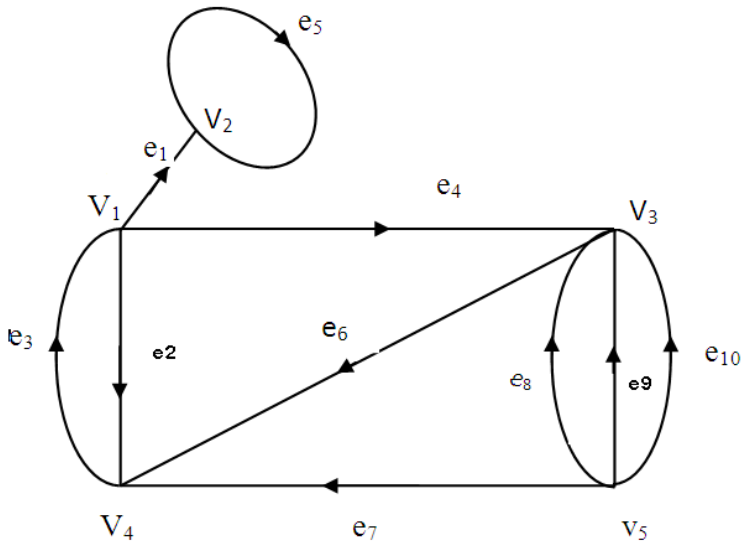
$d^+(v1)=3$, $d^-(v1)=1$,

$d^+(v2)=1$, $d^-(v2)=2$,

$d^+(v5)=4$, $d^-(v5)=0$,

$d^+(v4)=1$, $d^-(v_4)=3$

$d^+(v_3)=1$, $d^-(v3)=4$



In any Digraph G the all in-degrees is equal to the sum of all out-degrees.

$$\sum_{i=1}^{n} d^{+}(v_i) = \sum_{i=1}^{n} d^{-}(v_i)$$

An isolated vertex is a vertex in which the in-degree and the out-degree are both equal to zero. A vertex v in a digraph is called a pendent if it is of degree one, that is if

$$d^{+}(v) + d^{-}(v) = 1$$

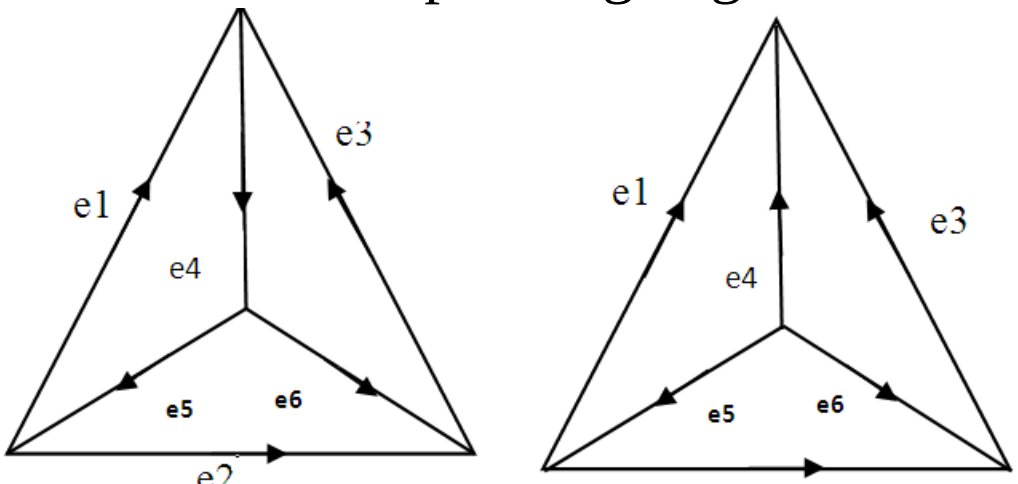
Two directed edges are said to be parallel if they are mapped onto the same ordered pair of vertices. That is, in addition to being parallel in the sense of undirected edges, parallel directed must also agree in the direction of their arrows.

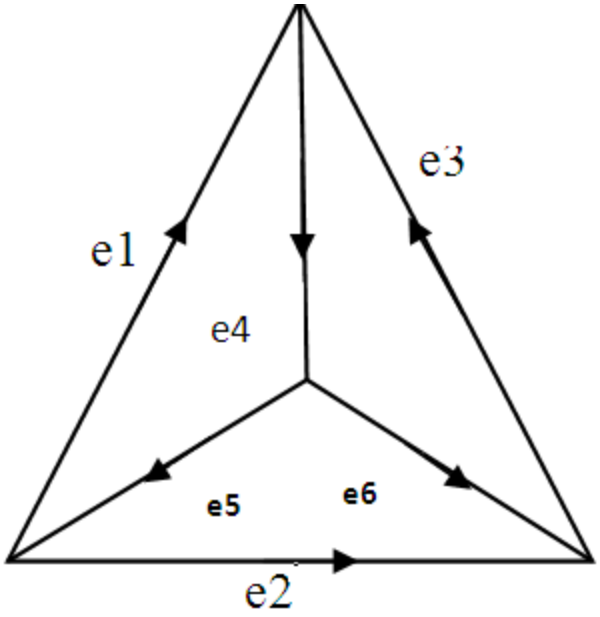
In the above example, edges e8, e9 and e10 are parallel whereas edges e2 and e3 are not.

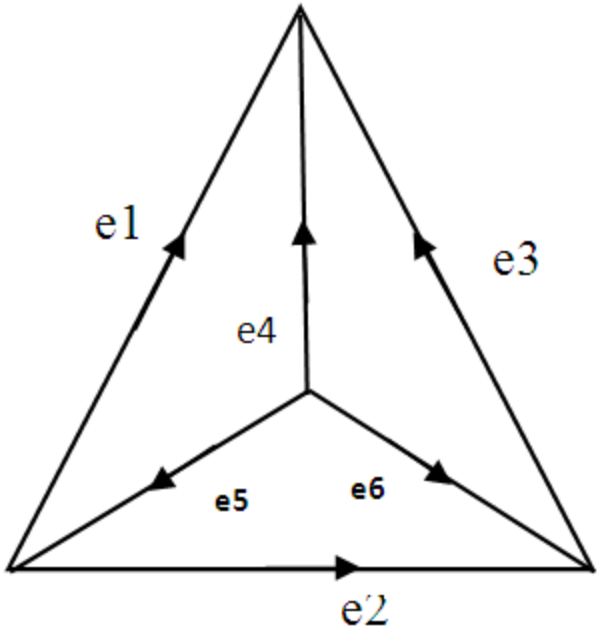
ISOMORPHIC DIGRAPHS:

Isomorphic digraphs were defined such that they have identical behavior in terms of graph properties. In other words, if their labels are removed, two isomorphic graphs are indistinguishable.

For two digraphs to be isomorphic not only must their corresponding undirected graphs be isomorphic, but the directions of the corresponding edges must also agree.







TYPES OF DIGRAPHS:

SIMPLE DIGRAPHS:

A digraph that has no self-loop or parallel edges is called a simple digraph.

ASYMMETRIC DIGRAPHS:

Digraphs that have at most one directed edge between a pair of vertices, but are allowed to have self-loops are called asymmetric or anti-symmetric.

SYMMETRIC DIGRAPHS:

Digraphs in which for every edge (a, b) there is also an edge (b, a). A digraph that is both simple and symmetric is called a simple symmetric digraph. Similarly, a digraph that is both simple and asymmetric is simple asymmetric. The reason for the terms symmetric will be apparent in the context of binary relations.

COMPLETE DIGRAPH:

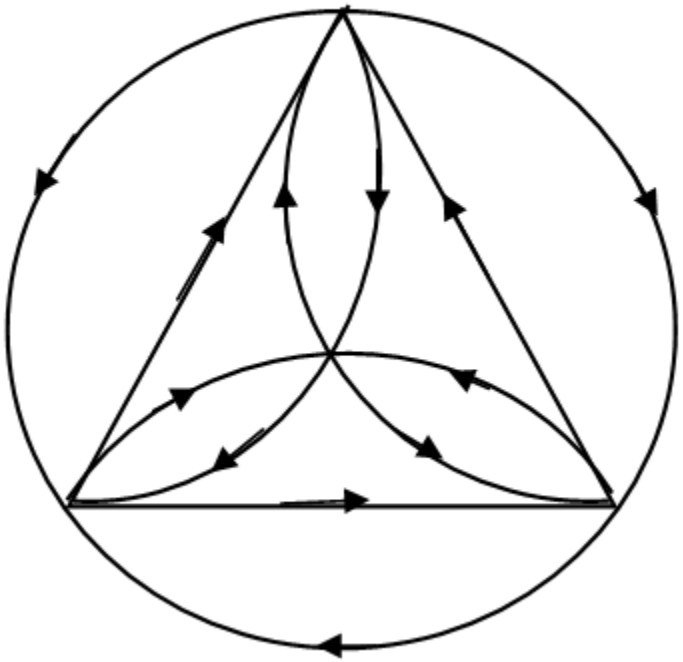
A complete undirected graph was defined as a simple graph in which every vertex is joined to every other vertex exactly by one edge. For digraphs we have two types of complete graphs

A complete symmetric digraph is a simple digraph in which there is exactly one edge directed from every other vertex, and a complete asymmetric digraph is an asymmetric digraph in which there is exactly one edge between every pair of vertices.

A complete asymmetric digraph of n vertices contains n(n-1)/2 edges, but a complete symmetric digraph of n vertices contains n(n-1) edges. A complete asymmetric digraph is also called as a tournament or a complete tournament.

A digraph is said to be balanced if for every vertex Vi the in-degree equals the out-degree; that is d^+ (v)+ d^- (v)=1. (A balanced digraph is also referred to as a pseudo-symmetric digraph or an isograph). A balanced digraph is said to be regular if every vertex has the same in-degree and out-degree as every other vertex.

A complete symmetric digraph of four vertices:



TREES WITH DIRECTED EDGES:

A tree is a connected digraph that has no circuit, neither a directed circuit nor a semi circuit. A tree of n vertices contains n-1 directed edges and has properties similar to those with undirected edges.

Trees with directed edges are of great importance in many applications such as electrical network analysis, game theory, theory of languages, computer programming, and counting problems, to name a few.

ARBORESENCE:

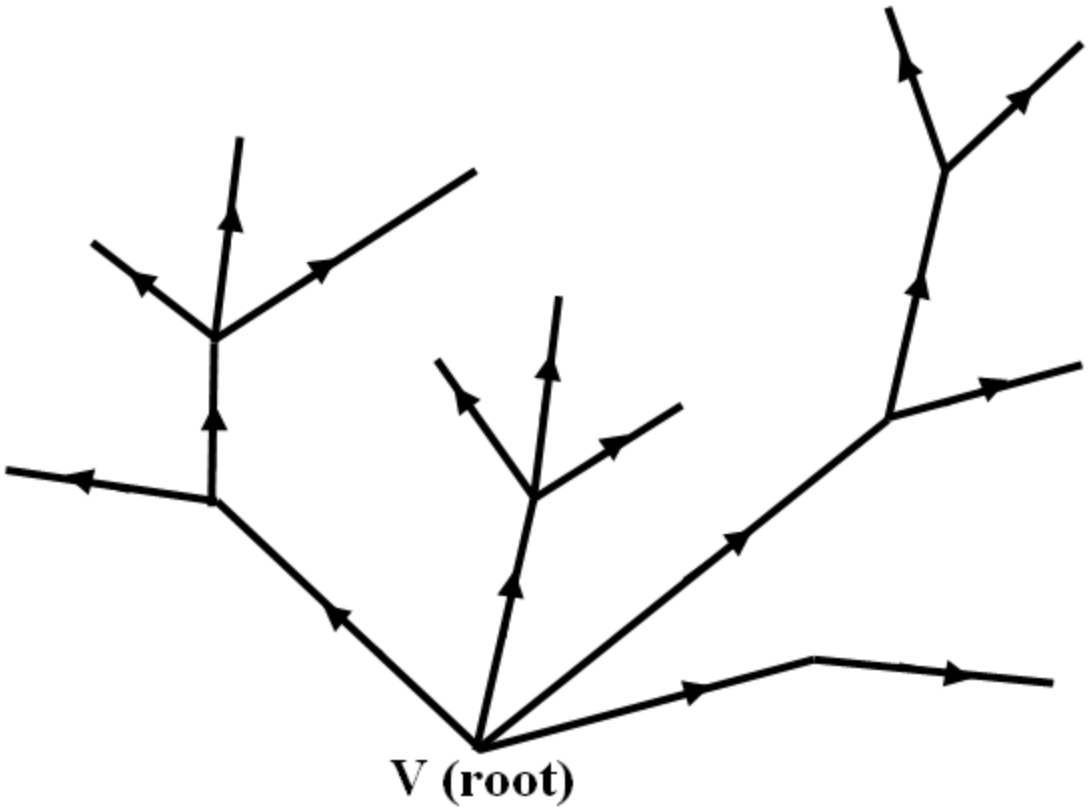
A digraph G is said to be an arborescence if

i) G contains no circuit neither directed nor semi circuit.

ii) In G there is precisely one vertex v of zero in-degree.

This vertex v is called the root of arborescence. It is shown in the following:

Arborescence.



THEOREM:

An arborescence is a tree in which every vertex other than the root has an in-degree of exactly one.

Euler digraph

- In a digraph G a closed directed walk (i.e., a directed walk that starts and ends at the same vertex) which traverses every edge of G exactly once is called a directed Euler line.
- A digraph containing a directed Euler line is called an Euler digraph. The graph in Fig. is an Euler digraph, in which the walk a b c d e f is an Euler line.

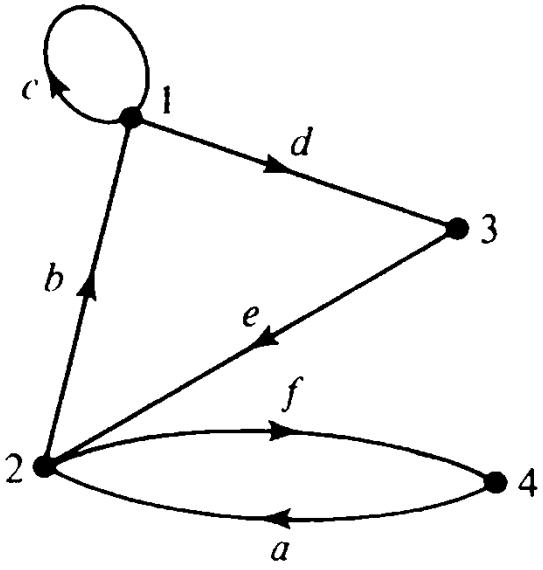


Fig. 9 Euler digraph.

COMPUTER REPRESENTATION OF A GRAPH

Prof. K.Maheswaran

ADJACENCY MATRIX

The most popular form in which a graph or digraph is fed to a computer is its adjacency matrix. Assign a distinct number to each of the n vertices of the given graph G, the n by n binary matrix X(G) is used for representing G during input, storage, and output. Since each of the n^2 entries is either a 0 or a 1, the adjacency matrix requires n^2 bits of computer memory.

Bits can be packed into words. Let w be the word length and n be the number of vertices in the graph. Then each row of the adjacency matrix may be written as a sequence of n bits in [n/w]machine words. ([x] denotes the smallest integer not less than x). The number of words required to store the adjacency matrix is, therefore *n [n/w]*

.

- The adjacency matrix of an undirected graph is symmetric, and therefore storing only the upper triangle is sufficient. This requires only n(n-1)/2 bits of storage. This saving in storage, however, often costs in increased complexity and computation time

INCIDENCE MATRI X

An incidence matrix is also used for storing and manipulation of a graph. An incidence matrix requires n.e bits of storage, which might be more than the n^2 bits needed for an adjacency matrix, because the number of edges e is usually greater than the number of vertices n.

On rare occasions it may be advantageous to use the incidence matrix rather than the adjacency matrix, in spite of the increased requirements in storage. Incidence matrices are particularly favored for electrical networks and switching networks.

EDGE LISTING:

Another representation often used is to list all edges of the graph as vertex pairs, having numbered the n vertices in some arbitrary order. For example, the digraph in the following would appear as a set of the following ordered pair: 1,2),(2,1),(2,4),(3,2),(3,3),(3,4),(4,1),(4,1),(5,2). Had this graph been undirected, we would simply ignore the ordering in each vertex pair.

Clearly, parallel edges and self loops can be included in this representation of a graph or digraph.

The number of bits required to label(1 through n) vertex is b, where

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

EDGE LISTING

The number of bits required to label (1 through n) each vertex is b, where

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

And since each of the e edges requires storing two such numbers, the total storage required is

$$2e \cdot b \text{ bits.}$$

Comparing this with n^2, we see that this representation is more economical than the adjacency matrix if

$$2e \cdot b < n^2.$$

In other words, for a graph whose adjacency matrix is sparse†, edge listing is a more efficient method of storing the graph.

Edge listing is a very convenient form for inputting a graph into the computer, but the storage, retrieval, and manipulation of the graph within the computer become quite difficult.

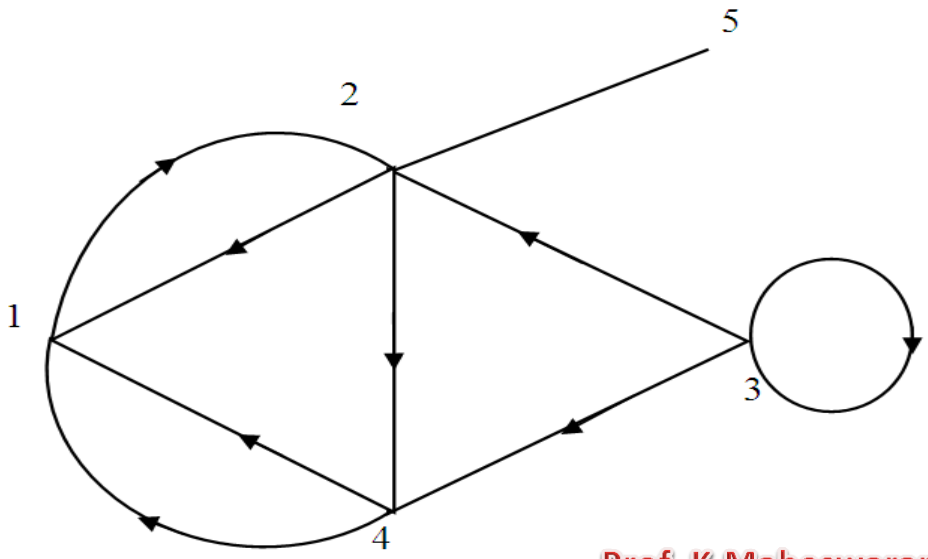
TWO LINEAR ARRAYS:

A slight variation of edge listing is to represent the graph by two linear arrays, say F=(f_1,f_2,……,f_e) and H=(h_1,h_2,……,h_e). Each entry in these arrays is a vertex label. The i^{th} edge ei is from vertex fi to vertex hi if G is a digraph.(If G is undirected, just consider e_i as between fi, and hi.) For example, the digraph in the following would represented by the two arrays

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

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

The storage requirements are the same as in Edge Listing.



SUCCESSOR LISTING:

Another efficient method used frequently for graphs in which the ratio e/n is not large is by means of linear arrays. After assigning the vertices, in any order, the numbers 1,2,……..,n, we represent each vertex k by a linear array, whose first element is k and whose remaining elements are the vertices that are immediate successors of k, that is, the vertices which have a directed path of length one from k.(In an undirected graph these are simply vertices adjacent to k.) The five-vertex is given in the above the representations are as follows:

1 : 2
2 : 1,4
3 : 2,3, 4
4 : 1,1
5 : 2

For an undirected graph the neighbors (rather than the successors) of every vertex are listed. Therefore, each edge appears twice an obvious redundancy.

To compare its storage efficiency with that of the adjacency matrix, let d_{av} be the average degree of the vertices in the graph. Assuming that one computer word is needed for the label of each vertex, the total storage requirement for an n vertex graph is $n(1 + d_{av})$ words. Thus the successor listing is more efficient than the adjacency matrix if

$$d_{av} < \lceil n/w \rceil - 1$$

Where w being the word length.

The successor or neighbor listing form is extremely convenient for path-finding algorithms.

GRAPH THEORETIC ALGORITHM

Prof. K.Maheswaran

- ALGORITHM FOR
  - CONNECTEDNESS AND COMPNENT
  - SPANNING TREE
  - SHORTEST PATH

- **SHORTEST SPANNING TREE**
  - KRUSKAL ALGORITHM
  - PRIMS ALGORITHM
- **SHORTEST PATH**

i) Shortest path from a specified vertex to Another specified vertex:

- DIJKSTRA'S ALGORITHM

ii) Shortest path between every vertex pair:

- WARSHALL FLOYDS ALGORITHM

Kruskal Algorithm:(for finding shortest spanning tree)

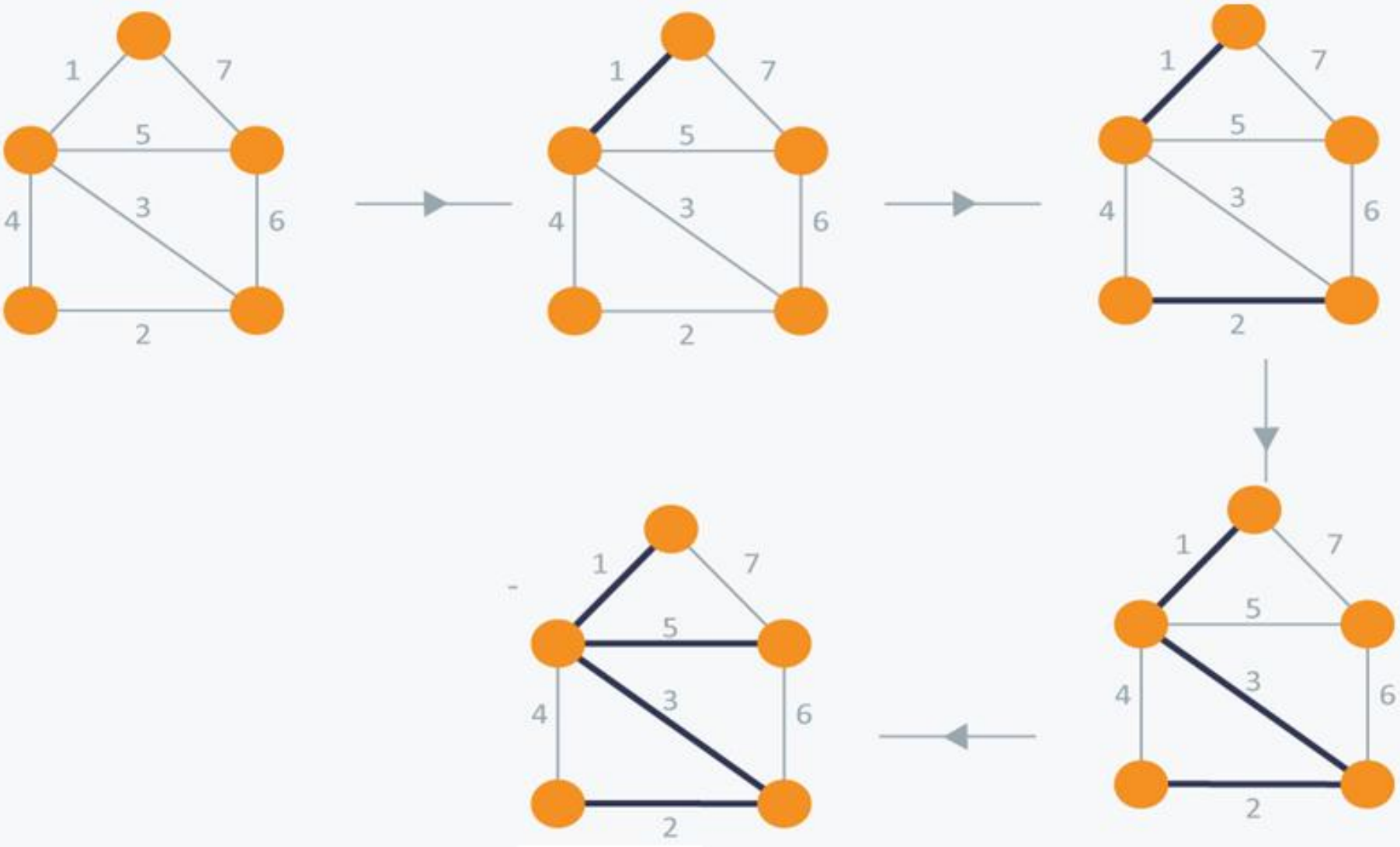
1. List all edges of the graph G in order of increasing weight.
2. Select a smallest edge of G.
3. Select another smallest edge that makes no circuit with the previously selected edges.
4. Continue step (3) until (n-1) edges have been selected .

These edges will constitute the desired shortest spanning tree.

E.g.

7,7,8,9,9,10,10,11,12,16,17,20.

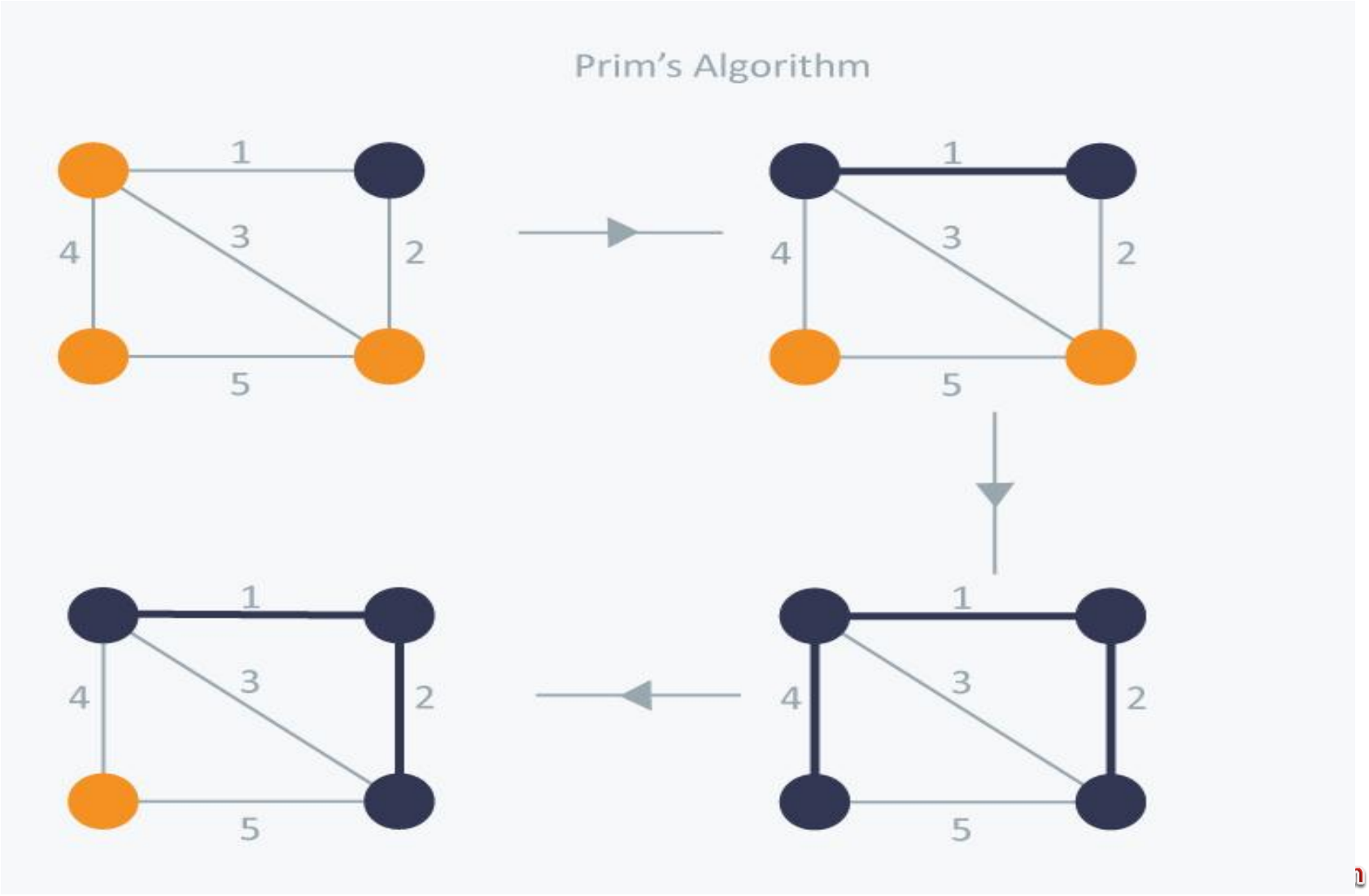
Kruskal's Algorithm



Prim's Algorithm:

1) Draw n isolated vertices and label them $v_1, v_2, \ldots v_n$.
2) Tabulate gm weights of the edges of G in an n by n table. (Note that the entries in the table are symmetric with respect to the diagonal and the diagonal is empty).
3) Set the weights of non existent edges as large.
4) Start from vertex v_1 and connect to its nearest neighbour (i.e. to the vertex which has the smallest entry in row1 of the table), says v_k.
5) Now consider v_1 and v_k as one sub graph and connect this sub graph to its close to neighbour(i.e. to a vertex other than v_1 and v_k that has the smallest entry among all entries in rows 1 and k). Let this new vertex be v_i.

6) Next regard the tree with vertices v1, vk vi as one sub graph, and continue the process until all n vertices have been connected by n-1 edges.



Shortest path from a specified vertex to Another specified vertex

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

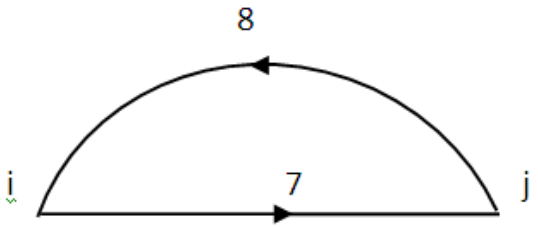
d_{ij} = length (or distance or weight) of the directed edge from vertex i to vertex j,

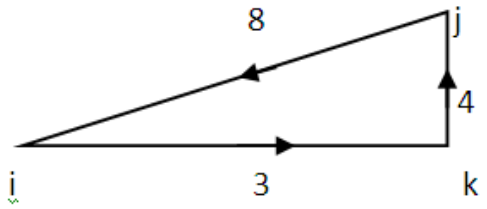
$d_{ij} \geq 0$.

$d_{ij} = 0$

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

ii) In general $d_{ij} \neq d_{ji}$ and triangle inequality need not be satisfied.





$d_{ij} \nleq d_{ik} + d_{kj}$

iii) The distance of a directed path P is defined to be the sum of the length of the edges in P.

Problem is the find the shortest possible path and its length of a starting vertex s to terminal vertex t.

Note:

i) Suppose in a graph we have self loops and parallel edges (it simple graph), it can be made simple by discarding all self loops and replacing every set of parallel edges by the shortest edge among them.

ii) If graph is not directed, then $d_{ij} = d_{ji}$ and each undirected edge is replaced by two appositely directed edge of same weight.

iii) If the graph is not weighted, $d_{ij} = 1$.

Among several algorithms that have been proposed for the shartest path between a specified vertex pair.

Perhaps, the most efficient one is an algorithm the to "Dijkstra".

DIAGRAM

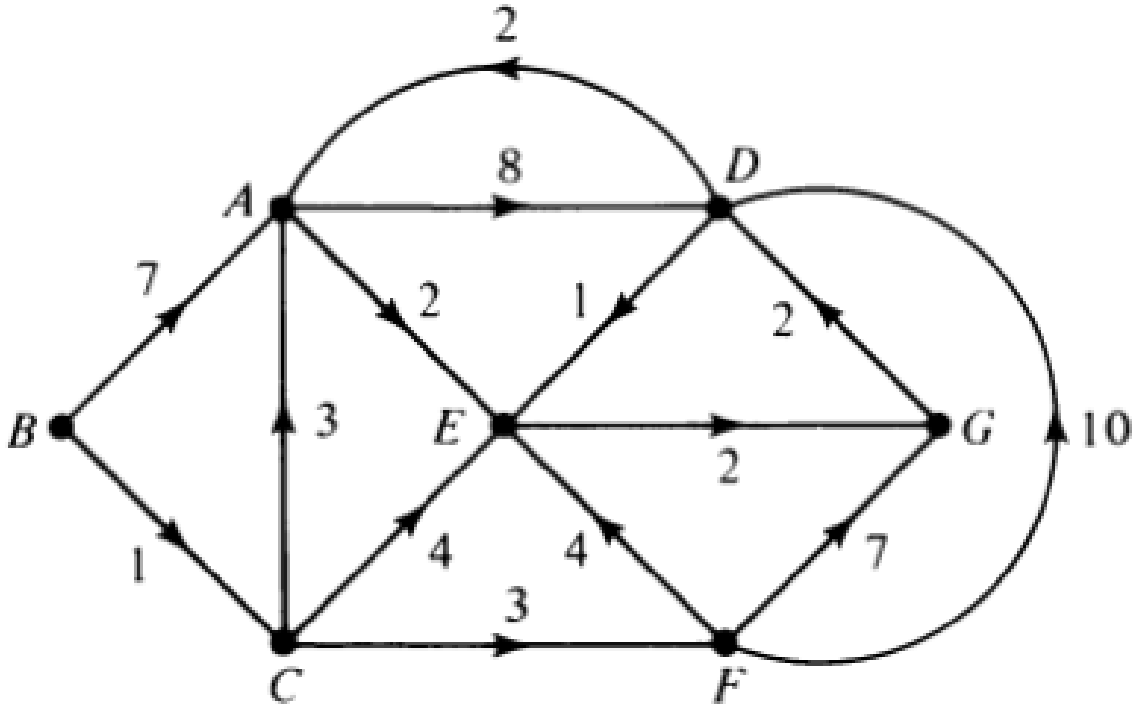


Fig. Simple weighted digraph.

Dijkstra's Algorithm:

i) This algorithm labels the vertices of the given digraph. At each stage in the algorithm some vertices have permanent labels and others temporary labels . The algorithms begin by assigning a permanent label by assigning a permanent label o to the starting vertex s and a temporary label ∞ to the remaining n-1 vertices.

ii) From then on, in each iteration another vertex gets a permanent label, according to the following rules.

a) Every vertex j that is not very permanently labeled gets a new temporary label whose value is given by

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

where i is the latest vertex permanently labeled, in the previously iteration and dij is the direct distance between vertices I and j. If it's i and j are not joined by an edge, then dij = ∞.

b) The smallest value among all the temporary labels is found, and this becomes the permanent label of the corresponding vertex. In the case of 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.

Note: The 1st vertex to be permanently labeled is at a distance of O from s.

The 2^{nd} 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 2^{nd} closest vertex to s. And so on.

Diagram

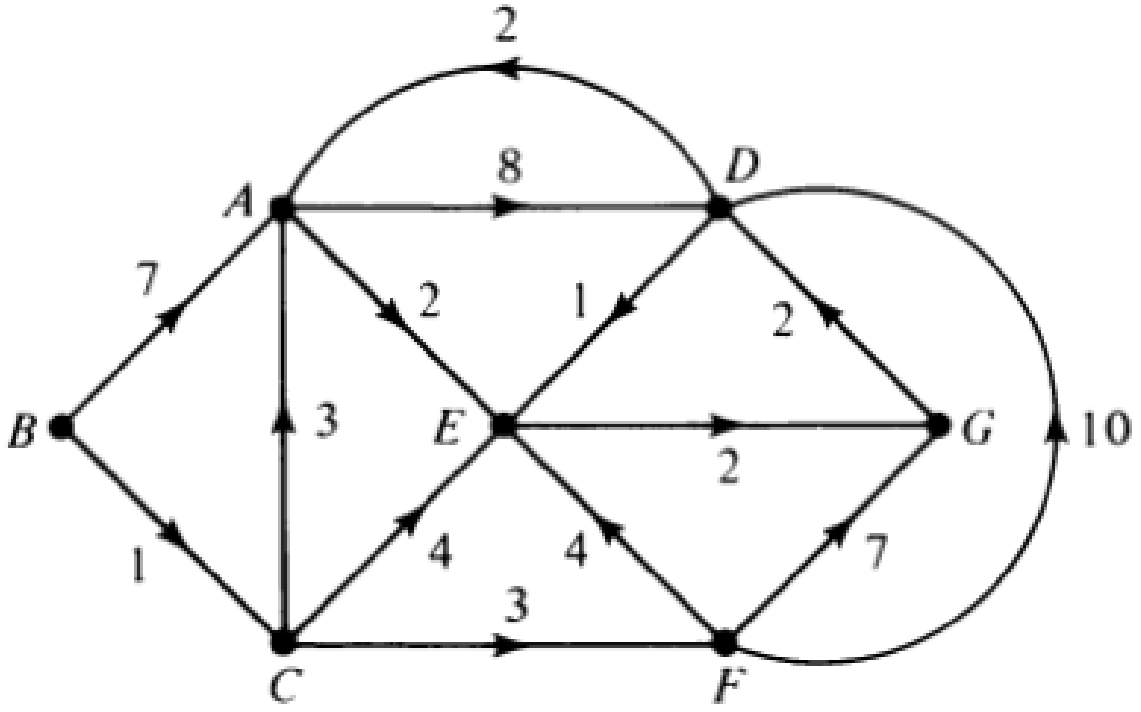
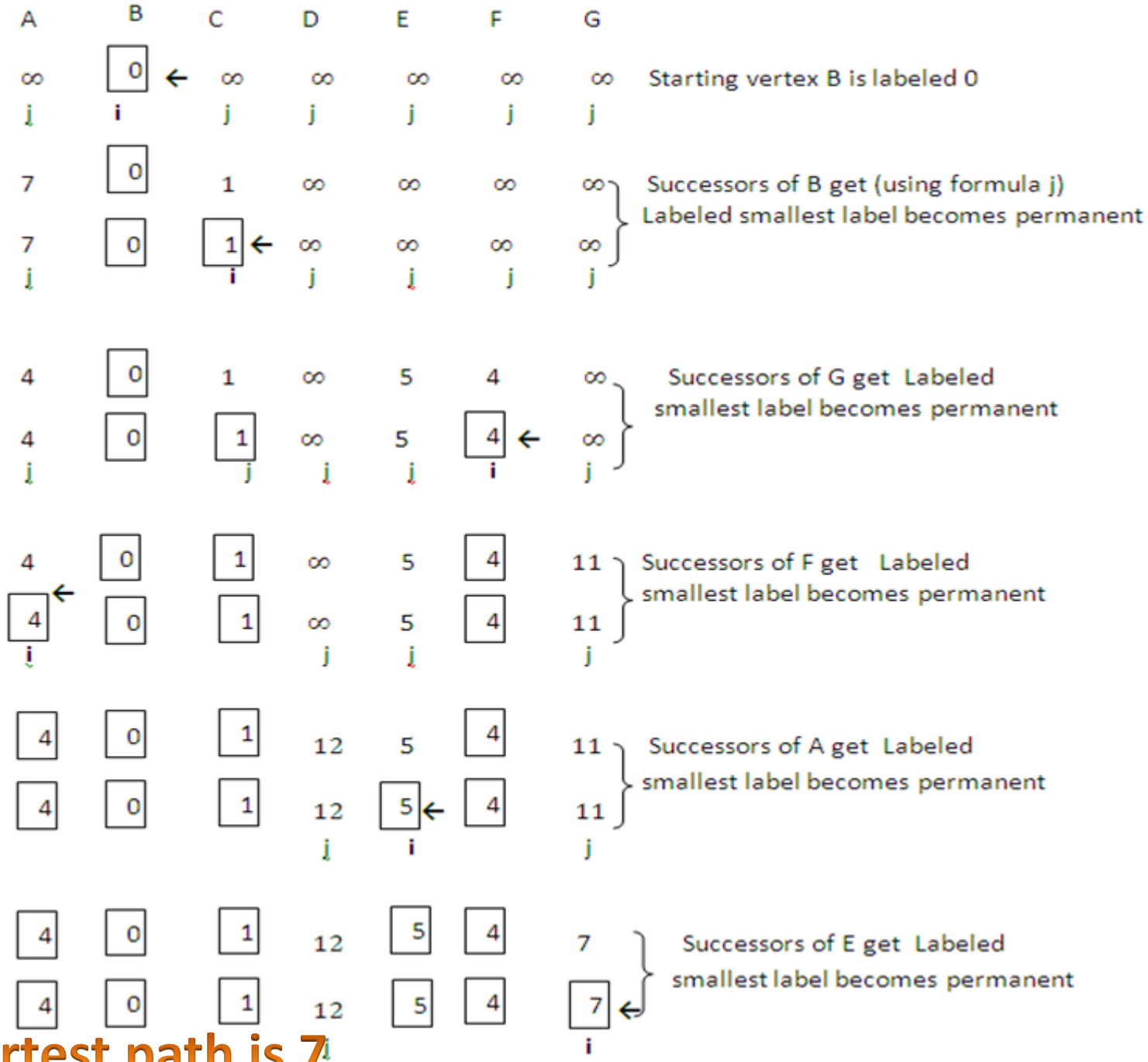


Fig. Simple weighted digraph.

The permanent labels will be shown enclosed in a square, and the most recently assigned permanent label in the vector is indicated by a tick ☐✓ The labeling proceeds as follows:

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.



Shortest path is 7

Shortest path is 7

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 label differ exactly by the length of the connecting edge(A tie indicates more than one shortest path).

$i \leftarrow \min (j, i+d_{ij})$.

Algorithm:

1) for h = 1 to n do

being

label(1) ← ∞

vect (1) ← 0

End

2) Label (s) ← o

vect (s) ← 1

i ← s.

"shortest"

s -----→ t

$j \leftarrow \min (j, i\ d_{ij})$

↑

label (j)

3) $M \leftarrow \infty$

in

for j=1 to n do

begin

if (vect(j) ≠ 1)

$z \leftarrow label(i) + d_{ij}$

if z < label (j) then

label(j) ← z

if (label (j) ≤ M

in

M ← label(j)

in

p ← j

end

4) Vect (p) ← 1

if (p≠ t) { i ←p ; go to step (3)

5) Label t ← label (t)

6) Print label t

7) stop.

Shortest path between all pair of vertices

Warshall - Floyd Algorithm

Prof. K.Maheswaran

Warshall - Floyd Algorithm:-

- Starting with the n by n (n x n) matrix D=[dij] of direct distances, n different matrices $D_1, D_2, \ldots, D_n$ they are constructed sequentially.
- Matrix D_k, $1 \leq k \leq n$ may be through of as matrix whose (i,j) entries gives the length of shortest directed path among all directed path from i to j, with vertices 1,2,…..,k. allowed as the intermediate vertices.

Matrix $D_k = [d_{ij}^{(k)}]$ is constructed from D_{k-1} occurring to the following rule.

$$d_{ij}^{(k)} = \min [d_{ij}^{(k-1)}, ([d_{ik}^{(k-1)} + d_{kj}^{(k-1)}])]$$

where k=1,2,……..n

$$d_{ij}^{(0)} = d_{ij}$$

That is iteration1, vertex1 is inserted in the path from vertex i to j if $d_{ij} > d_{i1}+d_{1j}$, In iteration 2, vertex 2 is inserted.

Suppose for example let as shortest directed path from 7 to 3 is 7 4 1 9 5 3

The following replacement occurs

Interation1: $d_{49}^{(0)}$ is replaced by ($d_{41}^{(0)} + d_{19}^{(0)}$)

Interation4: $d_{79}^{(0)}$ is replaced by ($d_{74}^{(3)} + d_{49}^{(3)}$)

Interation5: $d_{49}^{(0)}$ is replaced by ($d_{95}^{(4)} + d_{53}^{(4)}$)

Interation9: $d_{49}^{(0)}$ is replaced by ($d_{79}^{(8)} + d_{93}^{(8)}$)

Once the shortest distance is obtained in $d_{73}^{(9)}$ the value of this entry will not altered in subsequence operation.

Shortest path between every vertex pair:

```
for k ← 1 to n do
    for i ← 1 to n do
     if d_ik < > ∞
                then
                        for  j ← 1 to n do
                        if d_kj <> ∞
                                then
                                    min
                                        $≤ d_ik + d_kj
                                            min
                                      if $ < d_ij then
                                            min
                                       d_ij ← $
```

(or)

```
for k ← 1 to n do
        for i ← 1 to n do
        for j ← 1 to n do
                dij ← min {dij, dik + dkj} k = 1
```